SERTİFİKAT QEYDLƏRİM

JAVA SE

Oracle Certified Associate,
Java SE Programmer
imtahanına hazırlaşanlar üçün yardımçı vəsait

Müşfiq Məmmədov

SERTİFİKAT QEYDLƏRİM

Oracle Certified Associate, Java SE Programmer imtahanına hazırlaşanlar üçün yardımçı vəsait: istiqamət və xülasələr

Müşfiq Məmmədov

Bakı – 2018

ISBN 978-9952-37-127-7

Çapa hazırlanmışdır: 18.11.2018
Format: 60x84

MÜNDƏRİCAT

Müəllif haqqında **11**

Kitab haqqında **12**

Necə ərsəyə gəlib? 12

Hansı ədəbiyyatlardan istifadə olunub? 13

Kimlər üçün faydalı ola bilər? 14

Rəy və təkliflər 15

Bölmə 1. Sertifikat İmtahanına Hazırlıq üzrə İstiqamətlər **17**

Addım 1. İmtahanın seçilməsi 18

Addım 2. Hazırlıq kitabının alınması 21

Addım 3. Coderanch forumunda qeydiyyatdan keçmək 23

Addım 4. Test bankının seçilməsi və alınması 26

Addım 5. İmtahan üçün qeydiyyatdan keçmək 32

Addım 6. İmtahan günü 41

Addım 7. İmtahanın nəticəsini öyrənmək 42

Addım 8. Sertifikatın elektron və çap versiyalarını əldə etmək 45

Bölmə 2. İmtahan Mövzuları ilə Bağlı Xülasələr **49**

Chapter 1. Java Building Blocks **51**

Comments 51

Main method 52

Redundant Imports 53

Naming Conflicts 53

Package 54

Code formatting on the Exam 55

Constructors 55

Instance Initializer Blocks 55

Order of Initialization 56
Primitive types 57
Declaring Multiple Variables 61
Identifiers 62
Local Variables 63
Instance and Class Variables 64
Variable Scope 65
Ordering Elements in a Class 65
Garbage Collection 66
Əlavələr 74

Chapter 2. Operators and Statements 75

Understanding Java Operators 75
Numeric Promotion 76
Logical Complement and Negation Operators 78
Increment and Decrement Operators 78
Casting Primitive Values 79
Compound Assignment Operators 79
Logical Operators 80
Equality Operators 81
The if Statement 82
The if-then Statement (dangling else) 83
Ternary Operator 85
The switch statement 86
Compile-time Constant Values 87
The while Statement 93
The do-while Statement 93
The for Statement 94
The for-each statement 96

Adding Optional Labels 98
The break statement 100
The continue statement 102
Unreachable and dead code 103
Əlavələr 104

Chapter 3. Core Java APIs 109

Creating and Manipulating Strings 109
Concatenation 109
Immutability 110
The String Pool 110
Important String Methods 110
Method Chaining 115
StringBuilder Class 115
Mutability and Chaining 116
Creating a StringBuilder 116
Important StringBuilder Methods 117
StringBuilder vs. StringBuffer 120
Understanding Equality 121
Understanding Java Arrays 126
Sorting Arrays 128
Searching 129
Creating a Multidimensional Array 129
Using a Multidimensional Array 131
Understanding an ArrayList 131
ArrayList Methods 132
Wrapper Classes 137
Autoboxing 140
Converting Between array and List 141

Sorting list....142
Working with Dates and Times....143
Creating Dates and Times....143
Manipulating Dates and Times....145
Working with Periods....147
Formatting Dates and Times....148
Əlavələr....151

Chapter 4. Methods and Encapsulation....153

Designing Methods....153
Access modifiers....154
Optional Specifiers....154
Return Type....155
Method Name....155
Working with Varargs....155
Applying Access Modifiers....156
Designing Static Methods and Fields....160
Calling a Static Variable or Method....160
Static vs Instance....161
Static Imports....163
Final Initialization....164
Passing Data Among Methods....166
Overloading Methods....168
Overloading and Varargs....169
Autoboxing....169
Primitive and Reference Types....170
Putting It All Together....170
Creating Constructors....174
Default Constructor....175

Overloading Constructors....176
Order of Initialization....177
Encapsulating Data....182
Creating Immutable Classes....183
Writing Simple Lambdas....184
Lambda Example....185
Lambda Syntax....187
Predicates....192
Əlavələr....197

Chapter 5. Class Design....203

Introducing Class Inheritance....203
Extending a Class....203
Applying Class Access Modifiers....204
Creating Java Objects....205
Defining Constructors....205
Understanding Compiler Enhancements....207
Reviewing Constructor Rules....208
Calling Constructors....209
Calling Inherited Class Members....210
super() vs super....212
Overriding a Method....213
Redeclaring private Methods....221
Hiding Static Methods....221
Overriding vs. Hiding Methods....222
Creating final methods....223
Inheriting Variables....224
Hiding Variables....224
Creating Abstract Classes....225

Defining an Abstract Class....226
Creating a Concrete Class....227
Extending an Abstract Class....228
Implementing Interfaces....230
Defining an Interface....231
Inheriting an Interface....233
Classes, Interfaces, and Keywords....234
Abstract Methods and Multiple Inheritance....234
Interface Variables....236
Default Interface Methods....238
Default Methods and Multiple Inheritance....241
Static Interface Methods....241
Understanding Polymorphism....243
Casting Objects....246
Virtual Methods....250
Polymorphic Parameters....251
Əlavələr....253

Chapter 6. Exceptions....257

Understanding Exception Types....257
Throwing an Exception....258
Using a try Statement....259
Adding a finally Block....260
Catching Various Types of Exceptions....261
Throwing a Second Exception....263
Runtime Exceptions....265
Checked Exceptions....267
Errors....268
Calling Methods That Throw Exceptions....269

Subclasses....271
Printing an Exception....272
Əlavələr....274

Bölmə 3. İmtahan Təcrübəm....281

Hazırlığa necə başladım və hansı kitabdan istifadə etdim?....282
Kitabı bir dəfə oxuyub bitirdikdən sonra hansı qərara gəldim?....284
Hansı test bankından istifadə etdim?....287
İmtahan günü – 94% nəticə ilə imtahanı keçdim....288
Jeanne Boyarsky və Scott Selikoff`un kitabı haqqında fikirlərim....289
Enthuware test bankı haqqında qeydlərim....291
Coderanch forumu haqqında düşüncələrim....294
Real imtahan sualları haqqında qısa qeydlərim....295
SYBEX və Enthuware testlərində rastlaşmadığım sual tipləri....296
Məni ən çox çaşdıran 4 sual....297

Müəllif haqqında

Müşfiq Məmmədov – 1986-cı ildə anadan olub. 2004-2008-ci illərdə Azərbaycan Dövlət İqtisad Universitetinin Beynəlxalq iqtisadi münasibətlər fakültəsində bakalavr təhsili alıb. 2008-2009-cu illərdə hərbi xidmətdə olub. 2009-2011-ci illərdə Qafqaz Universitetində Beynəlxalq iqtisadi münasibətlər ixtisası üzrə magistr təhsili alıb.

Əmək fəaliyyətinə 2010-cu ildə Dövlət Statistika Komitəsində iqtisadçı kimi başlayıb, amma sonra proqramçı kimi davam edib. Daha sonra "Azərikard" şirkətində proqramçı kimi çalışıb. Hazırda isə "Azerconnect" şirkətində proqramçı kimi fəaliyyətini davam etdirir.

Java proqramlaşdırma dili ilə 2013-cü ildən məşğul olmağa başlayıb. Boş vaxtlarında şəxsi blogu üçün məqalələr yazmağı sevir. Ailəlidir, iki övladı var.

Blog: http://www.mycertnotes.com/

Linkedin: https://www.linkedin.com/in/mushfiq-mammadov-10461354/

Kitab haqqında

Necə ərsəyə gəlib?

Kitabın yazılması ilə bağlı əvvəlcədən hər hansı bir plan mövcud deyildi. Belə bir qərara sertifikata hazırlıq prosesinin ortalarında gəldim. Sertifikata hazırlıq üçün aldığım kitabı təkrarlamağa başlayanda mövzuların vacib hissələrini qeyd kimi dəftərçəmdə yazmağa başlamışdım. Sonradan qərara gəldim ki, qeydlərimi dəftərdə deyil, bir az daha artıq əziyyət çəkərək word faylda yazım. Və belə olan halda bu qeydləri gələcəkdə kiçik kitabça halına salaraq digərlərinin da faydalana bilməsi üçün paylaşmaq mümkün olsun. Beləcə, kitab hazırlamaq ideyasının təməli qoyuldu.

Əvvəlcə bir bölmə nəzərdə tutmuşdum, sırf sertifikat imtahanına düşəcək mövzular ilə bağlı. Amma facebookda "Java sertifikat sualları" ilə bağlı qrupu yaratdıqdan sonra ünvanlanan çoxsaylı fərdi suallardan başa düşdüm ki, ümumiyyətlə imtahan prosesi barədə informasiya qıtlığı var, imtahan vermək istəyənlər haradan və necə başlayacaqlarını bilmirlər. Özüm də imtahana hazırlaşmaq istədiyim vaxtlarda bu çətinliklərin hamısı ilə qarşılaşmışdım. Gah imtahan verənlərdən, gah forumlardan, gah da Oracle`ın rəsmi saytından imtahan barədə bacardığım qədər informasiya toplamağa can atırdım. İmtahan prosedurları barədə yetəri qədər məlumat almaq müəyyən qədər vaxt və əziyyət hesabına başa gəlmişdi. Ona görə də mövcud tələbatı nəzərə alaraq imtahan bələdçisi hazırlamaq və bütün bu addımları bir mənbədə cəmləmək qərarına gəldim. Və bundan sonra kitabça bir balaca da böyüdü və iki bölmədən ibarət oldu: imtahana hazırlıq addımları və imtahan mövzularının xülasəsi.

Ünvanlanan digər bir qrup suallar isə imtahana hazırlaşarkən necə bir yol izlənilməsi ilə bağlı idi. Daxil olan əsas suallar təxminən aşağıdakı məzmunda idi:

- Hazırlıq üçün hansı kitabı seçməliyəm?
- Tək bu kitab kifayət edəcəkmi?
- Kitabı oxuyub bitirdikdən sonra yenidən təkrarlamaq lazımdırmı yoxsa birbaşa test banklarındakı suallara keçmək olar?
- Testləri mövzular üzrə ardıcıl etməliyəm ya qarışıq?
- Özümü imtahana hazır hesab edə bilərəmmi?
- İmtahan üçün nə vaxt qeydiyyatdan keçmək məsləhətdir və s.

Sırf bu səpkili suallara cavab olsun deyə qərara aldım ki, öz imtahan təcrübəmi üçüncü bölmə kimi əlavə edim. Bu bölmədə imtahana hansı ardıcıllıqla hazırlaşmağım, necə bir yol izləməyim, həmçinin istifadə etdiyim kitabların, test banklarının keyfiyyəti ilə bağlı fikirlərim əks olunacaq.

Kitabı hazırlayarkən qonorar əldə etmək məqsədi nəzərdə tutmamışdım, yazılmasında bir sıra səbəblər var idi. Əsas səbəblərdən biri də sertifikat almaq arzusunda olan insanları öz arzu və məqsədlərinə doğru bir addım daha yaxınlaşdırmaq idi. Bu baxımdan, kitaba qiymət təyin edilərkən bu amil nəzərə alınacaq və bütün təbəqədən olan yoldaşların kitabı əldə edə bilməsi məsələsinə diqqət ediləcəkdir.

Bu mənim ilk kitabım olacaq və hal-hazırda kitabın həm korreksiya işləri, həm texniki redaktəsi, həm də mətnlərin dizaynı özüm tərəfindən həyata keçirilib. Təbii ki, tək başına keyfiyyətli bir məhsul ortaya çıxarmaq çox çətindir, xüsusilə də bu sadalanan sahələrin hər biri üzrə peşəkar deyilsinizsə. Ona görə də birinci nəşrdən sonra bildiriləcək rəylər əsasında kitab daha da təkmilləşdiriləcək və növbəti nəşr üçün daha faydalı və optimal formaya gətiriləcəkdir.

Hansı ədəbiyyatlardan istifadə olunub?

Birinci və üçüncü bölmələr şəxsi təcrübələrim əsnasında formalaşan məlumatlar əsasında hazırlanıb. İkinci - "İmtahan mövzuları ilə bağlı xülasələr" bölməsi isə imtahana hazırlaşdığım dövrdə istifadə etdiyim ədəbiyyatlardan götürdüyüm qeydlər əsasında yazılıb. Ədəbiyyatlar üzrə əsasən vacib hesab edilən hissələr seçilərək qeydlər aparılıb. Bu səbəbdən bəzi mövzular siyahıda olmaya bilər.

Kitabın yazılmasında istifadə edilən əsas ədəbiyyat Jeanne Boyarsky və Scott Selikoff`un müəllifləri olduğu *"OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808"* kitabıdır. Bu kitab imtahana hazırlıq üçün istifadə etdiyim "study guide"dır və ikinci bölmə bu kitabın mündəricatına uyğun ardıcıllıqla hazırlanıb (başlıqların tərcüməsi məqsədəuyğun hesab edilmədiyindən olduğu kimi saxlanılıb). İkinci bölmənin təqribən yarıdan çoxunu bu ədəbiyyatdan götürülən mövzular əhatə edir və istifadə edilən şəkillər (screenshot) və kod nümunələrində əsasən bu kitaba istinad edilib və ya bu kitabdan alıntıdır.

Bundan əlavə Mala Gupta`nın "OCA Java SE 7 Programmer I Certification Guide: Prepare for the 1ZO-803 exam", Kathy Sierra və Bert Bates`in "OCA/OCP Java SE 7 Programmer I & II Study Guide (Exams 1Z0-803 & 1Z0-804)" kitablarından, onlara aid olan test sualı nümunələrindən, imtahan üçün mövcud olan test banklarındakı maraqlı suallar və incə nüanslardan (xüsusilə Enthuware), java quiz`lərdən, stackoverflow, coderanch forumlarından və digər internet resurslardan istifadə olunmuşdur. Həmçinin öz imtahan təcrübəmdə rastlaşdığım maraqlı situasiyalar, vacib məqamlar da kitaba əlavə edilmişdir.

Kitabda imtahan mövzuları bölməsinin daxilində rastlaşacağınız "bu mövzu imtahana düşəcək yaxud düşməyəcək" tipli fikirlər oxuduğum mənbələr və öz imtahan təcrübəmə istinadən qeyd edilib.

Kitab daha böyük kütlə üçün anlaşıqlı olsun deyə elmi dildən bir az kənar, nisbətən danışıq dilinə yaxın bir üslubda yazılmışdır. Proqramlaşdırma üzrə bəzi terminlər ana dilimizə tərcümə edilməyib, ingilis dilində - orijinal versiyada saxlanılıb ki, başa düşmək nisbətən asan olsun. Qeyd edilən bəzi proseslər, prosedurlar öz imtahan təcrübəmdən real nümunələr verilməklə izah edilmişdir ki, bu da həmin məsələlərin daha aydın başa düşülməsi niyyəti ilə edilib. Ümumiyyətlə, kitabın metodik vəsaitdən daha çox praktik vəsait olmasına çalışılmışdır.

Bir çox kod nümunələri oxuduğum orijinal mənbələrdə hansı formadadırsa, kitabda da o formada saxlanılıb. Çünki qeydlərimdə izahlar daha öncədən həmin kod nümunələri üzərindən şərh edildiyinə görə, kod nümunələrini və izahlarını sonradan dəyişdirmək xeyli zaman alacaqdır. Bildiyiniz kimi daha keyfiyyətli iş daha çox zaman tələb edir. Kitabın birinci nəşrinin gecikməməsi üçün imtahan mövzuları ilə bağlı bəzi əlavə qeydlər də həmçinin kitaba daxil edilməyib, ikinci nəşr zamanı daxil edilməsi nəzərdə tutulub. Lakin qeyd edilən bütün bu məsələlər kitabın birinci nəşrindən sonra, oxuculardan gələn "feedback"lər əsasında nəzərə alına və növbəti nəşr üçün dəyişdirilə bilər.

Kimlər üçün faydalı ola bilər?

Kitabın birinci bölməsi - uzun müddətdir sertifikat almaq arzusunda olan, amma haradan və necə başlayacağını bilməyən yoldaşlar üçün nəzərdə tutulmuşdur. Bu bölmədə başlanğıcdan sona bütün mərhələlər addım-addım izah edilmişdir.

Kitabın ikinci bölməsi – OCA 8 imtahan mövzuları əsasında hazırlanıb və əsasən mövzuların xülasəsini əhatə edir. Ona görə də sırf sıfırdan sertifikat imtahanına hazırlıq üçün nəzərdə tutulmayıb, adından da göründüyü kimi bu kitab köməkçi vəsaitdir. İkinci bölmə hazırlanarkən əsas təyinat olaraq – sertifikat imtahanı ilə bağlı hər hansı bir “study guide” oxuduqdan sonra, imtahan öncəsi mövzuları sürətli şəkildə təkrarlamaq üçün nəzərdə tutulmuşdur. Sertifikat imtahanı mövzuları içərisində elə xırdalıqlar mövcuddur ki, kitabı təkrar oxumadan uğurlu nəticə əldə etmək ehtimalı azdır. Orijinal kitaba nisbətən daha yığcam formada və üstəgəl öz ana dilimizdə mövzuları təkrar etmək həm vaxta qənaət etmək, həm də daha yaxşı mənimsəmə baxımından optimaldır. Bu nöqteyi-nəzərdən kitabın ikinci bölməsi sertifikat imtahanına hazırlaşan namizədlər üçün faydalı ola bilər.

Kitabda bir neçə mənbəyə istinad edildiyinə görə hazırlaşdığınız “study guide”da olmayan bəzi xırdalıqlara burada rast gələ bilərsiniz. Bu yolla da kitabda olmayan, amma imtahana düşə biləcək bəzi mövzularla bağlı özünüzü sığortalamış olacaqsınız.

Kitab həmçinin Java SE barədə yetərli nəzəri bilikləri olan və yaxud Java SE ilə bağlı ingilis dilində hər hansı bir kitab oxumuş şəxslər üçün də imtahana hazırlıq baxımından faydalı ola bilər. Amma Java SE ilə bağlı nəzəri baza bilikləri olmayan şəxslər üçün kitab müəyyən qədər çətin gələ bilər. Çünki kitabda əsasən sertifikatla bağlı məqamlara toxunulur və anlayışların, terminlərin izahına, detallarına geniş yer verilmir və sizin həmin anlayışları daha öncədən bildiyiniz fərz edilir. Amma yetəri nəzəri biliklәriniz olmasa belə kitabı oxumaqla sertifikat imtahanı mövzuları və sualları ilə əlaqədar özünüz üçün müəyyən bir təsəvvür formalaşdıra bilərsiniz.

Digər bir tərəfdən, əgər intervüyə dəvət almısınızsa və qısa müddət ərzində Java SE ilə bağlı təməl mövzuları təkrarlayıb yadınıza salmaq istəyirsinizsə, kitab köməyinizə çata bilər.

Əlavə olaraq vurğulamaq istəyirəm ki, imtahan ingilis dilində olduğundan, ingilis dili biliklərinizin olması labüddür. Bu kitabı oxumadan öncə sertifikat imtahanı ilə bağlı ingilis dilində hər hansı bir "study guide" oxumanız məqsədəuyğundur.

Rəy və təkliflər

Kitab ilə bağlı rəy və təkliflərin bildirilməsi üçün blogumda xüsusi bir səhifə ayrılmışdır:

http://www.mycertnotes.com/az/kitablarim/java-sertifikat-imtahani-qeydlerim/

Bu səhifədə kitab ilə bağlı bütün mənfi və müsbət fikirlərinizi, rastlaşdığınız xətaları qeyd edə bilərsiniz. Daxil olan bütün "feedback"lər gözdən keçiriləcək və kitabın növbəti nəşrində nəzərə alınacaqdır.

Oxucuların işini asanlaşdırmaq üçün kitabda mövcud olan linklərin siyahısı blogda ayrıca bir post altında səhifələr üzrə cəmləşdirilmişdir. Artıq linkləri əl ilə daxil etməyinizə ehtiyac qalmayacaq, aşağıdakı linkə daxil olub müvafiq səhifədəki linki taparaq klikləyib baxa biləcəksiniz:

http://www.mycertnotes.com/az/links/

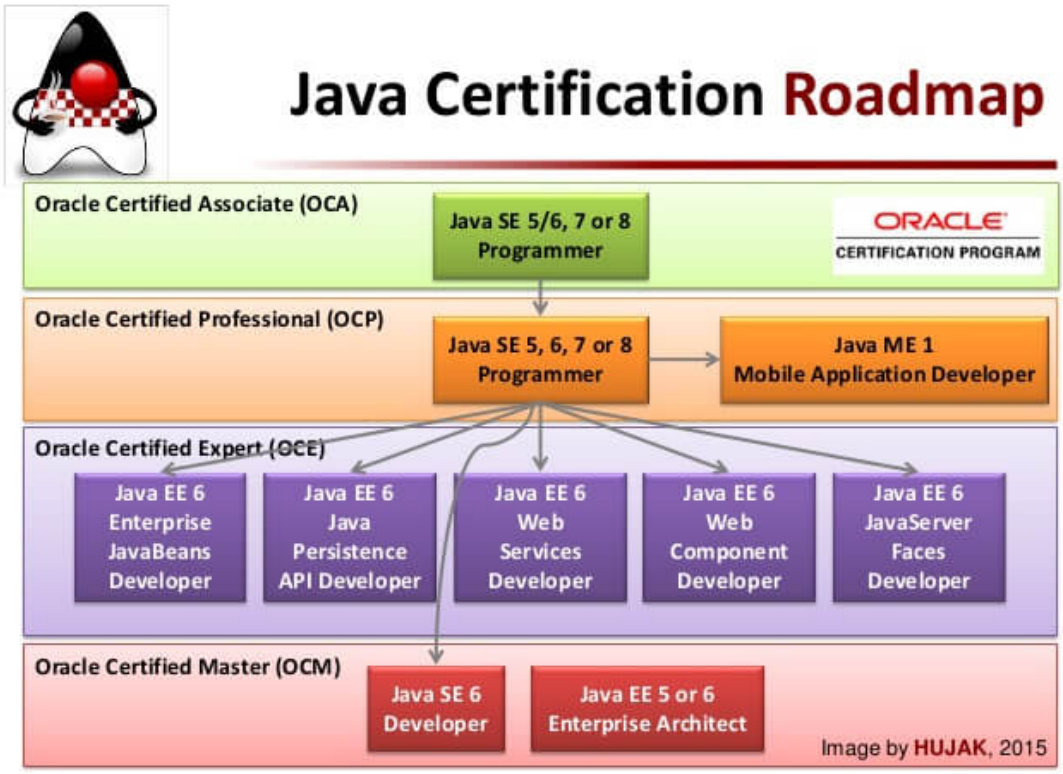
Əgər sizin də oxucu kimi işinizi asanlaşdıracaq hər hansı bir rəy, təklifiniz olarsa, bildirməkdən çəkinməyin.

Bölmə 1. Sertifikat İmtahanına Hazırlıq üzrə İstiqamətlər

- **Addım 1. İmtahanın seçilməsi**
- **Addım 2. Hazırlıq kitabının alınması**
- **Addım 3. Coderanch forumunda qeydiyyat keçmək**
- **Addım 4. Test bankının seçilməsi və alınması**
- **Addım 5. İmtahan üçün qeydiyyatdan keçmək**
- **Addım 6. İmtahan günü**
- **Addım 7. İmtahanın nəticəsini öyrənmək**
- **Addım 8. Sertifikatın elektron və çap versiyalarını əldə etmək**

Addım 1. İmtahanın seçilməsi

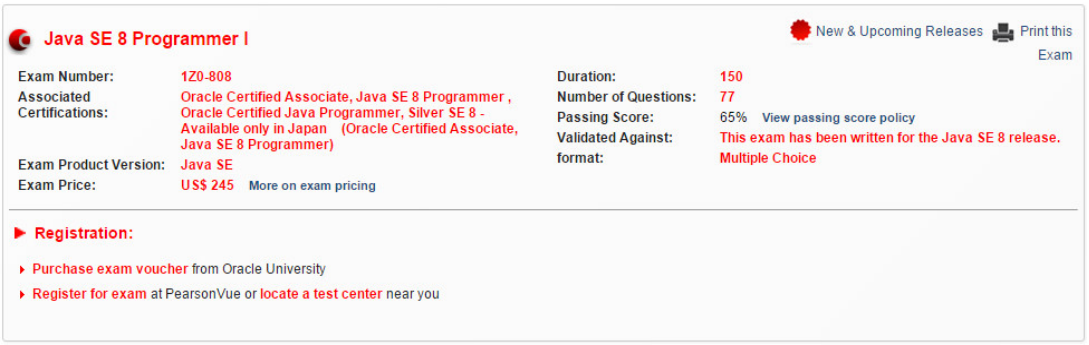
İlk öncə hansı imtahanı vermək istədiyinizə qərar verməlisiniz. Əgər indiyə kimi sertifikatınız olmayıbsa, ilkin olaraq Java Standard Edition üzrə imtahan verməlisiniz. Aşağıdakı şəkildə imtahanların iyerarxiyası aydın şəkildə göstərilib:



Şəkildən də göründüyü kimi ilk öncə OCJP 6 (1Z0-851), OCA 7 (1Z0-803), OCA 8 (1Z0-808) və s. imtahanlardan birini seçə bilərsiniz. OCJP 6 imtahanının OCA 7 və OCA 8 imtahanlarından fərqi ondadır ki, siz ancaq bir imtahan verirsiniz, həmin imtahanı keçdikdən sonra artıq EE mərhələsi üzrə imtahana daxil ola bilərsiniz. Amma 7-ci versiyadan sonra SE pilləsi üzrə imtahanlar iki hissəyə ayrılıb: **Associate** və **Professional.** EE pilləsi üzrə imtahana daxil olmaq üçün həm Associate, həm də Professional imtahanından keçməlisiniz.

Amma 31.05.2018-ci il tarixindən etibarən Oracle OCJP 6 imtahanını ləğv edib. Artıq OCA 7 və OCA 8 imtahanlarından birini seçə bilərsiniz. Lakin 31.12.2018-ci il tarixindən OCA 7 (1Z0-803) imtahanının Oracle tərəfindən ləğv edilməsi nəzərdə tutulub (Mənbə: http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=206). Bütün imtahanlarla bağlı Oracle`ın rəsmi internet səhifəsində ətraflı məlumat vardır.

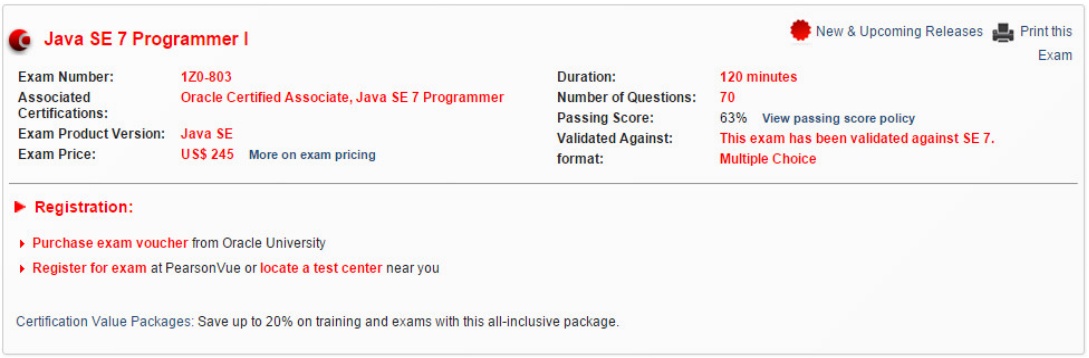
Şəkil 1. 1Z0-808 imtahanı ilə bağlı 14.01.2016-cı ilə olan məlumat:



1Z0-808 imtahanı ilə bağlı son məlumatları aşağıdakı linkdən izləyə bilərsiniz:

https://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=5001&get_params=p_exam_id:1Z0-808

Şəkil 2. 1Z0-803 imtahanı ilə bağlı 14.01.2016-cı ilə olan məlumat:



1Z0-803 imtahanı ilə bağlı son məlumatları aşağıdakı linkdən izləyə bilərsiniz:

https://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=5001&get_params=p_exam_id:1Z0-803

OCA SE 7 imtahanı ilə OCA SE 8 imtahanı arasındakı fərqləri şəkillərdən (şəkillər yenilənməyib, çünki məlumatlar tez-tez dəyişir, ona görə də linklərə klikləyərək son məlumatlara baxa bilərsiniz) görə bilərsiniz (imtahan üçün ayrılan vaxt, sual sayı, keçid faizi). Amma əsas fərq mövzular ilə əlaqəlidir, daha dəqiq OCA SE 8 imtahanına yeni mövzular daxil edilib:

- *static and default interface methods;*
- *Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period;*

- *Write a simple Lambda expression that consumes a Lambda Predicate expression.*

Amma *time* paketi və *lambda* ilə bağlı təxminən 4-5 sual düşür. Mövzu yükü elə də böyük deyil. Mən OCA SE 8 imtahanını seçməyinizi məsləhət görərdim. Ona görə ki, bu imtahan 8-ci versiyada gəlmiş bu yenilikləri öyrənməyinizə vəsilə olacaq, digər tərəfdən də 2018-ci ilin sonunda OCA SE 7 imtahanının müddəti bitəcək. Ümumiyyətlə, imtahana düşəcək bütün mövzuların siyahısına isə yuxarıda qeyd olunmuş linklərdə *"Exam Topics"* başlığına klikləməklə baxa bilərsiniz.

Şəkildə gördüyünüz imtahanın qiyməti (**$245**) sizi qorxutmasın. Oracle`ın ölkələr üzrə fərqli qiymət siyasəti var, Azərbaycan üçün imtahanın qiyməti **$150**-dır. İmtahanın dəqiq qiymətini http://www.pearsonvue.com/oracle/ saytında qeydiyyatdan keçərkən öyrənə bilərsiniz. Endirimlər vaxtına təsadüf etsə, bu qiymət daha da aşağı ola bilər. Mən 2015-ci ilin noyabr ayında imtahana girmişdim və 2015-ci ildə Java`nın 20 illiyi münasibəti ilə Oracle bütün java imtahanlarına ilin sonuna kimi *20%* endirim etmişdi. Ona görə də mən $150 əvəzinə $120 ödəmişdim. 2016-cı ildə isə ilin ikinci yarısından ilin sonunadək Oracle təxminən 10-dan çox imtahan üçün 35% endirim etmişdi. Endirimlərdən hansı formada yararlanmaq barədə 5-ci addımda – imtahandan necə qeydiyyatdan keçmək mərhələsində qeyd etmişəm.

Addım 2. Hazırlıq kitabının alınması

İmtahanı seçdikdən sonra növbəti addım həmin imtahana hazırlaşmaq üçün optimal kitabın seçilməsidir. Kitabın doğru seçilməsi çox vacibdir, əks təqdirdə həm uğursuz nəticə əldə edə, həm də vaxtınızı əbəs yerə itirə bilərsiniz. Kitab imtahan mövzularını tam əhatə etməli, mövzuları konkret izah etməli, suallar və kod nümunələrini real imtahan suallarına bənzər stil/formatda verməli, artıq mövzularla sizi yükləməməli və bir sözlə imtahan prosesini düzgün işıqlandırmalıdır.

OCA Java SE 7 imtahanına hazırlaşmaq üçün aşağıdakı kitablar mövcuddur:

1. "OCA Java SE 7 Programmer I Certification Guide: Prepare for the 1Z0-803 exam", **Mala Gupta**
2. "OCA/OCP Java SE 7 Programmer I & II Study Guide (Exams 1Z0-803 & 1Z0-804)", **Kathy Sierra**, **Bert Bates**
3. "Oracle Certified Associate, Java SE 7 Programmer Study Guide", **M. Reese Richard**
4. "OCA Java SE 7 Programmer I Study Guide (Exam 1Z0-803)", **Edward Finegan**, **Robert Liguori**
5. "Oracle Certified Associate, Java SE 7 Programmer Exam (1Z0-803) *Complete Video Course*", **Simon Roberts**

1-ci və 2-ci kitablar oxucular tərəfindən böyük rəğbətlə qarşılanıb və hal-hazırda OCA SE 7 imtahanından uğurla keçənlərin əksəriyyəti təcrübələrini paylaşarkən bu kitablardan hazırlaşdıqlarını qeyd edirlər. Əgər OCA SE 7 imtahanı verəcəksinizsə, bu iki kitabdan birini seçməyiniz tövsiyə olunur. Mala Gupta`nın müəllifi olduğu kitabın sonunda bir "full mock exam", Kathy Sierra və Bert Bates`in müəllifləri olduqları kitabın əlavəsində (CD) isə OCA imtahanı ilə əlaqəli iki "mock exam" var.

OCA SE 8 imtahanına hazırlaşmaq üçün isə aşağıdakı kitablar mövcuddur:

1. "OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808", **Jeanne Boyarsky**, **Scott Selikoff**
2. "OCA Java SE 8 Programmer I Certification Guide", **Mala Gupta**
3. "OCA Java SE 8 Programmer I Exam Guide (Exams 1Z0-808)", **Kathy Sierra, Bert Bates**
4. "OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808) (Oracle Press)", **Edward Finegan**, **Robert Liguori**

5. "A Programmer's Guide to Java SE 8 Oracle Certified Associate (OCA)", **Khalid A. Mughal, Rolf W Rasmussen**
6. "OCA Java SE 8 Programmer I (1Z0-808) *Complete Video Course*", **Simon Roberts**

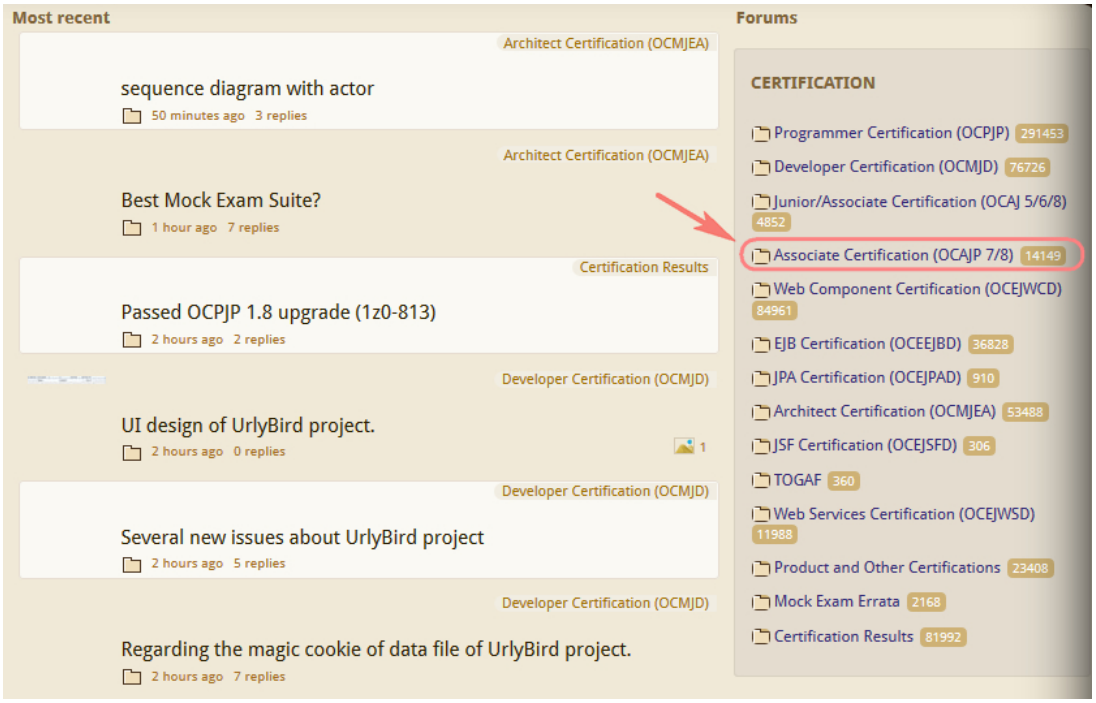
Mən 1-ci kitabdan hazırlaşmışam və bu kitabı seçməyə qərar verənə qədər bir xeyli tərəddüd etmişdim. Əvvəlcə OCA SE 7 imtahanını verməyi qərara almışdım və Mala Gupta`nın "OCA Java SE 7 Programmer I Certification Guide: Prepare for the 1ZO-803 exam" kitabından hazırlaşmağı planlaşdırmışdım. Amma hazırlığa başlamaq istədiyim ərəfələrdə OCA SE 8 imtahanının beta versiyası çıxdı. Fikrimi dəyişdim, OCA SE 8 imtahanını verməyi qərara aldım. Sertifikat imtahanına qəti şəkildə hazırlaşmağı qərara aldığım vaxt OCA SE 8 üzrə cəmi bir hazırlıq kitabı var idi, Jeanne Boyarsky və Scott Selikoff`un adını yuxarıda qeyd etdiyim kitabı. Kitab yenicə çıxmışdı və Amazonda yaxud digər forumlarda həmin kitabı oxuyub fikir bildirən hələ olmamışdı. Mala Gupta`nın OCA 8 ilə bağlı kitabının yaxın zamanlarda çıxıb-çıxmayacağı ilə bağlı sualıma cavab ala bilmədiyimdən qeyd etdiyim bu kitabı araşdırmağa başladım. Başqa alternativim yox idi və bu kitabı aldım. Kitab gözlədiyimdən də yaxşı çıxdı. Kitabla bağlı fikirlərimi ətraflı şəkildə "İmtahan Təcrübəm" bölməsində qeyd etmişəm. Qısacası, heç bir tərəddüd etmədən bu kitabı alıb imtahana hazırlaşmağa başlaya bilərsiniz.

Əlavə olaraq qeyd edim ki, sertifikatla bağlı hazırlıq kitablarının demək olar ki, hamısı ingilis dilindədir. İmtahan ingilis dilində olduğundan ingiliscə hazırlaşmaq daha məqsədəuyğundur. Ola bilər ki, ingilis dili bilginiz zəif olsun, amma bu sizi qorxutmasın. Haşiyəyə çıxaraq öz təcrübəmdən bir məqamı paylaşmaq istəyirəm. Dil ilə bağlı qorxu hissini mən də yaşamışam. Sertifikata hazırlaşmaq üçün ilk kitabı hələ 2014-cü ilin yayında almışdım. Amma hər dəfə kitabı əlimə götürəndə gözlərim böyüyürdü, 10 dəqiqədən artıq oxuya bilmirdim. 8 il olardı ingilis dili ilə mütəmadi məşğul olmurdum və bildiklərimin əksəriyyətini unutmuşdum. Az-çox xatırladıqlarım məktəb vaxtında öyrəndiklərim idi. Bu qorxu, bu minvalla sertifikata hazırlıq təqribən 9 ay yubandı, ta ki, qarşıma ciddi şəkildə məqsəd qoyanadək. Yeni imtahan və yeni kitab çıxmışdı. Bu dəfə israrlı idim, kitabı aldım və başladım oxumağa. Təbii ki, asan olmadı, çətinlik bir aya qədər davam elədi. Google translate'də danışıq kitabı yaratmışdım, bilmədiyim sözləri ora əlavə edirdim. Müəyyən sayda söz yığılandan sonra danışıq kitabını excel formatına çevirərək çap edirdim. Parkda gəzəndə yaxud axşamlar evdə həmin sözlərə təkrar nəzər yetirirdim. Sonra artıq kitabın dilinə öyrəşə bildim. Sözsüz ki, çətinliklər olacaq, amma əgər məqsədiniz qorxunuzdan böyükdürsə, o zaman kitabı alın və başlayın.

Addım 3. Coderanch forumunda qeydiyyatdan keçmək

Sertifikata hazırlıq dönəmində qəbul etdiyim ən doğru qərarlardan biri coderanch forumuna qoşulmaq oldu. Forumun əhəmiyyətli dərəcədə faydasını gördüm. Sizə də forumdan faydalanmağı tövsiyə edirəm. Forumda OCA SE ilə bağlı bölməyə aşağıdakı linkdən daxil ola bilərsiniz:

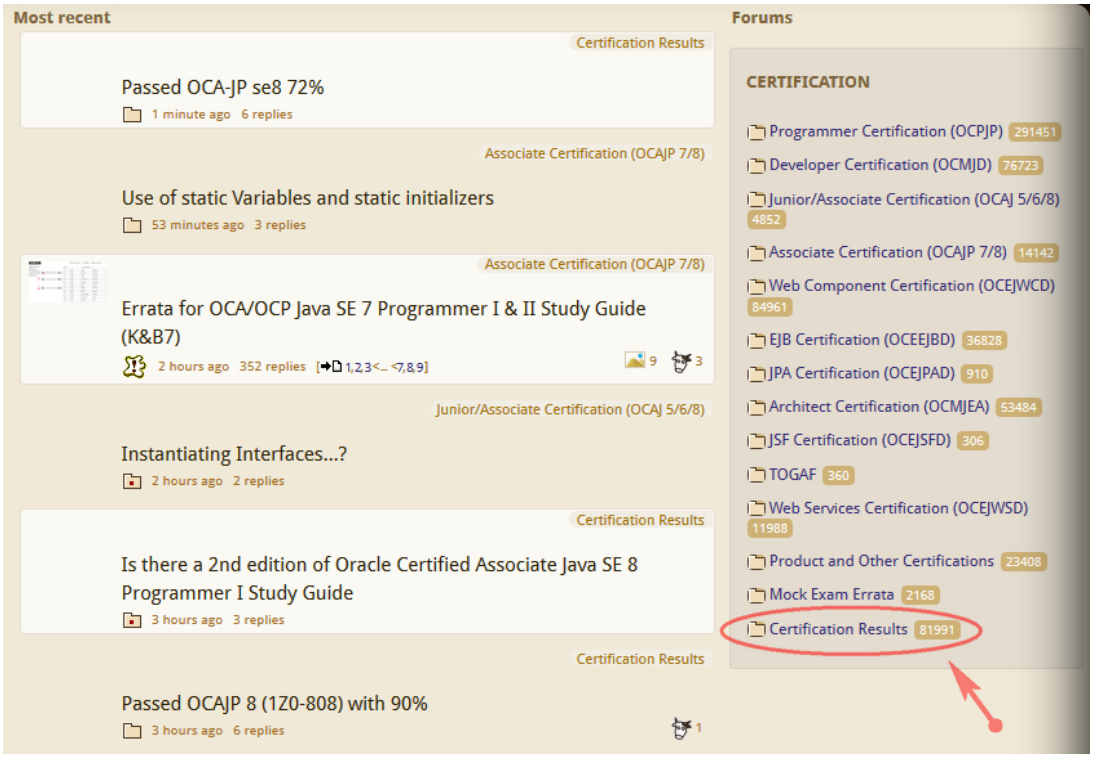
http://www.coderanch.com/forums/c/7/certification



Foruma heç nə yazmasanız belə gündəlik olaraq daxil olub yeni postları oxumağın informasiyalaşmaq baxımından sizə çox böyük faydası olacaq. Forumda üzvlər tərəfindən sertifikata hazırlıq prosesinin hər mərhələsi üzrə suallar soruşulur və moderator və ya digər üzvlər tərəfindən həmin suallar ya ətraflı cavablandırılır, ya da ətraflı məlumat üçün müvafiq linklərə yönləndirilir.

Bundan əlavə *"Certification Results"* bölməsində imtahandan keçmiş istifadəçilər imtahanla bağlı təəssüratlarını, təcrübələrini paylaşırlar. Bu məlumatların da sizə böyük faydası dəyə bilər:

http://www.coderanch.com/forums/f-44/certification-results



Həmçinin "Ocajp Wall Of Fame"də ancaq OCA SE imtahanından keçənlərin təcrübələri ilə tanış ola bilərsiniz:

http://www.coderanch.com/t/659980/Wiki/Ocajp-Wall-Fame

Oracle Certified Associate, Java SE 8 Programmer Wall of Fame [Order by Date]				
Name	**Date**	**Score**	**Story**	**Location**
Jacques Ouellet	February 17, 2016	94%	Took 1 month and 6 days. No exp. in Java before. Material: Book + Enthuware Exams (Except the final)	QC (Canada)
Mahesh Trikannad	February 14, 2016	94%	OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808 Enthuware Mock Exams (Took all tests)	NJ (USA)
Ricard Nàcher Roig	February 11, 2016	97%	OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808 Enthuware Mock Exams (Took 6 tests)	Barcelona (Catalonia)
Chirag Patel	February 09, 2016	94%		India
Jude Niroshan	February 09, 2016	94%	My Story	Sri Lanka
Jan Stückrath	January 26, 2016	95%	See this thread	Düsseldorf, Germany
Mushfiq Mammadov	November 30, 2015	94%	My experience with in detail	Baku, Azerbaijan
Heng Chong Ming	January 21, 2016	97%	OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808 Enthuware Mock Exams (Took all 8 tests)	Singapore
Felipe Kunzler	January 18, 2016	97%	Passed OCAJP 8 with 97%	Brazil
Ben Pittens	January 18, 2016	72%	My story	Netherlands

Bundan əlavə forum ingilis dilinizi inkişaf etdirmək baxımından da sizə faydalı ola bilər. Forumda əksər suallar addım-addım və gözəl şəkildə izah edilir (xüsusilə Roel De Nijs

tərəfindən). Bu da istifadəçilərin onlar üçün qaranlıq qalan suallarını forumda soruşmağa bir stimul, maraq yaradır.

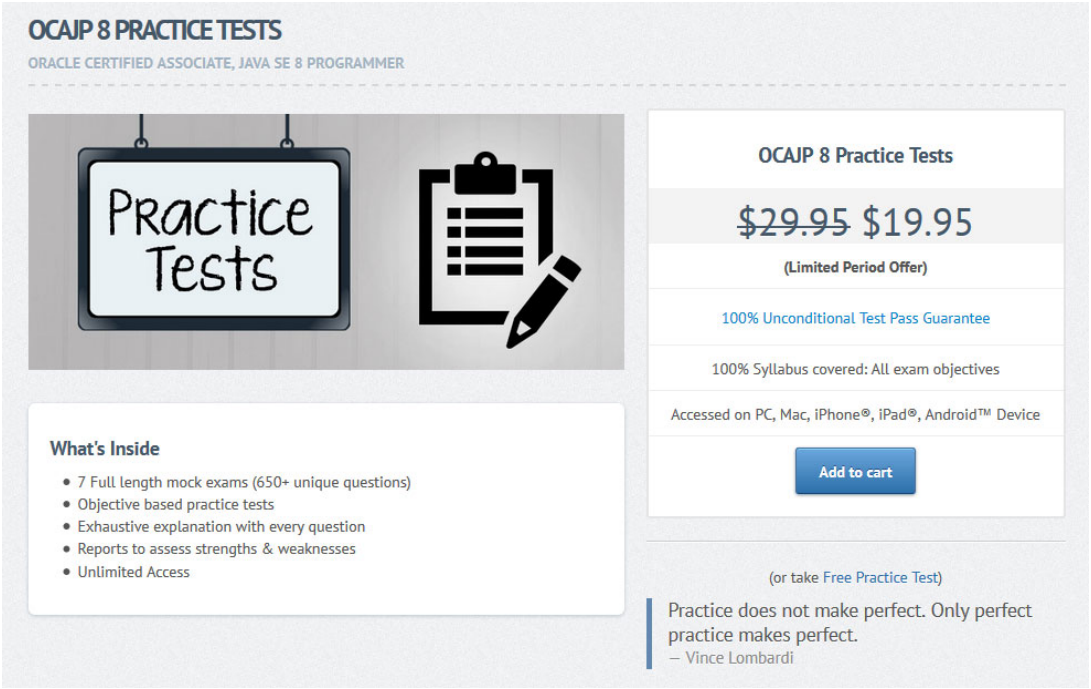
Addım 4. Test bankının seçilməsi və alınması

OCA SE imtahanı üçün müxtəlif test bankları mövcuddur. Hazırlıq boyu bir neçəsinə rastladım və məhsulun keyfiyyəti barədə ümumi təsəvvürüm olsun deyə nümunə test suallarını etdim. Həmin test bankları ilə bağlı təəssüratlarım belə oldu:

Whizlabs

Suallar çox asan idi, nümunə sualların içərisində qaranlıq qalan, çətin sual olmadığından çox dərininə getmədim və izahlara da baxmadım. Whizlabs`ın nümunə sualları ilə real imtahan suallarını müqayisə etdikdə bu test bankının yetəri qədər faydalı olacağını hesab etmirəm.

OCAJP 8 imtahanı üçün test bankı **7 mock exam**`dən və ümumilikdə **650+** sualdan ibarətdir. Qiyməti isə **$19.95** *(endirimsiz $29.95)*-dır.



Nümunə suallar və digər əlavə məlumatlarla aşağıdakı linkdən tanış ola bilərsiniz:

http://www.whizlabs.com/oracle-certified-associate-java-se-8-programmer/ocajp8-free-test/

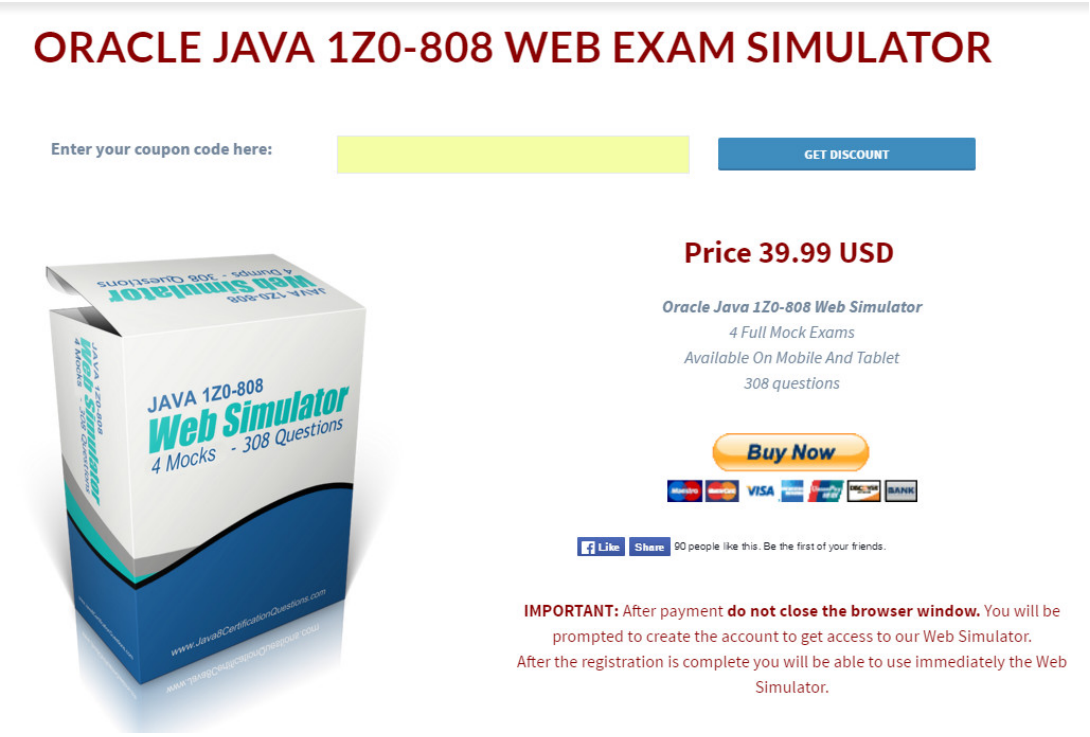
David Mayer's Java8CertificationQuestions sample

Bu test bankı ilə bağlı təəssüratlarım da yaxşı olmadı:

- 25 nümunə sualdan təxminən 4-nün cavabı yanlış idi;
- iki sual eyni idi, təkrarlanırdı;
- suallardan birində 3 cavab bəndi tələb olunsa da doğru cavabların sayı 2 idi;

- sualların izahlarında bəzi qüsurlar var idi və s.

Test bankı **4 mock exam**`dən və ümumilikdə **308** sualdan ibarətdir. Qiyməti isə **$39.99**-dır.

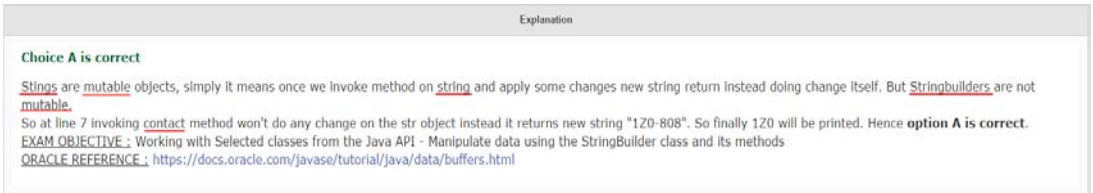


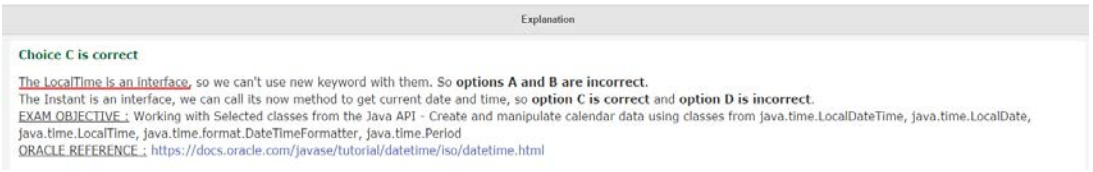
Nümunə suallar və digər əlavə məlumatlarla aşağıdakı linkdən tanış ola bilərsiniz:

https://www.java8certificationquestions.com/java8/1Z0-808/free-test.html?affiliatecode=fcff36fd-557a-4713-abf6-973e9924770f&productid=java8

MyExamCloud

Bəzi suallarda texniki qüsurlar olsa da içərisində maraqlı suallar var. Amma bəzi izahlarda buraxılan ciddi səhvlər istər-istəməz test bankının keyfiyyəti ilə bağlı şübhələr yaradır.





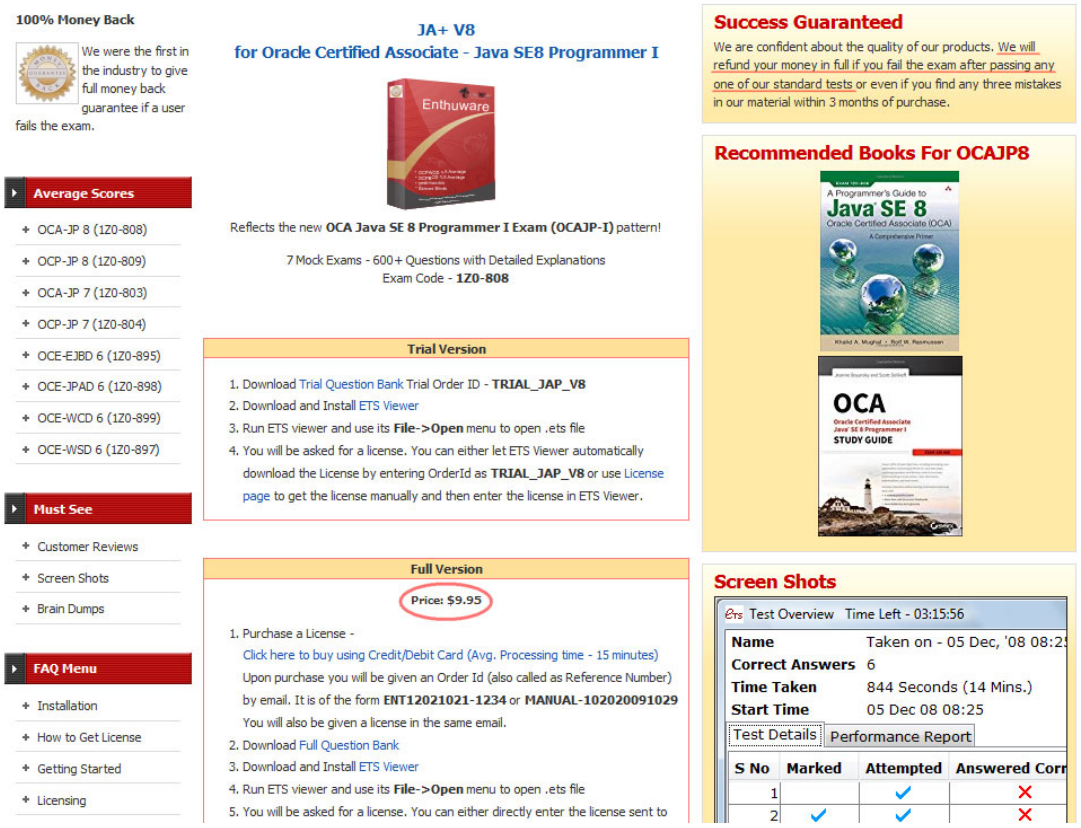
`String` mutable deyil, `StringBuilder` isə mutable`dir. `LocalTime` isə interfeys deyil, classdır. Bunların səhv yazılması test bankının keyfiyyətinə olan inamı azaldır. Test bankının qiyməti hazırda **$19.98**-dır.

Nümunə suallar və digər əlavə məlumatlarla aşağıdakı linkdən tanış ola bilərsiniz:

http://www.myexamcloud.com/onlineexam/viewExam.html?t=Uc1TNwrGb08=

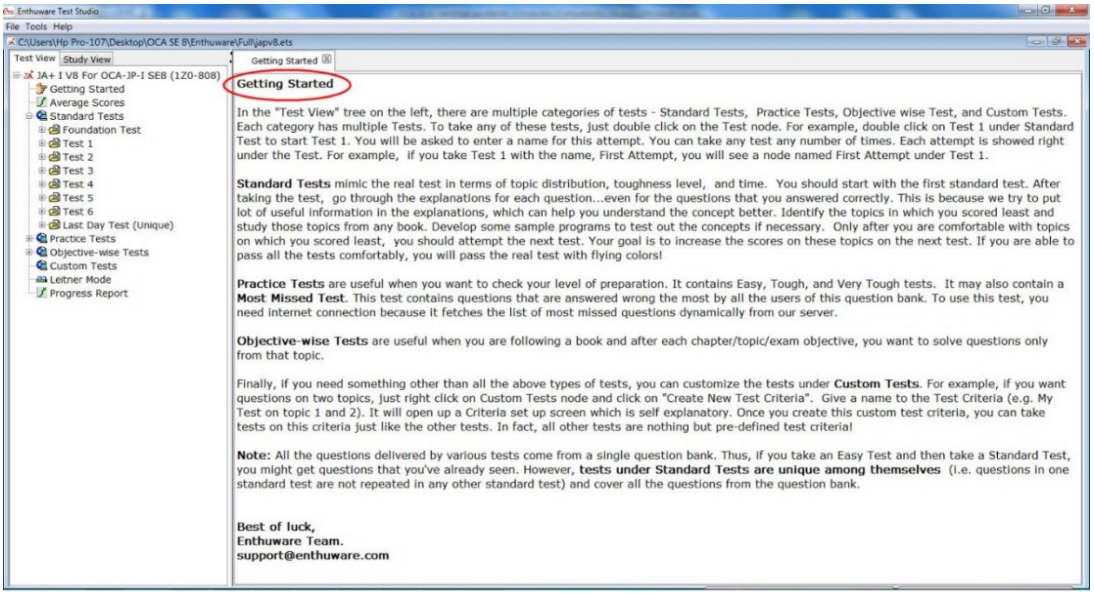
Enthuware

Enthuware`nin bir neçə imtahan üzrə test bankı mövcuddur. Qiymət və keyfiyyət baxımından digər alternativlərindən əhəmiyyətli dərəcədə üstündür. OCA Java SE 8 Programmer I Exam (OCAJP-I) test bankı **7 mock exam** + **Foundation Test**`dən və ümumilikdə **600+** sualdan ibarətdir. Qiyməti isə cəmi **$9.95**-dır və şirkət olaraq da öz məhsullarına tam zəmanət verir. Əgər imtahandan keçə bilməsəniz və ya test bankında hər hansı 3 səhv tapsanız pulunuzu sizə geri ödəyir:



İmtahandan uğurla keçənlərin əksəriyyəti də öz təcrübələrində qeyd edir ki, test bankı olaraq Enthuware`dən istifadə ediblər və razı qalıblar. Mən özüm də bu test bankını alıb istifadə etmişəm və əhəmiyyətli dərəcədə faydasını hiss etmişəm. Ona verilən pulun haqqını artıqlaması ilə ödəyir. Sizə də tərəddüd etmədən bu məhsulu tövsiyə edə bilərəm. Kitabın "İmtahan Təcrübəm" bölməsində Enthuware`nin mənə verdiyi faydalar barədə yazmışam, amma bu məhsulu ilk dəfə istifadə edənlər üçün proqramın imkanları və bir sıra üstünlükləri barədə qeydlərimi də əlavə etmək istəyirəm ki, ondan daha effektli şəkildə yararlana bilsinlər.

Proqramı açdıqda *"Getting Started"* başlıqlı yazıya rast gələcəksiniz:



Bu yazıda proqramın bölmələri haqqında məlumatlar və həmçinin proqramı hansı ardıcıllıqla istifadə etməyiniz barədə tövsiyələr verilir. İlk öncə *"Standard Tests"* bölməsindən başlamağınız məsləhət görülür. Amma testlərə başlamadan öncə mütləq sertifikat imtahanı ilə bağlı hər hansı bir study guide oxumağınız zəruri hesab edilir. Çünki kitab oxumadan testlərə başladıqda bilmədiyiniz sualların sayı çox olacaq. Bu zaman səhv etdiyiniz suallarla bağlı müvafiq mövzuları kitabdan oxuyub bir müddət sonra həmin testləri təkrar etməyə başlayanda, bu suallar artıq sizin üçün effektli olmayacaq. Daha əvvəl bu sualları gördüyünüzə görə cavablar istər-istəməz yadınızda qalacaq və testdən əldə etdiyiniz nəticə sizin real nəticənizi düzgün əks etdirməyəcək. Real nəticəniz adətən birinci cəhddən sonra əldə etdiyiniz nəticə hesab olunur. Sertifikat imtahanı üzrə təcrübəli mütəxəssislər qeyd edir ki, əgər Enthuware Standard Test`lər üzrə ortalama nəticəniz 80 faizin üzərindədirsə, o zaman siz özünüzü imtahana tam hazır hesab edə bilərsiniz. Amma bu 80 faiz nəticə ilk cəhddən sonra əldə etdiyiniz nəticə olsa daha yaxşı olar, əks halda sizi yanılda bilər.

"Standart Tests" bölməsindəki bütün testləri **("Last Day Test"dən başqa)** ardıcıl olaraq edin. Həmin testləri bitirdikdən sonra səhv etdiyiniz mövzuları kitabdan yenidən oxuyun. Oxuduqdan sonra *"Objective-wise Tests"* bölməsindən sırf həmin mövzuya aid testləri edə bilərsiniz. Bundan əlavə *"Practice Tests"* bölməsindəki testlərə də baxa bilərsiniz. Bu bölmədəki testlər çətinlik dərəcəsinə görə ayrı-ayrılıqda qruplaşdırılıb. Hər iki bölmədəki testlər *"Standard Tests"* bölməsindəki testlərin içindən seçilərək təkrarlanır. Ancaq ***"Last Day Test"dəki suallar unikaldır*** və heç yerdə təkrarlanmır. "Last Day Test"i bütün hazırlığı bitirdikdən sonra imtahana təxminən 2 gün qalmış etməniz məqsədəuyğundur. Bu testdəki suallar imtahan suallarına çox bənzəyir və bu testdən əldə etdiyiniz nəticə adətən real imtahan nəticənizə çox yaxın olur.

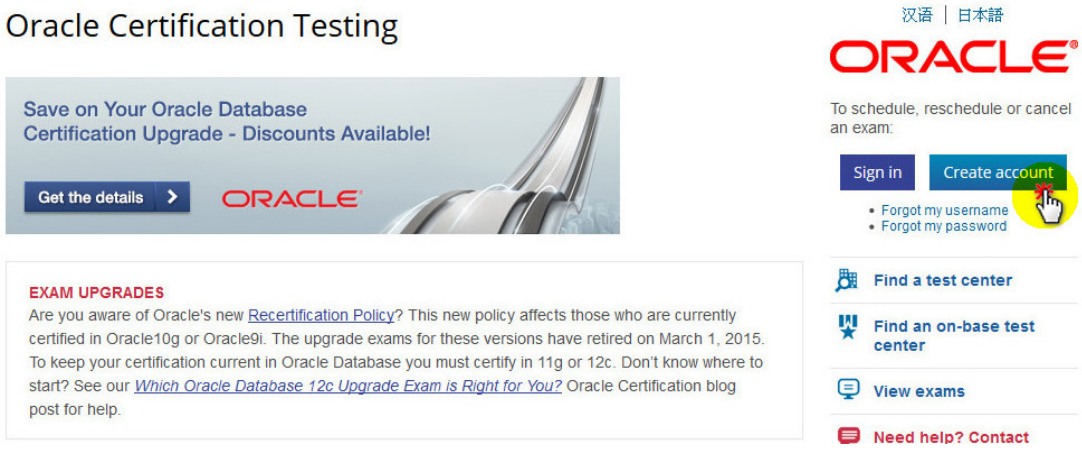
Testlərə başlamadan öncə qeyd edim ki, proqramın strukturu ilk öncə adama primitiv görünsə də istifadəçinin rahatlığı üçün demək olar ki, bütün imkanlar yaradılıb, bu barədə imtahan təcrübəmdə ətraflı məlumat vermişəm.

Əlavə olaraq bildirim ki, sırf sual nümunələri ilə bağlı J.Boyarsky və S.Selikoff`un "OCA/OCP Java SE 8 Programmer Practice Tests" kitabı da mövcuddur. Mən imtahana hazırlaşdığım dönəmlərdə mövcud deyildi, sonradan yazılıb. Ona görə də özüm istifadə etməmişəm. Amma bu kitabı Coderanch forumunda "book promo"da udmuşdum. Alıb vərəqləmişəm, faydalı kitabdır, yararlana bilərsiniz.

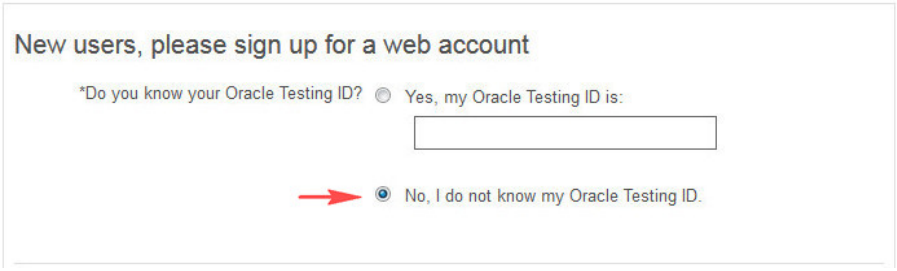
Qeyd. Yuxarıda test bankları haqqında yazdığım fikirlər imtahana hazırlıq dövründə (2015-ci il) rastlaşdığım situasiya və gördüyüm nəticələrə əsasən yazılıb. Hal-hazırda həmin test banklarının keyfiyyətində, sualların sayında, qiymətində və s. dəyişiklik ola bilər. Xüsusilə də son vaxtlar Whizlabs ilə bağlı müsbət "feedback"lər oxuyuram. Ona görə də test bankını seçməmişdən öncə təkrar bir də araşdırmağınız məqsədəuyğundur.

Addım 5. İmtahan üçün qeydiyyatdan keçmək

İmtahana qeydiyyat http://www.pearsonvue.com/oracle/ veb səhifəsi üzərindən aparılır və buna görə də ilk öncə bu saytda qeydiyyatdan keçmək lazımdır:



"Create account" düyməsini sıxdıqdan sonra qeydiyyat formu açılır və ilkin olaraq sizdən *"Oracle Testing ID"*niz soruşulur, ilk dəfə imtahan verirsinizsə, o zaman 2-ci seçimi edirsiniz:

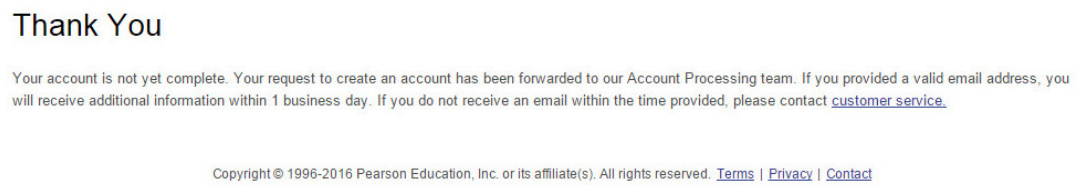


Ad və soyadınızı şəxsiyyət vəsiqənizdə olduğu formada yazmalısınız (ancaq ingilis hərflərindən istifadə etməklə). İmtahan günü test mərkəzinə yaxınlaşdıqda qeydiyyatdan keçərkən qeyd etdiyiniz ad və soyadınızın şəxsiyyət vəsiqənizə uyğun olub olmadığı yoxlanılır. Problemlə rastlaşmamaq üçün hər ehtimala qarşı ad və soyadınızın ingiliscə qarşılığını (şəxsiyyət vəsiqənizə uyğun) düzgün yazmağınız tövsiyə olunur.

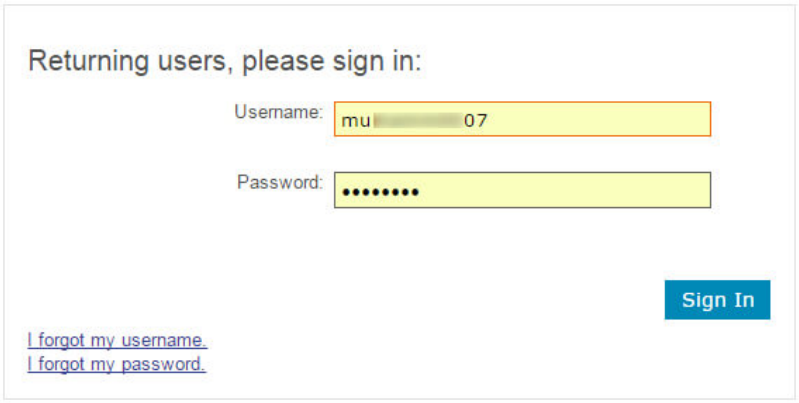
Həmçinin ad və soyad sertifikatın üzərində də qeyd etdiyiniz formada əks olunacaq.

Sonra isə aktiv istifadə etdiyiniz email ünvanınızı daxil edib (doğru yazdığınızdan əmin olun) *"next"* düyməsini sıxın.

Növbəti pəncərədə əlaqə məlumatlarınızı daxil edirsiniz. Sonra *"next"* düyməsini sıxaraq *"Additional Information"* pəncərəsinə keçirsiniz və burada soruşulan suala *"no"* deyərək təkrar *"next"* düyməsini sıxırsınız və prosesi sonlandırırsınız. Ekranda aşağıdakı məzmunda bildiriş görsənəcək:

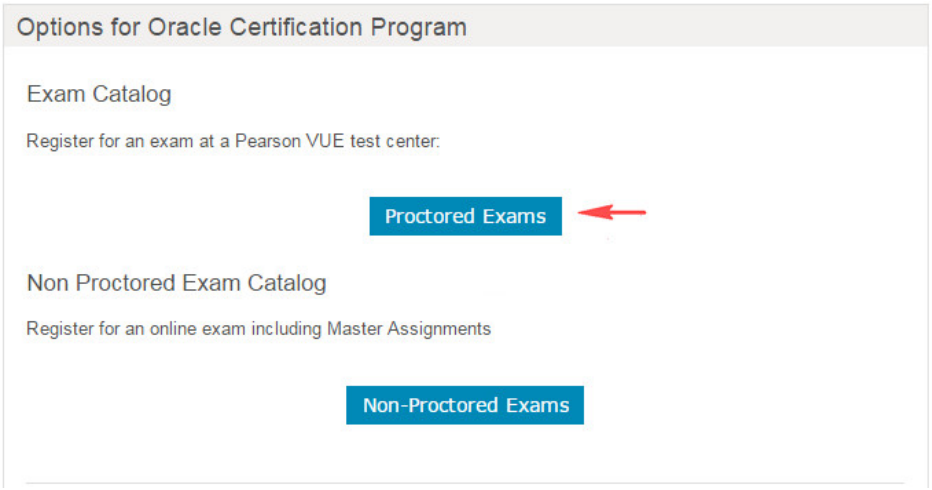


Təxminən bir neçə dəqiqə sonra emailinizə istifadəçi adı və şifrəniz göndəriləcək. Artıq öz hesabınızla sistemə daxil ola bilərsiniz:

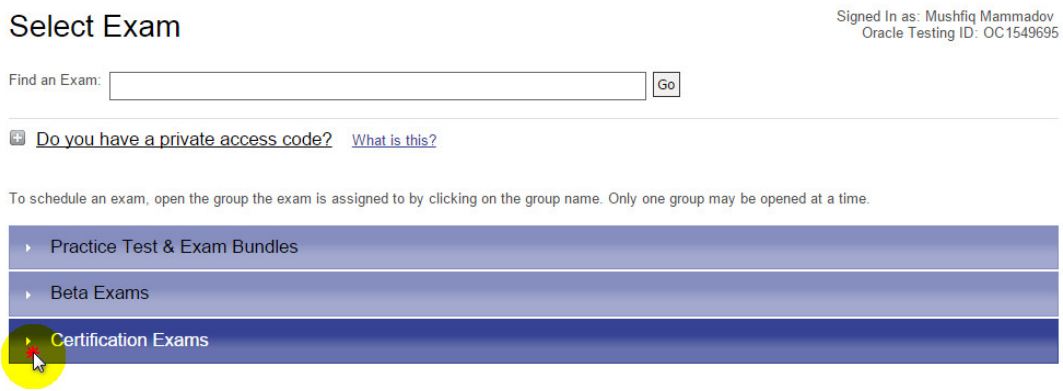


Artıq imtahan üçün qeydiyyata başlamaq olar. Addım-addım gedək.

İlk öncə ana səhifədə *"Proctored Exams"* düyməsini sıxırıq:

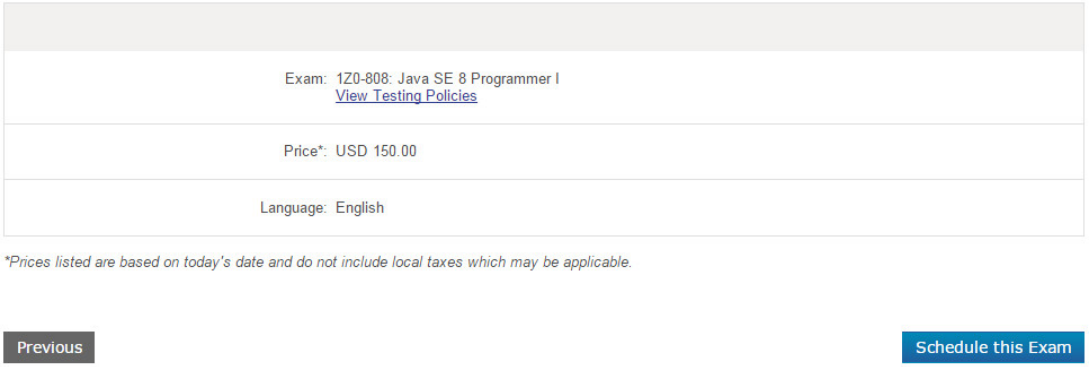


Açılan pəncərədə *"Certification Exams"* düyməsini sıxırıq:



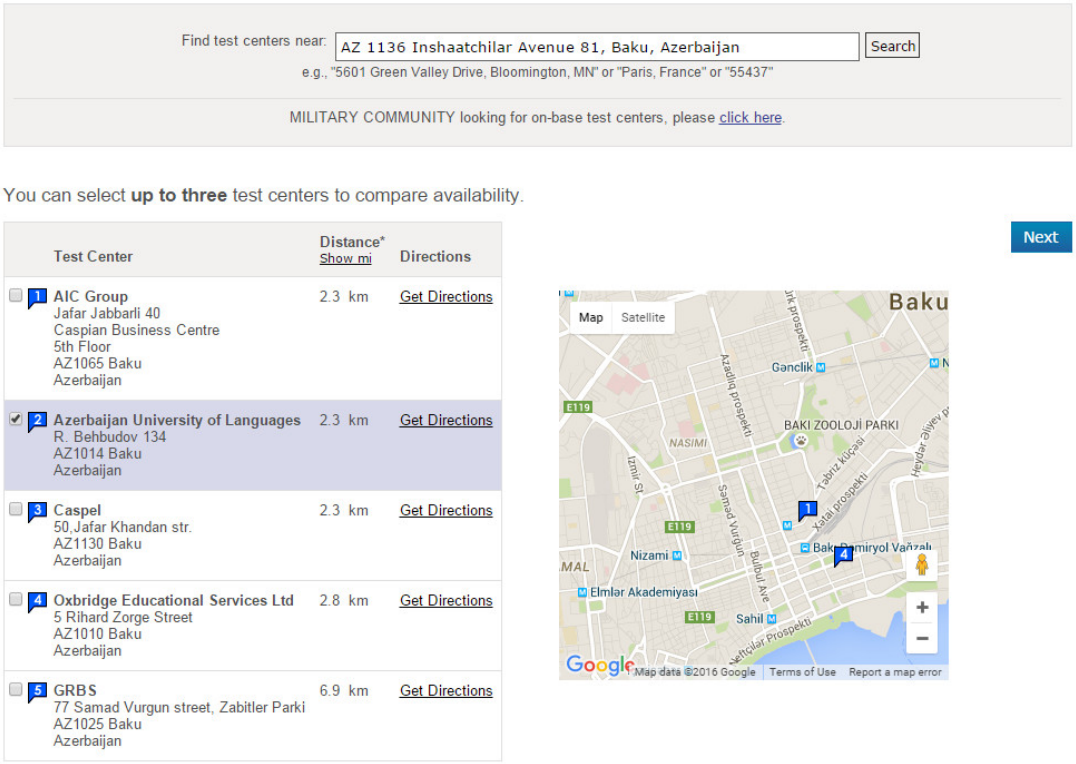
Qeyd olunan düyməni sıxdıqdan sonra bütün imtahanların siyahısı görünür və bu siyahıdan verəcəyimiz imtahanı seçirik:

İmtahanı seçdikdən sonra aşağıdakı pəncərə açılır:



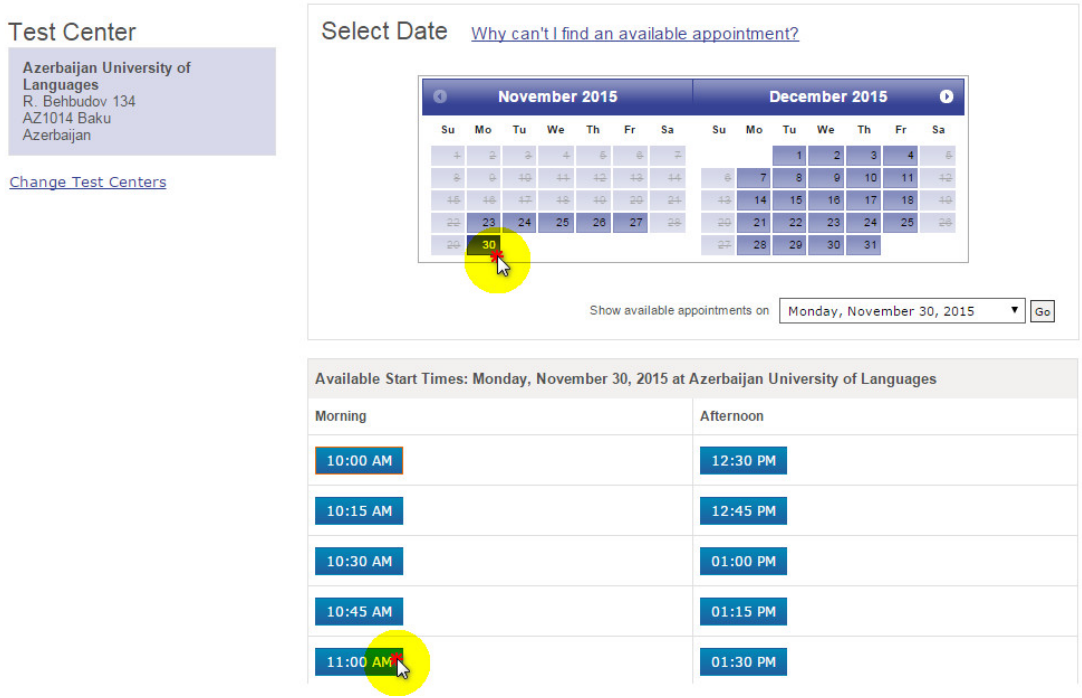
Bu pəncərədə seçdiyimiz imtahanın adı, qiyməti və hansı dildə olması ilə bağlı informasiya göstərilir. Artıq imtahanın vaxtını və yerini təyin edə bilərik. Bunun üçün *"Schedule this Exam"* düyməsini sıxırıq. *"Confirm Exam Selection"* pəncərəsi açılır və seçdiyimiz imtahanı təsdiqləməmiz istənilir. İmtahanı doğru seçdiyimizə əmin olduqdan sonra, davam etmək üçün *"Proceed to Scheduling"* düyməsini sıxırıq.

Bu pəncərədə müvafiq Test Mərkəzlərinin siyahısı açılır:

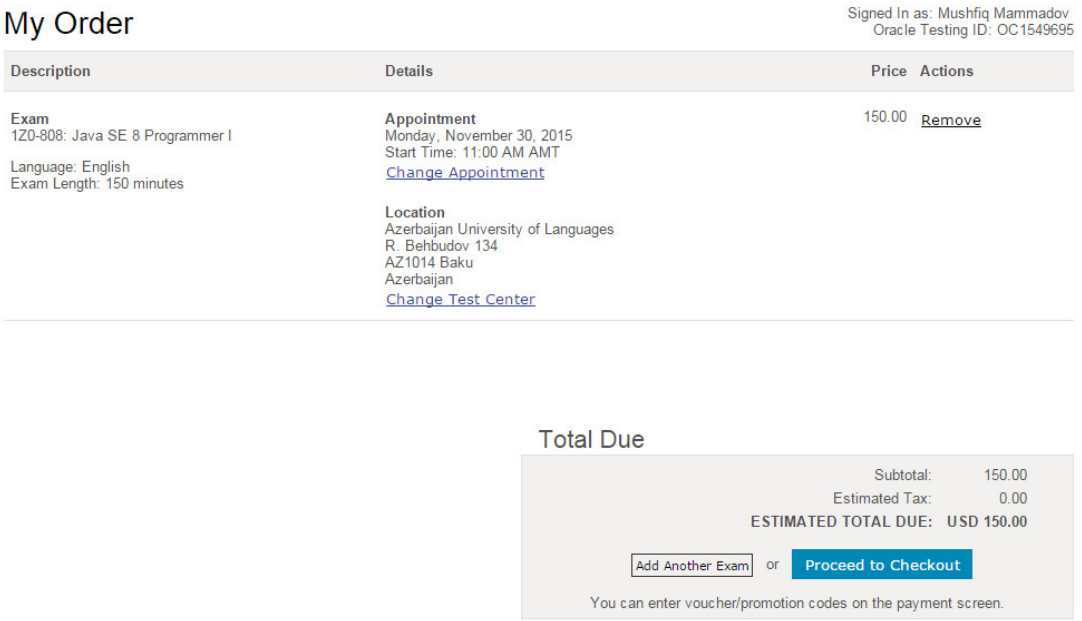


http://www.pearsonvue.com/oracle/ saytında qeydiyyatdan keçərkən ünvan məlumatlarımızı qeyd etdiyimizə görə bu Test Mərkəzlərinin siyahısı da bizim ünvan məlumatlarımıza müvafiq olaraq axtarılıb sıralanır. Test Mərkəzinin seçilməsi vacib mərhələlərdən biridir və imtahan nəticənizə yetərli dərəcədə təsir edə bilər. Ona görə də Test Mərkəzini seçərkən "yüz ölçüb, bir biçmək" lazımdır.

Test Mərkəzini seçdikdən sonra *"next"* düyməsini sıxırıq və imtahan qrafiki açılır:

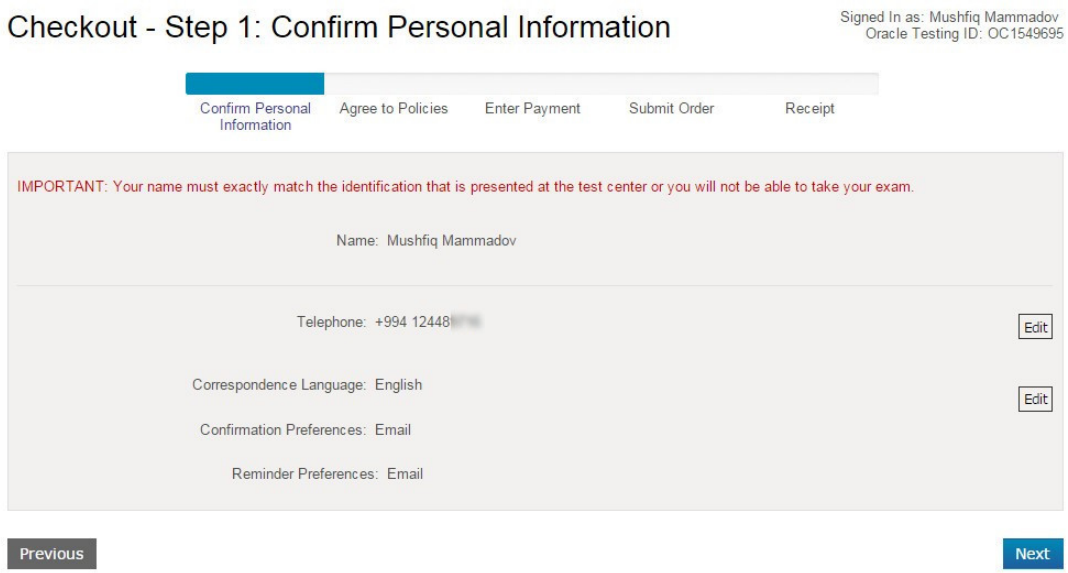


İmtahan qrafikləri test mərkəzlərindən asılı olaraq fərqli-fərqli ola bilər. İmtahan vermək istədiyimiz günü və saatı seçirik. Bundan sonra *"My Order"* pəncərəsi açılır:



Sonuncu *"Checkout"* mərhələsinə keçmədən öncə daxil etdiyimiz məlumatların düzgün olub-olmamasını təkrar yoxlayırıq. Bu mərhələdə imtahan qrafiki və Test Mərkəzini dəyişmək

mümkündür. Hər şeyin qaydasında olduğuna əmin olduqdan sonra *"Proceed to Checkout"* düyməsini sıxırıq və aşağıdakı pəncərə açılır:



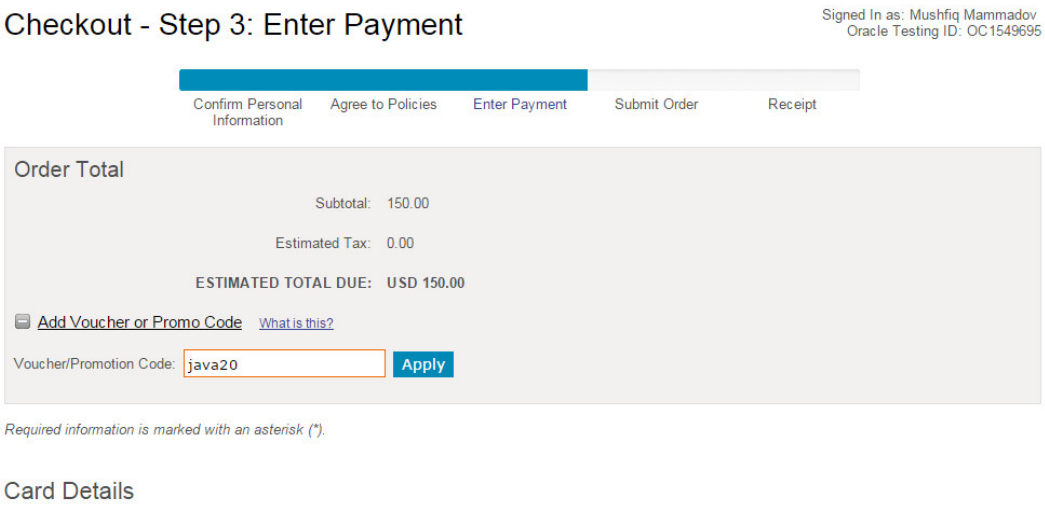
"Checkout" pəncərəsi göründüyü kimi 5 addımdan ibarətdir:

- Step 1. Confirm Personal Information;
- Step 2. Agree to Policies;
- Step 3. Enter Payment;
- Step 4. Submit Order;
- Step 5. Receipt.

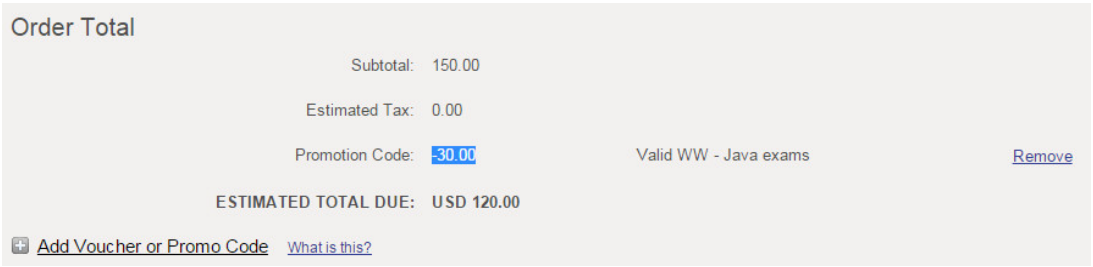
Addım 1`də şəxsi məlumatlarımızla tanış olub *"next"* düyməsini sıxırıq.

Addım 2`də imtahan şərtləri göstərilir və tanış olduqdan sonra razı olduğumuzu qeyd edib *"next"* düyməsini sıxırıq.

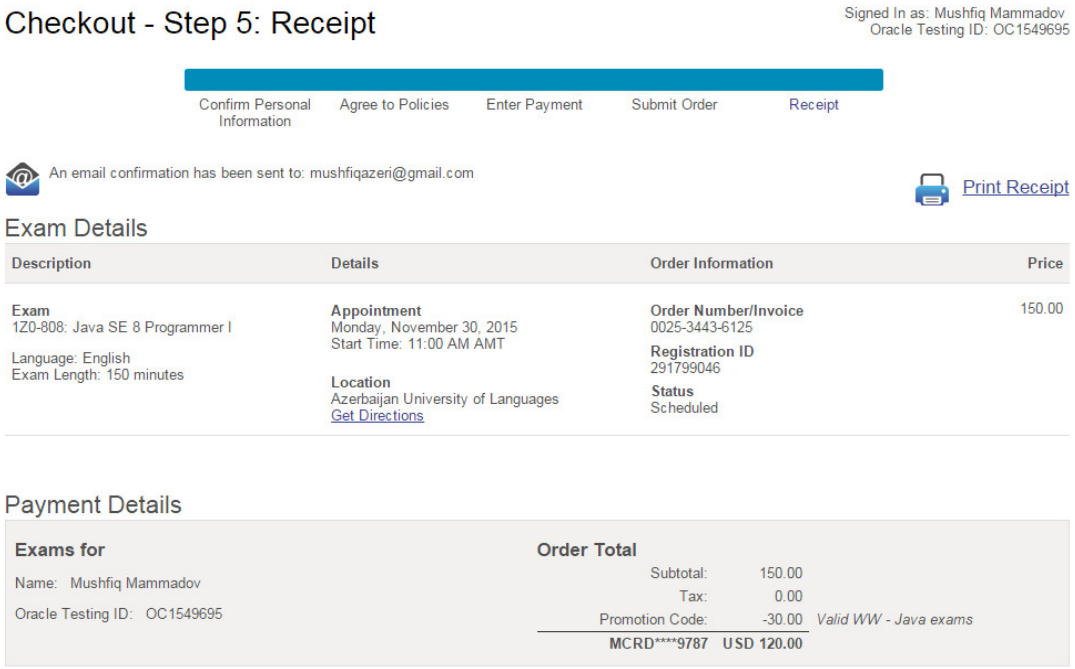
Addım 3`də ödəniş kartı və ünvanla bağlı məlumatları daxil edirik. Bu məlumatları daxil etmədən öncə sizdən *"voucher"* və ya *"promo code"* soruşulur:



Mən imtahan verdiyim il Java`nın 20 illiyinə təsadüf edirdi və bu münasibətlə Oracle bütün java imtahanlarına 20 faiz endirim kompaniyası elan etmişdi. Kompaniyadan yararlanmaq üçün sadəcə ***"java20"*** promo kodunu daxil etmək tələb olunurdu. Promo kodu daxil edib *"apply"* düyməsini sıxdıqdan sonra artıq ödənişin məbləği dəyişərək **$150** əvəzinə **$120** olurdu:



Kart və ünvan məlumatlarını daxil etdikdən sonra *"next"* düyməsini sıxırıq və *"Submit Order"* pəncərəsi açılır. Artıq sifarişimiz tam hazır vəziyyətdədir və təsdiq olunmağını gözləyir. Əgər *"Submit Order"* düyməsini sıxmasaq sifarişimiz tamamlanmayacaq və kartımızdan heç bir məbləğ tutulmayacaq. *"Submit Order"* düyməsini sıxıb sifarişimizi təsdiq edirik:



Sifariş qəbul edildikdən sonra bununla bağlı emailimizə məktub gəlir.

İmtahanın vaxtına 24 saat qalanadək hesabınıza daxil olub *"reschedule"* edərək imtahan verəcəyiniz tarixi dəyişə bilərsiniz. Buna görə əlavə məbləğ və ya cərimə ödəmirsiniz. Amma imtahana 24 saatdan az vaxt qalıbsa, artıq dəyişiklik etmək mümkün deyil və imtahana getməyəcəyiniz təqdirdə pulunuzu itirmiş olacaqsınız.

İmtahana qeydiyyat prosesimin proqram vasitəsilə ekran görüntüsünü almışdım, çünki daha öncədən planlaşdırmışdım ki, bu video görüntünü sertifikat qeydlərimlə bağlı yazacağım kitaba əlavə edəcəm. Qeyd etdiyim həmin bu qeydiyyat prosesinə video görüntü şəklində aşağıdakı linkdən baxa bilərsiniz:

https://youtu.be/DOTfS44bG8o

Addım 6. İmtahan günü

İmtahanı çalışın çox erkən saatlara salmayın ki, gecə yuxunuzu tam alasınız. İmtahandan bir gün öncə yaxşıca dincəlin və imtahan günü səhər yeməyini yeməyi unutmayın (əgər imtahanınızı səhər saatlarına təyin etmisinizsə). Çünki 2-2.5 saat ərzində beyniniz fasiləsiz olaraq işləyəcək, mümkün qədər enerjili və gümrah olmalısınız.

Çalışın imtahan mərkəzinə 30 dəqiqə öncədən gəlin. Özünüzlə şəxsiyyətinizi təsdiq edən 2 sənəd gətirməlisiniz, şəxsiyyət vəsiqəsindən əlavə ikinci rəsmi sənəd də tələb olunur. Bu sürücülük vəsiqəsi, xarici pasport, plastik bank kartı və s. ola bilər. İmtahan mərkəzində sənədləriniz yoxlanıldıqdan sonra şəkliniz çəkilir. İmtahan otağına daxil olmadan öncə üzərinizdəki bütün əşyaları təhvil verirsiniz, içəriyə heç bir əşya (hətta adi kağız) keçirilməsinə icazə verilmir. Test Mərkəzi tərəfindən sizə kağız və qələm verilir. İmtahan zalına daxil olmadan öncə qələmin yazıb-yazmamasını yoxlayın.

İmtahan zalı müasir standartlara tam olaraq cavab verməsə də, imtahan üçün bütün lazımı şərait yaradılmışdı. Zalda 4 ya 5 nəfər eyni zamanda imtahan verir və nəzarətçilər tərəfindən maksimum sakitçilik təmin edilir. Nəzarətçi otaqdan çıxdıqda qapı açarla bağlanılır və kənar şəxslər nəzarətsiz otağa daxil ola bilmir. Həmçinin siz özünüz də imtahana başladıqdan sonra, imtahanı bitirənədək otağı tərk edə bilməzsiniz. İmtahanın 2-2.5 saat çəkəcəyini nəzərə alaraq hər şeyi əvvəlcədən *"götür-qoy"* etməyiniz arzuolunandır.

Kağız-qələmlə yanaşı kiçik qulaqlıq da verilir, əgər tənəffüs zamanı səs-küy olarsa istifadə etmək üçün. Amma məndə ehtiyac olmadı.

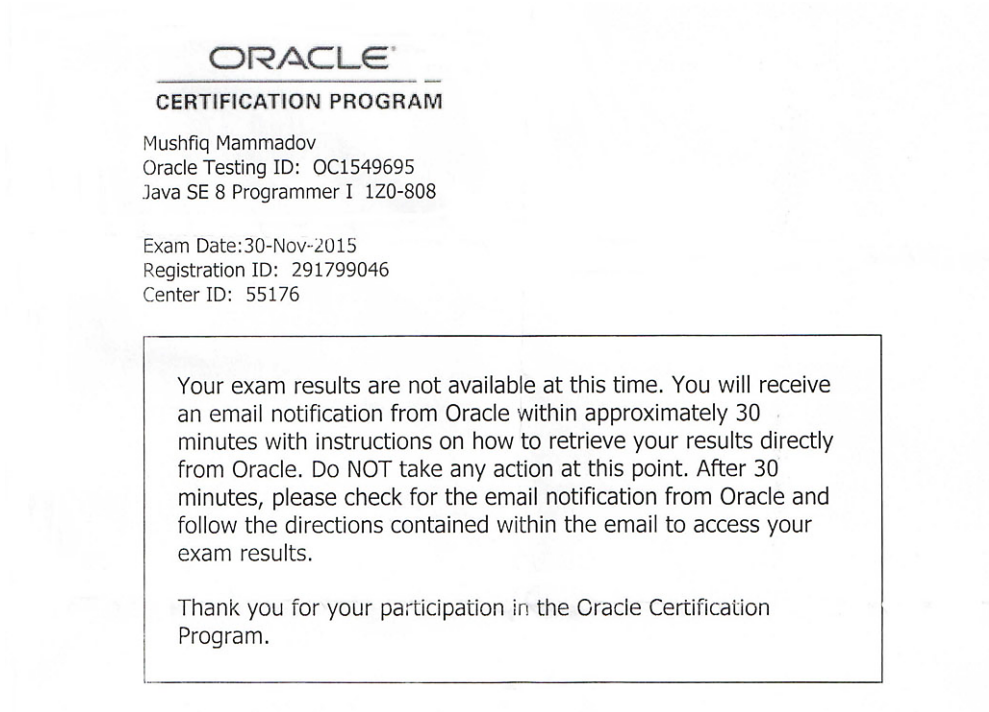
İmtahan zamanı təmkinli olmağa çalışın. Ola bilər ki, ilkin suallar sizə çətin gəlsin. Belə olduqda dərhal narahatçılıq hissi və ya həyəcan keçirməyin, təmkininizi qorumağa çalışın. Bir neçə sualdan sonra artıq hər şey qaydasına düşəcək. Özüm də imtahanda bu situasiya ilə rastlaşmışdım. Müəyyən qədər həyəcan, narahatçılıq keçirdim. Amma Enthuware mock exam`lardan müəyyən təcrübəm var idi artıq, orada da bəzi Standard Test`lər çətin suallarla başlayıb getdikcə rahatlaşırdı. Bilmədiyiniz suallar üzərində çox vaxtınızı itirməyin. Əgər hər hansı bir sualı tapmaq üçün 1.5-2 dəqiqədən artıq vaxt xərcələmiş və doğru cavabını tapa bilmədinizsə, növbəti suala keçin. Həmin sualı *"mark"* edin və sonda yenidən baxın. Bəlkə də sualdakı hansısa kiçik nüans diqqətinizdən qaçır, ola bilər ki, təkrar baxdıqda bir neçə saniyəyə də tapasınız. Sualların hamısını bitirdikdən sonra çalışın sonda bütün suallara təkrar bir də göz gəzdirmək üçün 20-25 dəqiqə vaxtınız qalsın. Yazdığınız cavablara əmin olmaq baxımından çox böyük faydası var.

Vacib bir məqamı da qeyd edim ki, imtahandan öncə mütləq https://certview.oracle.com saytında qeydiyyatdan keçməlisiniz. Bu barədə növbəti addımda ətraflı məlumat veriləcək.

Qeyd. Yuxarıda qeyd edilən fikirlər imtahan verdiyim Test Mərkəzinə (Xarici Dillər Universiteti, IBT Test Mərkəzi) istinadən yazılıb.

Addım 7. İmtahanın nəticəsini öyrənmək

İmtahanın nəticəsi dərhal elan olunmur. İmtahanı bitirdikdən sonra sizə Test Mərkəzi tərəfindən aşağıdakı yazılar qeyd olunmuş bir fayl verilir:



ORACLE

CERTIFICATION PROGRAM

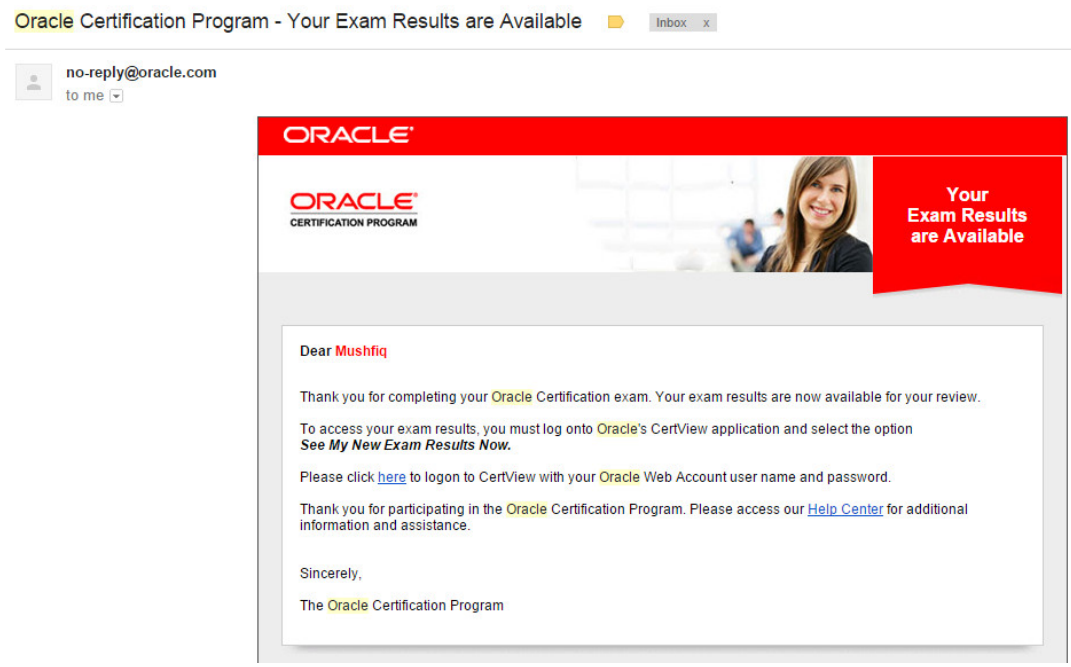
Mushfiq Mammadov
Oracle Testing ID: OC1549695
Java SE 8 Programmer I 1Z0-808

Exam Date:30-Nov-2015
Registration ID: 291799046
Center ID: 55176

Your exam results are not available at this time. You will receive an email notification from Oracle within approximately 30 minutes with instructions on how to retrieve your results directly from Oracle. Do NOT take any action at this point. After 30 minutes, please check for the email notification from Oracle and follow the directions contained within the email to access your exam results.

Thank you for your participation in the Oracle Certification Program.

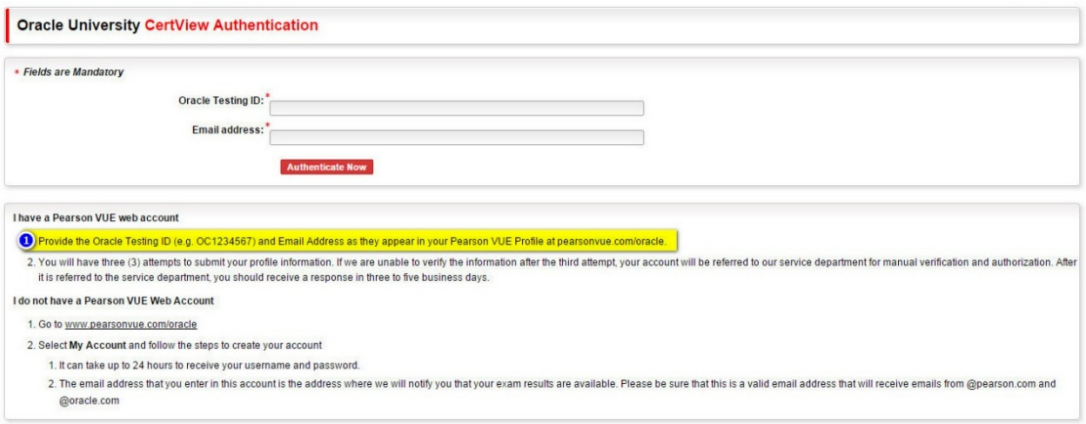
Yazıdan da aydın görünür ki, imtahan bitdikdən təxminən yarım saat ərzində sizin email ünvanınıza imtahan nəticəsinin hazır olması ilə bağlı məktub gəlir.



İmtahanın nəticəsini görə bilmək üçün mütləq https://certview.oracle.com saytında qeydiyyatdan keçmək lazımdır, çünki imtahanın nəticəsi məhz bu saytda yayımlanır. Sayta daxil olarkən ardıcıl/alt-alta 3 pəncərə ilə rastlaşacaqsınız:

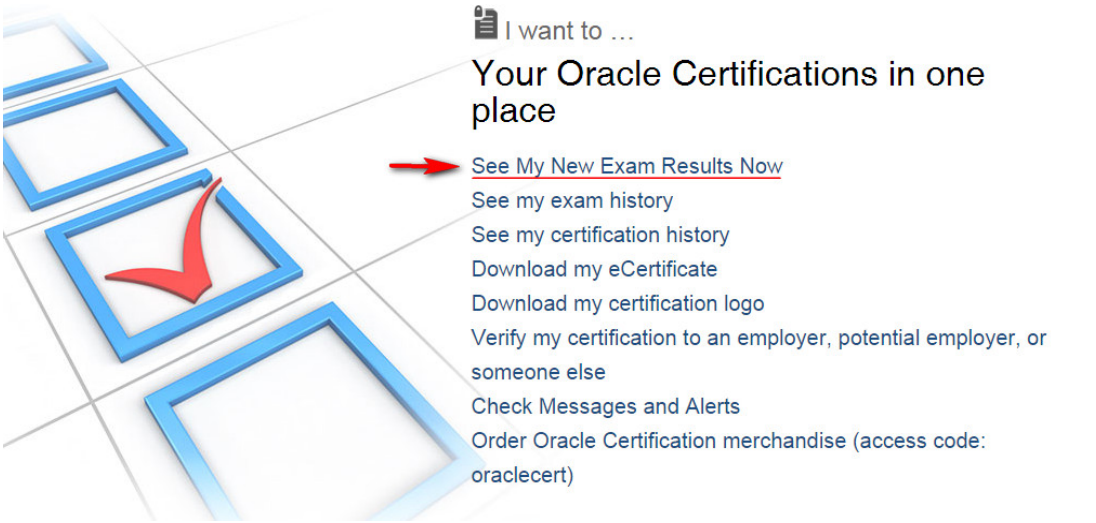
- **Login** - *Returning CertView Users;*
- **Sign up** - *First Time Users;*
- **Register** - *Create Web Account.*

Əgər artıq certview saytında qeydiyyatınız varsa, o zaman **Login** pəncərəsində *"Sign In To CertView"* linkini sıxaraq hesabınıza daxil ola bilərsiniz. Yox əgər Oracle hesabınız var, CertView hesabınız yoxdursa o zaman **Sign Up** pəncərəsində *"Authenticate My CertView Account"* linkini sıxırsız və açılan pəncərədə *Oracle Testing ID* və email ünvanınızı daxil edirsiniz (1-ci punkta diqqət):

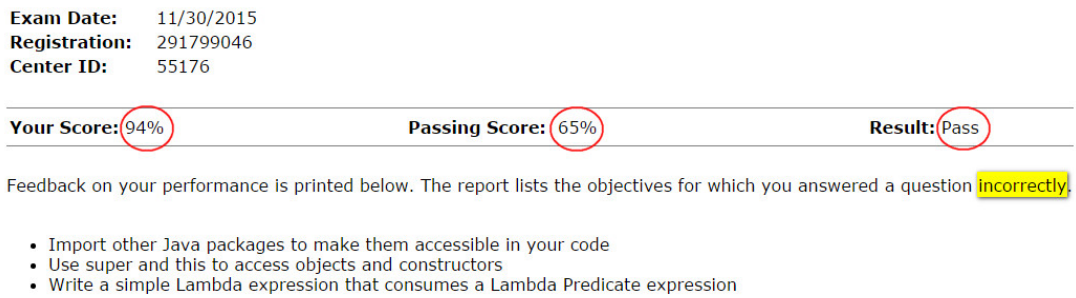


Ümumiyyətlə, əgər Oracle hesabınız yoxdursa, o zaman **Register** pəncərəsindən *"Create My Oracle Web Account Now"* linkini sıxaraq əvvəlcə Oracle hesabınızı yaradırsınız. Sonra isə həmin hesabla **Sign Up**, daha sonra isə **Login** olursunuz.

Qeydiyyatı uğurla başa çatdırıb CertView hesabınıza daxil olduqda belə bir menyu ilə rastlaşacaqsınız:



İmtahan nəticənizin hazır olması barədə emailinizə bildiriş gəldikdə CertView hesabınıza daxil olub, nəticəni görmək üçün *"See My New Exam Results Now"* linkini sıxırsınız. Aşağıdakı pəncərəyə bənzər pəncərə açılır:



Bu pəncərədə topladığınız bal, imtahan üçün keçid balı və səhv cavablandırdığınız sualların hansı mövzulara aid olması ilə bağlı "feedback" göstərilir.

Problem yaşanmaması üçün CertView saytında imtahandan 1-2 həftə öncə qeydiyyatdan keçməyiniz tövsiyə olunur:

http://www.coderanch.com/t/662653/ocajp/certification/received-OCA-results-Pearson#3079561

Addım 8. Sertifikatın elektron və çap versiyalarını əldə etmək

Elektron sertifikat imtahandan təqribən 48 saat sonra (daha tez də ola bilər) hazır olur. Hazır olduqda emailə bununla bağlı məlumat gəlir:

Dear Mushfiq Mammadov,

Congratulations on earning your Oracle Certification credential!

To provide you with immediate access to your certification credential, you can now download the eCertificate from your account at certView.oracle.com. Certification History from the Certification Status tab.

Upon request, a hard copy of your certificate will be sent via standard shipping to your personal or business address. Oracle does not ship certificates to 3rd party training or testing centers.

To request a hard copy certificate, you are required to log onto your **Pearson VUE Profile** and confirm your mailing address. Once your address is confirmed/updated in your Pearson VUE Profile, please request a hard copy of your certificate using the online request form. You should expect delivery of your certificate approximately 4 weeks from the request date.

We encourage you to continue your Oracle education and to keep your Oracle Certification up-to-date with the latest Oracle Product release. You can find more information on Oracle University training and certification offerings on our website at http://www.oracle.com/education/certification.

If you require further assistance, do not respond to this email correspondence. Please contact Oracle Certification Support.

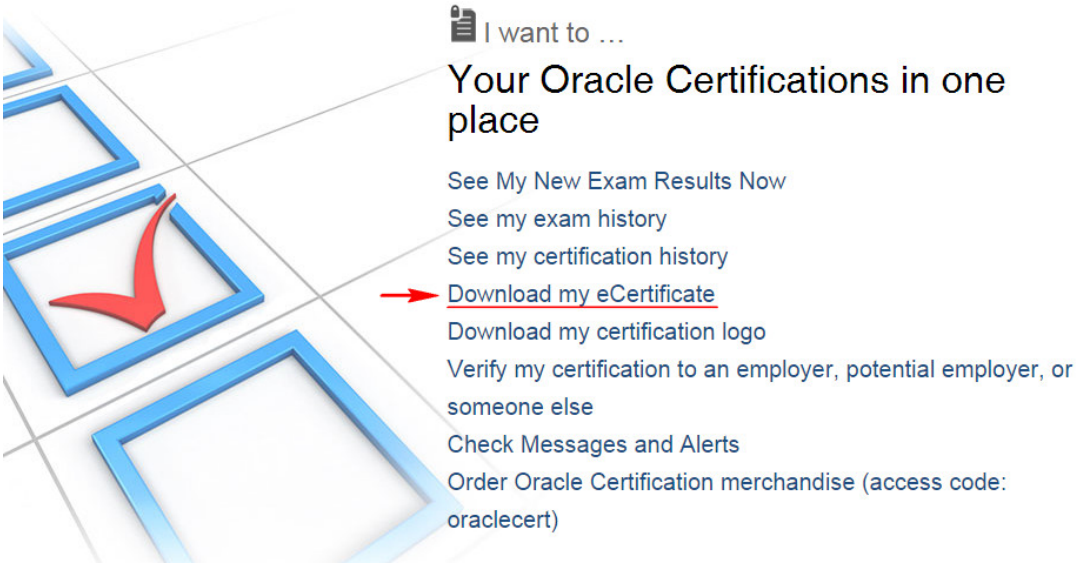
You can now order Oracle Certification branded merchandise to publicize your accomplishment! Use the access code "oraclecert".

Best Regards,

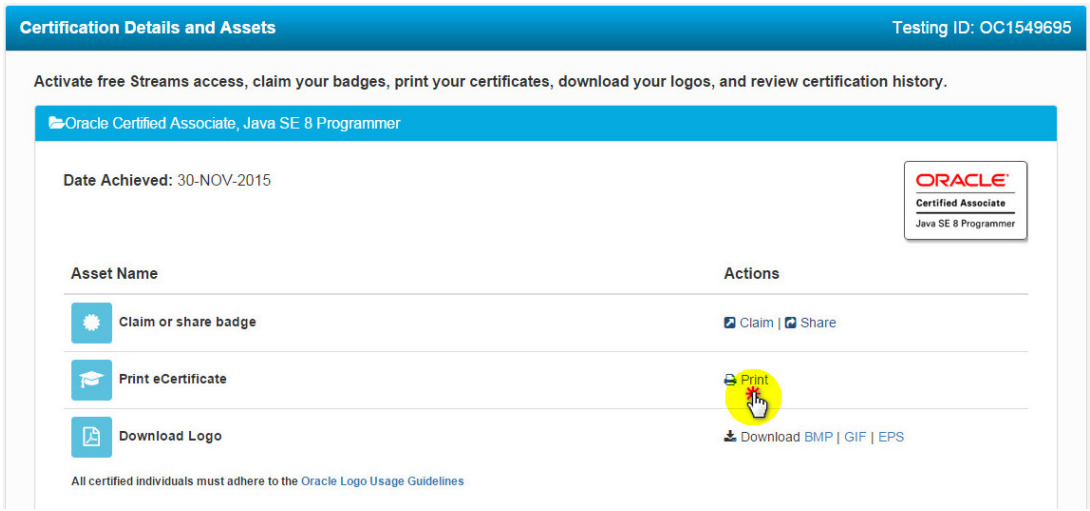
The Oracle Certification Team

Elektron sertifikatı çap etmək və ya pdf formatında yükləmək üçün https://certView.oracle.com saytına daxil olaraq ilk öncə *"Dowload my eCertificate"* və ardınca da *"Print"* linkini sıxırsınız:

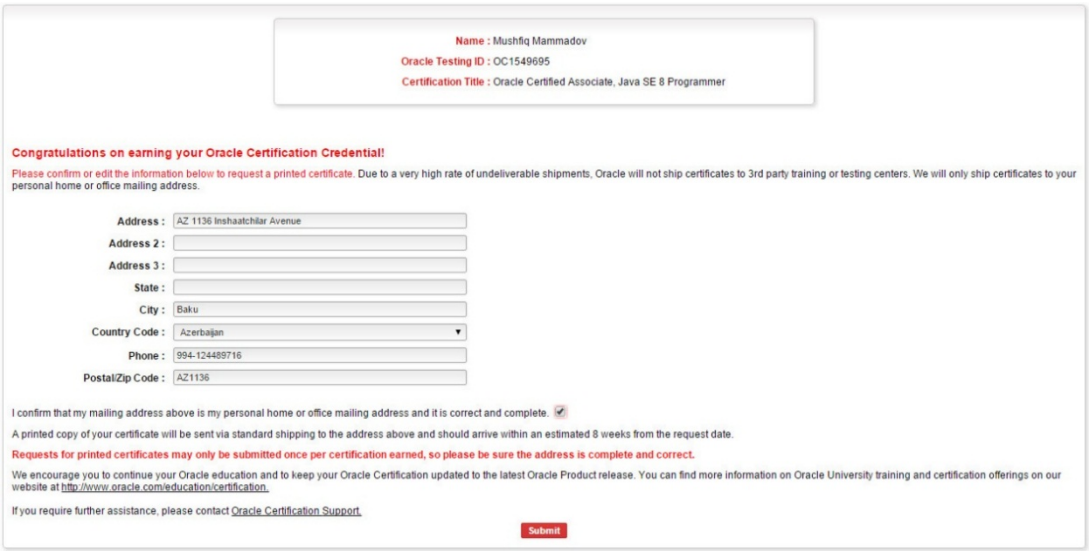
Addım 1:



Addım 2:



Əgər sertifikatın print olunmuş (kağız) formasını istəyirsinizsə, bunun üçün mütləq müraciət etməlisiniz. Müraciət edilmədiyi təqdirdə sertifikat göndərilmir. Müraciət prosesi çox sadədir, sadəcə müraciət formunu doldurub göndərməlisiniz. Yuxarıda qeyd etdik ki, elektron sertifikat hazır olduqda emailə bu barədə məktub gəlir. Həmin məktuba diqqətlə baxsanız görəcəksiniz ki, orada online müraciət formunun linki qeyd edilir (*"online request form"*). Həmin linkə kliklədikdə müraciət formu pəncərəsi açılır:



Name : Mushfiq Mammadov
Oracle Testing ID : OC1549695
Certification Title : Oracle Certified Associate, Java SE 8 Programmer

Congratulations on earning your Oracle Certification Credential!

Please confirm or edit the information below to request a printed certificate. Due to a very high rate of undeliverable shipments, Oracle will not ship certificates to 3rd party training or testing centers. We will only ship certificates to your personal home or office mailing address.

Address : AZ 1136 Inshaatchilar Avenue
Address 2 :
Address 3 :
State :
City : Baku
Country Code : Azerbaijan
Phone : 994-124489716
Postal/Zip Code : AZ1136

I confirm that my mailing address above is my personal home or office mailing address and it is correct and complete.

A printed copy of your certificate will be sent via standard shipping to the address above and should arrive within an estimated 8 weeks from the request date.

Requests for printed certificates may only be submitted once per certification earned, so please be sure the address is complete and correct.

We encourage you to continue your Oracle education and to keep your Oracle Certification updated to the latest Oracle Product release. You can find more information on Oracle University training and certification offerings on our website at http://www.oracle.com/education/certification.

If you require further assistance, please contact Oracle Certification Support.

Submit

Təlimatı izləyin və məlumatları düzgün daxil etdiyinizə əmin olduqdan sonra *"submit"* düyməsini sıxın.

Müraciət qəbul edildikdən təxminən 4-8 həftə ərzində sertifikatınız qeyd etdiyiniz ünvana göndəriləcək. Mənim sertifikatım 4 həftənin tamamında gəlib çatmışdı və dizayn baxımından elektron sertifikatdan müəyyən qədər fərqlənirdi:

Qeyd. Oracle şirkətinin yeni siyasətinə görə artıq sertifikatın çap olunmuş versiyası göndərilmir. Emailə belə bir bildiriş gəlir:

Important Note: Oracle is committed to developing practices and products that help protect the environment. Hard copy printed certificates are not available.

Siz CertView`dan elektron sertifikatı pdf kimi yükləyib özünüz çap edə bilərsiniz.

Bölmə 2. İmtahan Mövzuları ilə Bağlı Xülasələr

- **Chapter 1. Java Building Blocks**
- **Chapter 2. Operators and Statements**
- **Chapter 3. Core Java APIs**
- **Chapter 4. Methods and Encapsulation**
- **Chapter 5. Class Design**
- **Chapter 6. Exceptions**

Chapter 1. Java Building Blocks

Comments

Kommentin 3 tipi var:

1. Single-line comment

```
// comment until end of line
```

2. Multiple-line comment

```
/* Multiple
*   line comment
*/
```

3. Javadoc comment

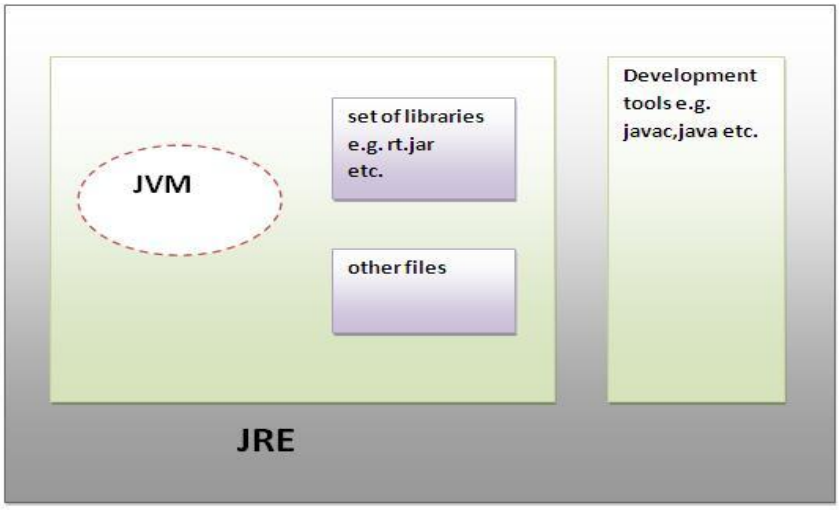
```
/**
*   Javadoc multiple-line comment
*   author MM
*/
```

Multiple-line kommentdə hər açılan 1-ci komment `/*` bağlanan 1-ci kommentə `*/` uyğun gəlir.

```
/*
 * //one
 */
// two
// // three
// /* four */
/* five */
/*
 * /* six */
 */
```

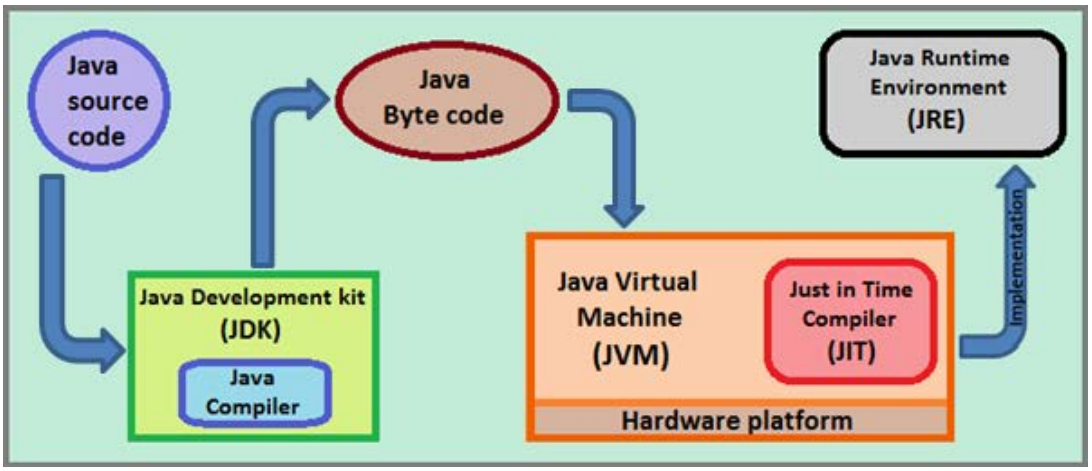
Burada 3 single-line (`two, three, four`) və 3 multiline (`one, five, six`) komment var. Sonuncu sətir kompayl (compile) xətası verir, çünki artıqdır, *"six"* sözündən sonra komment artıq bağlanır.

Kodu kompayl etmək üçün **JDK**`ya ehtiyac var. Ancaq kodu icra (run) etmək üçün **JDK**`ya ehtiyac yoxdur, **JRE** kifayət edir.



JDK

Java class faylları **JVM**`də icra edilir:



Main method

Main metod `public, static, void` olmalı və `String` tipində birölçülü *massiv* və ya *varargs* parametr qəbul etməlidir. İmtahanda rastlaşa biləcəyiniz doğru `main` method sintaksisləri aşağıdakılar ola bilər:

```
public static void main(String[] args)
static public void main(String[] args)
```

```
public static void main(String arv[])
public static void main(String... arr)
final public static void main(String args[])
```

`public` əvəzinə `private` və ya `protected` yazsaq java`nın bəzi köhnə versiyalarında main metod kimi çalışacaq. Amma yeni versiyalarda `private` və ya `protected` yazdıqda kompayl xətası verməməsinə baxmayaraq, icra etdikdə *"main metod yoxdur"* xətasını verəcək. Yadda saxlayın ki, OCA 7/8 imtahanının tələblərinə görə main metod mütləq `public` olmalıdır.

Redundant Imports

Əgər iki class eyni paket daxilində yerləşərsə, bir classın daxilində digər classa müraciət etmək üçün həmin classı import etməyə ehtiyac yoxdur. Amma həmin classı import etsək, bu xəta verməyəcək, normal kompayl olunacaq. Bu hal "redundant import" sayılır, yəni import etməsək belə kod kompayl olunacaq, etsək də kod xəta verməyəcək. Başqa sözlə, bu halda import etmək artıq, lazımsız işdir.

Hətta classın özünü belə mövcud olduğu fayl daxilində import etsək kompilyator (compiler) xəta verməyəcək, kod normal işləyəcək:

```
1: package mm;
2: public class MasterITM {}
```

```
1: package mm;
2: import mm.MasterITM;
3: import mm.OrientITM;
4: public class OrientITM {
5:    private MasterITM masterItm;
6: }
```

Naming Conflicts

Java`da paketlərin istifadə edilmə səbəblərindən biri də class adlarının unikal olmamasıdır. Əgər eyniadlı classlar varsa, biz onları yerləşdikləri paketə görə fərqləndiririk. Tutaq ki, aşağıdakı kimi bir kod nümunəmiz var:

```
public class Conflicts {
   Date date;       // line D
   //some more code
}
```

Bu kod nümunəsində biz `Date` classını import etməliyik, amma `Date` classı iki fərqli paketdə mövcuddur: `java.util.Date` və `java.sql.Date`. Fərqli import nümunələrinə baxaq:

```
// import 1
import java.util.*;
import java.sql.*;

// import 2
import java.util.Date; // Date bu paketdən seçilir
import java.sql.*;     // Date`dən başqa bütün classlar import olunur

// import 3
import java.util.Date;
import java.sql.Date;  // doesn't compile
```

import 1 – yuxarıdakı kod nümunəsində *D* sətri kompayl olunmur, çünki bu sətirdə konflikt yaranır, java `Date` classının hansı paketdən import olunduğunu təyin edə bilmir;

import 2 – kod normal kompayl olunur;

import 3 – `import java.sql.Date;` sətri kompayl olunmur, *"..the same simple name is already defined.."* xətası verir. Əgər hər iki `Date` classını eyni vaxtda istifadə etmək istəyiriksə, o zaman aşağıdakı formada yazırıq:

```
public class Conflicts {
   java.util.Date date;
   java.sql.Date sqlDate;
}
```

Package

temp qovluğunda iki package və hər birində də bir class yaradırıq:

C:\temp\packagea\ClassA.java
C:\temp\packageb\ClassB.java

```
package packagea;
public class ClassA {}
```

```
package packageb;
import packagea.*;
class ClassB {
   public static void main(String[] args) {
      ClassA a;
      System.out.println("Class A uğurla import olundu");
   }
}
```

Cmd ilə bu kodu dərləyib icra etmək üçün aşağıdakı əmrlər ardıcıl icra olunmalıdır:

```
1. cd C:\temp
2. javac packagea\ClassA.java packageb\ClassB.java
3. java packageb.ClassB
```

Əgər mənbə kodda (source code) paket (package) adı qeyd olunmayıbsa, həmin classın default paketdə yaradıldığı fərz edilir. Default paketdə yaradılmış classların digər paketdəki classlar tərəfindən import edilməsi mümkün deyil! Ona görə də default paketlərdən istifadə etmək tövsiyə edilmir.

`lava.lang` paketi java`da xüsusi paket hesab olunur və avtomatik import olunur.

Code formatting on the Exam

Əgər imtahanda verilən sualdakı kodların ilk sətri 1-dən başlayırsa (the line number 1), o zaman zəruri olan bütün classların import edilib edilmədiyinə diqqət yetirmək lazımdır. Məsələn, aşağıdakı kodda problem yoxdur, çünki kod 6-cı sətirdən başlayır və belə olan halda importun edilib edilməməsi ilə bağlı dəqiq təxmin yürütmək mümkün deyil:

```
6: public void method(ArrayList list) {
7:   if (list.isEmpty()) { System.out.println("e");
8:   } else { System.out.println("n");
9: }  }
```

Bu kodda isə `java.util.ArrayList` import edilmədiyindən kompayl xətası verir:

```
1: public class LineNumbers {
2: public void method(ArrayList list) {
3:   if (list.isEmpty()) { System.out.println("e");
4:   } else { System.out.println("n");
5: }  } }
```

Constructors

Konstruktorun içində istənilən kod yazmaq mümkündür, lakin onun əsas məqsədi dəyişənlərə dəyər mənimsətməkdir (initialize fields). Daha geniş şəkildə “Chapter 4”-də baxacağıq.

Instance Initializer Blocks

Bəzən kod blokları metodun daxilində olur və metod çağırılanda həmin kod blokları metoddakı kod ardıcıllığına uyğun icra edilir. Bəzən isə bu kod blokları metoddan kənarda olur. Həmin kod blokları ***instance initializers blocks*** adlanır.

```
public class TestCodeBlocks {
   public static void main(String[] args) {
     new TestCodeBlocks(); //bu kommentə salinsa, ancaq step 2 --> step 3 --> step 4
     System.out.println("step 2");
     { System.out.println("step 3"); }
     System.out.println("step 4");
   }
   { System.out.println("step 1"); }
}
```

Output:

```
step 1
step 2
step 3
step 4
```

Order of Initialization

"Order of initialization" ilə bağlı aşağıdakı qaydalar mövcuddur:

- Dəyişənlər (fields) və instance initializer bloklar faylda (və ya classda) mövcud olduğu ardıcıllığa uyğun olaraq icra edilir;
- Konstruktor - bütün dəyişənlər (all fields) və instance initializer bloklar icra edildikdən sonra (have run) icra olunur.

```
public class MySweety {
   private String name = "Leila";   // step-1
   { System.out.println(name + " became a mother."); }   // step-2
   public MySweety() {
      System.out.println("My sweety was born.");   // step-3
      name = "Aliya";   // step-4
   }

   public static void main(String[] args) {
      MySweety mySweety = new MySweety();
      System.out.println("Her name is " + mySweety.name);   // step-5
   }
}
```

Output:

```
Leila became a mother.
My sweety was born.
Her name is Aliya
```

Primitive types

Java`da 8 primitiv tip var:

Cədvəl 1.1 Javada primitiv tiplər

Açar söz	Ölçü	Dəyər aralığı		Varsayılan dəyər	Nümunə
		min	max		
byte	8 bit	-128	127	0	123
short	16 bit	-32,768	32,767	0	123
int	32 bit	-2^{31}	$2^{31}-1$	0	123
long	64 bit	-2^{63}	$2^{63}-1$	0L	123
float	32 bit float			0.0f	123,45f
double	64 bit float			0.0d	123,456
boolean	N/A			false	true
char	16 bit unicode	'\u0000'	'\uffff'	'\u0000'	'm'

Bu tiplərin qiymət aralıqları aşağıdakı qaydada hesablanır (nümunə üçün `byte` götürülür):

`byte = 8 bit = 2^8 = 256`

256`nın yarısı pozitiv və yarısı da neqativ dəyər üçün ayrılır:

`256/2 → -128 < byte < 127` *(0 pozitiv ədədlər siyahısına daxil edildiyindən 127 olur)*

Aldığı dəyərin böyüklüyünə görə ardıcılıq belə gedir (kiçikdən böyüyə):

`byte → short → int → long → float → double`

Kod nümunəsi:

```
long i1 = 1_000_000_000;    // does compile

/* bunu int hesab edir və int dəyişəninin max dəyəri keçdiyini görüb xəta verir, mütləq sonuna L artırmaq lazımdır.*/
long i2 = 1_000_000_0000;   // doesn't compile
int i3 = 129L;              // doesn't compile

float f1 = 10;
float f2 = 1_000_000_000_000_000_000L;
float f3 = 10.0;  // xəta verir, çünki bunu double kimi görür, mütləq sonuna f əlavə etmək lazımdır.
int i3 = 9f;     // doesn't compile
int a = new Integer(5);
```

Char tipinə `0-65535` aralığında istənilən rəqəm tipli dəyəri mənimsətmək olar. Ancaq mənfi dəyər mənimsətmək mümkün deyil, kompayl xətası verir.

```
char c1 = 0;
char c2 = 65535;
char c3 = 65536;  // doesn't compile
char c4 = -0;
char c5 = -10;    // doesn't compile
System.out.println((int)'a');  // output: 97
```

Java`da onluq ədədlərdən başqa 8-lik, 16-lıq və 2-lik ədədlər də var:

- ***octal (digits 0-7)*** – 8-lik ədədlər 0-7 arasındakı (0 və 7 daxil olmaqla) rəqəmləri götürür və prefix olaraq 0 qəbul edir, yəni 0 ilə başlayır: `017`, `07`, `023567`
- ***hexadecimal (digits 0-9 and letters A-F)*** – 16-lıq ədədlər 0-9 arasındakı rəqəmlər və A, B, C, D, E, F rəqəmlərini qəbul edir. Prefix kimi **0x** və yaxud **0X** istifadə edir: `0xFF`, `0Xff`, `0x198D`
- **binary (digits 0-1)** – 2-lik ədədlər ancaq 0 və 1 rəqəmlərindən ibarət ola bilər və prefix **0b** və ya **0B**: `0b10`, `0B0_0011_10_0`;

```
byte b1 = 07;
byte b2 = (byte) 023567;  // byte`ın max qiymətindən > olduğu üçün cast olunmalıdır
byte b3 = 08;            // doesn't compile, max 7 olmalıdır

class Hexadecimal {
   public static void main(String[] args) {
      int x = 0X0001;
      int y = 0x7fffffff;
      int z = 0xDeadCafe;
      System.out.printf("x=%d; y=%d; z=%d %n", x, y, z);
   }
}

class Octal {
   public static void main(String[] args) {
      int six = 06;      // Equal to decimal 6
      int seven = 07;    // Equal to decimal 7
      int eight = 010;   // Equal to decimal 8
      int nine = 011;    // Equal to decimal 9
      System.out.println("Octal 010 = " + eight);
   }
}

public class Q6 {
   public static void main(String[] args) {
      int a[] = {1, 2, 053, 4};
      int b[][] = { {1, 2, 4}, {2, 2, 1}, {0, 43, 2} };
      System.out.println(a[2]);  // 053 -> 0*8^2 + 5*8^1 + 3*8^0 = 43
```

```
        System.out.println(a[3] == b[0][2]);       // true
        System.out.println((a[2] == b[2][1]));     // true
    }
}
```

Cədvəl 1.2 0-dan 15-dək onluq ədədlərin digər say qarşılığı

10`luq	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4`lük	0	1	2	3	10	11	12	13	20	21	22	23	30	31	32	33
8`lik	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
16`lıq	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2`lik	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Mənbə: https://en.wikipedia.org/wiki/Quaternary_numeral_system

Rəqəmlərin yaxşı oxunması üçün java 7 ilə yeni xüsusiyyət gəlib – *altxətt* (underscore):

```
int million1 = 1000000;
int million2 = 1_000_000;
```

Altxətti rəqəmlərin əvvəlinə və sonuna, həmçinin onluq nöqtənin (decimal point) sağına və soluna əlavə etmək olmaz. Bir sözlə, altxətt ancaq 2 rəqəm arasında işlədilə bilər:

```
double notAtStart = _1000.00;            // does NOT compile
double notAtEnd = 1000.00_;              // does NOT compile
double notByDecimal = 1000_.00;          // does NOT compile
double annoyingButLegal = 1_00_0.0_0;    // does compile

float f1 = 1_00_0f;       // does compile
float f2 = 1_00_0_f;      // does NOT compile
long L1 = 1_000__000L;    // does compile
long L2 = 1_000_000_L;    // does NOT compile

float d = 0 * 1.5;    // doesn't compile, but float d = 0 * 1.5f; and float d = 0 * (float)1.5 ; are OK
float f1 = 43e1;      // DOES NOT COMPILE
float f2 = 43e1f;     // 430.0
double d1 = 43e;      // DOES NOT COMPILE
double d2 = 43e1;     // 430.0
double d3 = 43e4;     // 430000.0
double d4 = 43e6;     // 4.3E7
double d = .05;       // 0.05

float f5 = 43d;       // DOES NOT COMPILE
double d5 = 43d;      // 43.0
```

Suallarda diqqətinizdən yayına biləcək bir nüansı da nəzərinizə çatdırmaq istəyirəm. Yuxarıda qeyd etdiyimiz səbəblərə görə aşağıdakı kod kompayl olunmur:

```
long z = _123_456L;
```

Qeyd etdik ki, altxəttin əvvəldə gəlməsi doğru deyil. Amma dəyişən adlarında (identifier) altxətt əvvəldə gələ bilər, yəni `_123_456L` doğru elan edilmiş dəyişən adıdır. Bu səbəbdən aşağıdakı kod səhv deyil:

```
int _123_456L = 10;
long z = _123_456L;
```

Altxətt bütün tip – `long`, `double`, `float`, eləcə də 2-lik, 16-lıq say sistemində olan ədədlərlə də işlədilə bilər:

```
int hex = 0xCAFE_BABE;
float f = 9898_7878.333_333f;
int bin = 0b1111_0000_1100_1100;
```

Qeyd etdiyimiz primitiv dəyişən tipləri arasındakı əlaqələri daha yaxşı qavramaq və bəzi çaşdırıcı məqamları aydınlaşdırmaq üçün bəzi əlavə nümunələrə də nəzər salaq. Deməli qeyd etdik ki, `char` tipli dəyişənə `0` və `65535` aralığında olan istənilən dəyəri birbaşa mənimsətmək mümkündür. Amma bu aralığa daxil olan dəyəri `int` dəyişənə mənimsədib və sonra həmin dəyişəni `char` dəyişənə mənimsətdikdə kompayl xətası baş verir. Xətanın aradan qalxması üçün ya həmin dəyişən `char`a cast olunmalıdır yaxud da `final` elan edilməlidir:

```
char a = 0x892;      // hexadecimal literal
char b = 982;        // int literal
int i = b;
final int m = b;
final int n = 982;
char c1 = i;         // DOES NOT COMPILE
char c2 = m;         // DOES NOT COMPILE
char c3 = n;
```

Amma `0` və `65535` aralığına daxil olmayan `int` dəyərləri birbaşa `char`a mənimsətdikdə yenə xəta verir və cast mütləq tələb olunur:

```
// char e = -29;      // Possible loss of precision; needs a cast
// char f = 70000;    // Possible loss of precision; needs a cast

char c = (char) 70000;   // The cast is required; 70000 is out of char range
char d = (char) -98;     // Ridiculous, but legal
```

`char` dəyişəni `short` dəyişənə, `short` dəyişəni isə `char` dəyişənə cast etmədən mənimsətmək mümkün deyil. Amma `char` dəyişəni cast etmədən `int` dəyişənə mənimsətmək mümkündür, əksi isə doğru deyil:

```
char c = 'a';
long l = c;
```

```
int i = c;
c = i;          // DOES NOT COMPILE
short s = c;    // DOES NOT COMPILE
c = s;          // DOES NOT COMPILE

char c1 = Short.MAX_VALUE;

short s2 = Short.MAX_VALUE;
char c2 = s2;   // DOES NOT COMPILE

short s3 = Short.MAX_VALUE;
char c3 = (char) s3;

final short s4 = Short.MAX_VALUE;
char c4 = s4;

byte b5 = Byte.MAX_VALUE;
char c5 = b5;    // DOES NOT COMPILE
```

Eyni qaydalar digər primitiv dəyişənlər üçün də keçərlidir:

```
short s1 = 1;
byte b1 = s1;       // DOES NOT COMPILE

short s2 = 1;
byte b2 = (byte) s2;

final short s3 = 1;
byte b3 = s3;

final short s4 = 200;
byte b4 = s4;      // DOES NOT COMPILE, final olsa da aralıqdan kənar dəyərdi
```

Declaring Multiple Variables

```
int i1, i2, i3 = 0;       // declare 3, initialize only 1 (i3)
boolean b1, b2;
String s1 = "1", s2;
int j1; int j2;

int num, String value;    // doesn't compile
double d1, double d2;     // doesn't compile
int j3; j4;               // doesn't compile
```

Identifiers

Düzgün identifikator elan etmək üçün əsas 3 qaydaya riayət edilməlidir:

- Adlar hərflə və yaxud da $ və ya _ (underscore) simvollarından biri ilə başlamalıdır;
- Sonrakı simvolların içərisində həmçinin rəqəmlər də ola bilər;
- Java açar sözlərindən (reserved word) istifadə edilə bilməz.

Java Unicode simvolları dəstəkləyir, ona görə də öz əlifbamıza uyğun hərflərdən istifadə etmək səhv sayılmır.

Düzgün elan edilmiş dəyişən adları:

```
_
t_e_s_t
Odəniş$
c2c
__SStillOkbutKnotsonice$
$OK2Identifier
Public
```

*Düzgün **olmayan** dəyişən adları:*

```
Em@il
3DPointClass
*$cofee
public
```

Java`da açar sözlər (reserved word) kiçik hərflərlə başlayır, ona görə də `String` açar söz hesab edilmir:

```
String String = "String";    // is a perfectly valid syntax!
```

Açar sözlərin siyahısı aşağıda verilmişdir:

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private

`protected`	`public`	`return`	`short`	`static`
`strictfp`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`true`	`try`
`void`	`volatile`	`while`		

`const` və `goto` hal-hazırda Java`da istifadə olunmur, amma ola bilsin ki, gələcəkdə Java bunları istifadə etməyi qərara alsın. O səbəbdən də açar söz kimi qəbul edib ki, istifadəçilər tərəfindən dəyişən adı kimi istifadə olunmasının qarşısını alsın.

`static` və `final` açar sözləri həm dəyişənlər, həm də metodlar üçün istifadə edilə bilər. `transient` və `volatile` açar sözləri (modifiers) ancaq dəyişənlər, `abstract` və `native` açar sözləri isə ancaq metodlarla istifadə edilə bilər. `abstract` açar sözü classlarla işlədilə bilər, amma `native` işlədilə bilməz.

Həmçinin siz class daxilində eyni adlı dəyişən və metod istifadə edə bilərsiniz:

```
class Test {
   String test;
   public void test() {}
}
```

Local Variables

Lokal dəyişənlər metod daxilində yaradılmış dəyişənlərdir. Lokal dəyişənlərin varsayılan (default) dəyəri olmur, ona görə də onları istifadə etməzdən əvvəl mütləq dəyər mənimsədilməlidir.

Aşağıdakı kod nümunəsi üzərindən davam edək:

```
public void testLocalInitialization() {
   int count = 5;
   int number;

   if (count > 5)       number = 1;
   else if (count <= 5) number = 2;

   System.out.println(number);    // doesn`t compile
}
```

`number` dəyişəninə dəyər mənimsədilməmiş istifadə edildiyinə görə kompayl xətası verir. Biz fikirləşə bilərik ki, axı `if` şərtinin içərisində dəyər mənimsədilib, `count` ya 5-dən böyükdür, ya bərabərdir, ya da kiçikdir. Axı istənilən halda bu iki şərtdən biri ödənəcək. Bəs onda niyə kompilyator `number`i dəyəri mənimsədilməmiş kimi görür?! Çünki kompayl zamanı (compile

time) kompilyator bunu təyin edə bilmir, ancaq icra vaxtı (runtime) bilir. Amma əgər biz `count` dəyişənini `final` elan etsək o zaman kod kompayl olunacaq. Və yaxud da sona `else` ifadəsi əlavə etsək, kompayl olunacaq. Çünki *if-else* strukturunda dəyər mənimsədildikdə kompilyator başa düşür ki, heç bir şərtdən asılı olmayaraq istənilən halda `if` şərti ödənməsə, `else` blokuna müraciət olunacaq və dəyər mənimsədiləcək.

```
public void testLocalInitialization() {
   int count = 5;
   int number;

   if (count > 5)       number = 1;
   else if (count <= 5) number = 2;
   else                 number = 3;

   System.out.println(number);    // does compile
}
```

Dəyişən `final` elan edildikdə isə kompilyator başa düşür ki, həmin dəyişənin dəyəri heç vaxt dəyişə bilməz və ona görə də kompilyator həmin dəyişənin dəyərini birbaşa şərtə qoyub yoxlayır.

```
public void testLocalInitialization() {
   final int count = 5;
   int number;

   if (count > 8)       number = 1;
   else if (count <= 4) number = 2;

   System.out.println(number);    // doesn`t compile
}
```

Bu şəkildə də başa düşə bilərik:

```
if (5 > 8)       number = 1;
else if (5 <= 4) number = 2;
```

Əgər şərtlərdən biri geriyə `true` dəyər qaytararsa, kod kompayl olacaq:

```
if (5 > 8)       number = 1;
else if (5 <= 5) number = 2;
```

Instance and Class Variables

Java`da lokal dəyişənlərdən əlavə *instance* və *class* dəyişənləri də mövcuddur. Instance dəyişənlər həm də "fields" adlandırılır. Instance və class dəyişənlər lokal dəyişənlərdən fərqli olaraq classın daxilində, amma metodun xaricində elan edilir. Lokal dəyişənlərdən digər fərqləri isə odur ki, bu növ dəyişənlər istifadə edilmədən öncə dəyər mənimsədilməsi

(initialize) tələb etmir. Instance və class dəyişənlər artıq elan olunan vaxtı onlara susmaya görə (default) dəyər mənimsədilir. Susmaya görə dəyərlərlə aşağıdakı cədvəldən tanış ola bilərsiniz:

Cədvəl 1.3 Dəyişən tiplərinə görə default dəyərlər

Dəyişənin tipi	Mənimsədilən default dəyər
`boolean`	`false`
`byte, short, int, long`	`0`
`float, double`	`0.0`
`char`	`'\u0000'`
Bütün obyekt tipləri üçün	`null`

Class dəyişənləri ilə instance dəyişənlərinin fərqi isə odur ki, class dəyişənlərin önündə `static` açar sözü yazılır, buna görə də bu dəyişənlər həm də `static` dəyişənlər adlandırılır. Static dəyişənlər classın instansını yaratmadan birbaşa class adının özü ilə müraciət etməklə digər classlarda çağırıla bilər. Daha ətraflı növbəti mövzularda baxacağıq.

Variable Scope

Dəyişənlərin təsir dairəsi (variable scope) bu formadadır:

- Lokal dəyişənlər – elan olunduğu sətirdən daxilində olduğu kod bloku bitənədək;
- Instance dəyişənlər – elan olunduğu vaxtdan GC tərəfindən silinənədək;
- Class dəyişənlər – elan olunduğu vaxtdan proqram sonlananadək.

Metoda göndərilən parametr həmin metodun lokal dəyişəni hesab edilir:

```
1.  public void testMethod(int a) {
2.      int b = 5;
3.  }
```

Bu metodun 2 lokal dəyişəni var – `a` və `b`. 3-cü sətirdə həm `a`, həm də `b` dəyişəninin təsir dairəsi bitir.

Ordering Elements in a Class

Cədvəl 1.4 Class`ın elementləri

Element	Nümunə	Tələb edilir?	Harada yerləşməlidir?
Paket elanı	`package abc;`	Xeyr	Faylda 1-ci (ilkin) sətirdə
Import ifadələri	`import java.util.*;`	Xeyr	Paketdən sonra
Class elanı	`public class C`	Bəli	Import`dan sonra
Dəyişən (field) elanı	`int value;`	Xeyr	Cass daxilində istənilən yerdə
Metod elanı	`void method()`	Xeyr	Cass daxilində istənilən yerdə

Komment kodun içərisində istənilən yerdə işlədilə bilər, məhdudiyyət qoyulmur, hətta paketdən əvvəl də gələ bilər. Metod və dəyişənlər isə məlumdur ki, ancaq classın daxilində istənilən yerdə işlədilə bilər. Burada əsas yadda saxlamalı məqam *package*, *import* ve *class*`ın faylda yerləşmə ardıcıllığıdır. package və import optional`dır, yəni istəyə bağlı olaraq istifadə edilə və ya edilməyə bilər. Ancaq class mütləq olmalıdır. Əgər package ve import istifadə edilərsə, onda ardıcıllıq mütləq belə olmalıdır:

package -> import -> class

Study Guide`da bunu yadda saxlamaq üçün yaxşı metod tövsiyə edilir: *Picture* sözünün ilk hecasını yada salırsız - **PIC**

***P**(ackage)**I**(mport)**C**(lass)*

Bir faylın (.java) içərisində bir neçə class yaradıla bilər, amma onlardan ancaq biri `public` ola bilər. public olan classın adı faylın adı ilə mütləq eyni olmalıdır.

```
1. public class Names {}
2. class Surnames {}
```

Bu zaman faylın adı mütləq `Names.java` olmalıdır. Amma bütün classlar *default access* ola bilər, yəni hər hansı bir classın `public` olması mütləq deyil.

Garbage Collection

Bildiyimiz kimi Java`da yaradılan bütün obyektlər heap yaddaşda saxlanılır və müəyyən yer tutur. Heap yaddaşın da müəyyən limitli həcmi olur və bu həcm dolduqda proqram `OutOfMemoryError` xətası verir. Ona görə də istifadə edilməyən obyektləri heap yaddaşdan silmək lazımdır. Amma bunu bizim əvəzimizə Java özü edir. Bunun üçün Java`da Garbage Collector (GC) mexanizmi var və onun da özünəməxsus işləmə alqoritmləri mövcuddur. Həmin alqoritmlər vasitəsilə istifadə edilməyən obyektlər təyin edilir və heap yaddaşdan silinir.

OCA imtahanı üçün GC alqoritmləri və onların necə işləməsini bilməyə ehtiyac yoxdur. Əsas iki şeyi bilmək lazımdır:

- Obyektlər nə vaxt GC üçün əlçatan (eligible) olur;
- `System.gc()` (və yaxud `Runtime.getRuntime().gc()`) metodu çağırılarkən GC sizə istifadə olunmayan obyektlərin silinəcəyinə zəmanət vermir.

Birinci punktdan başlayaq, obyektin GC üçün əlçatan (eligible) olması nə deməkdir? Bu o deməkdir ki, obyekt artıq istifadə edilmir və o heap yaddaşdan silinə bilər.

Bəs obyekt GC üçün nə vaxt əlçatan olur?

O zaman ki, həmin obyektə heç bir referans işarə etmir. Tutaq ki, bizim `Test` adlı classımız var. Bu classın obyektini yaradaq və baxaq görək bu obyekt hansı hallarda GC üçün əlçatan ola bilər:

```
3:  public class Test {
4:     public static void main(String[] args) {
5:        Test t1 = new Test();
6:        // line6
7:     }
8:  }
```

1-ci hal - əgər line6 sətrinə `t1 = null;` ifadəsini əlavə etsək 5-ci sətirdə yaradılan `new Test();` obyekti artıq *6-cı sətirdə* GC üçün əlçatan (eligible) olacaq. Çünki `t1` referansını `null` etməklə onun 5-ci sətirdə yaradılan obyektlə əlaqəsini kəsirik və həmin obyektə artıq heç bir referans işarə etmədiyindən o istifadəsiz hesab edilir və eligible olur.

2-ci hal - əgər line6 sətrinə `t1 = new Test();` ifadəsini əlavə etsək 5-ci sətirdə yaradılan `new Test();` obyekti yenə *6-cı sətirdə* GC üçün əlçatan (eligible) olacaq. Çünki `t1` referansına yeni obyekt mənimsətməklə onun köhnə obyektlə əlaqəsini kəsirik. Köhnə obyektə artıq heç bir referans işarə etmədiyindən eligible hesab edilir.

3-cü hal - əgər line6 sətrinə heç nə yazmasaq o zaman 5-ci sətirdə yaradılan obyekt *7-ci sətirdə* GC üçün əlçatan (eligible) olacaq. Çünki bütün dəyişənlərin/obyektlərin həyat dövrü (təsir dairəsi) olur. Metod daxilində yaradılan bütün lokal dəyişənlərin həyat dövrü metod bitdikdə bitir. Ümumiyyətlə, dəyişənlərin/obyektlərin həyat dövrü yaradıldığı sətirdə başlayır və daxilində mövcud olduğu kod bloku bitəndə bitir. Ola bilər ki, obyekt metodun daxilində mövcud olan hər hansı bir kod blokunun içərisində yaradılsın. Bu zaman həmin obyektin həyat dövrü metod bitəndə deyil, həmin kod bloku bağlanıldıqda bitəcək.

Nümunələr üzərindən baxaq, o zaman daha aydın olacaq. İlk öncə haşiyəyə çıxaraq bir məqamı qeyd edim ki, GC ilə bağlı suallar ilk vaxtlar mənim üçün də qaranlıq idi. Gələn "feedback"lərdən başa düşürəm ki, imtahana hazırlaşan bir çox insan üçün də bu mövzu çətin və qarışıq gəlir. Əslində bu mövzuya aid sualların tapılması üçün çox sadə yanaşma üsulu var; bunun üçün sizə sadəcə kağız və qələm lazımdır. Bu vasitələrin köməkliyi ilə GC suallarını

çox rahatlıqla tapmaq mümkündür. Nümunəmizə baxaq və sonra kağız-qələm ilə necə rahat tapa bilərik görək:

```
3:  public class MyClass {
4:     public Object getObject() {
5:        Object obj = new String("object");
6:        Object arr[] = new Object[1];
7:        arr[0] = obj;
8:        obj = null;
9:        arr[0] = null;
10:       return obj;
11:    }
12: }
```

5-ci sətirdə yaradılmış obyekt GC üçün nə vaxt eligible olacaq?

A. 6-cı sətirdən sonra
B. 7-ci sətirdən sonra
C. 8-ci sətirdən sonra
D. 9-cu sətirdən sonra
E. 10-cu sətirdən sonra

Addım-addım baxaq. Deməli, 5-ci sətirdə yeni `String` obyekti yaradılır və `obj` referansı həmin obyektə işarə edir. 6-cı sətirdə 1 elementdən ibarət `Object` tipində massiv yaradılır. 7-ci sətirdə isə həmin massivin 0-cı indeksinə 5-ci sətirdə yaratdığımız obyekt mənimsədilir. Deməli, artıq 7-ci sətirdən etibarən 5-ci sətirdə yaratdığımız `new String("object")` obyektinə iki referans işarə edir: `obj` və `arr[0]`. Bu məqama diqqət etmək lazımdır, çünki sualımızın cavabına birbaşa təsir edir.

Artıq 8-ci sətirdə `obj` referansına `null` mənimsədilir. Yuxarıda "1-ci hal"-da qeyd etdik ki, əgər referansa `null` mənimsədilirsə, onun obyektlə əlaqəsi kəsilir. Deməli, 8-ci sətirdən etibarən `obj` referansı 5-ci sətirdə yaradılan `String` obyektinə işarə etmir. O zaman sual yaranır:

Bəs əgər `obj` referansı `String` obyektinə işarə etmirsə, həmin obyekt 8-ci sətirdən etibarən GC üçün eligible olurmu?

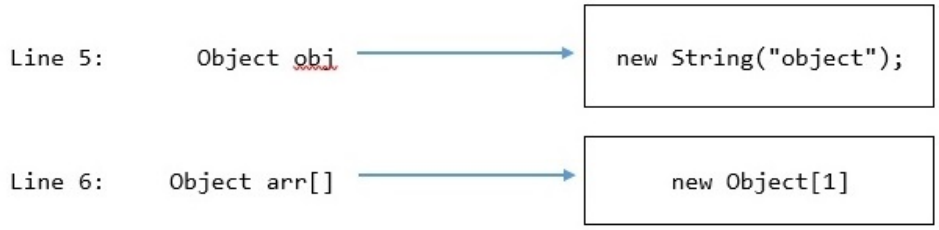
Cavab: **xeyr**. Çünki biz qeyd etmişdik ki, obyekt o zaman GC üçün eligible olur ki, həmin obyektə heç bir referans işarə etməsin. Amma 7-ci sətirdə gördük ki, `arr[0]` referansı da 5-ci sətirdə yaradılan `String` obyektinə işarə edir. 8-ci sətirdə `obj` referansı `null` edilsə də `arr[0]` referansı hələ də həmin `String` obyektinə işarə edir. Bu səbəbdən də 8-ci sətirdə həmin obyekt eligible olmur.

Artıq 9-cu sətirdə `arr[0]` referansı da `null` edilir. Bu o deməkdir ki, artıq 9-cu sətirdən etibarən 5-ci sətirdə yaradılmış `String` obyektinə heç bir referans işarə etmir. Məhz bu səbəbdən də 9-cu sətirdən sonra həmin obyekt eligible olur. Cavab: **D**.

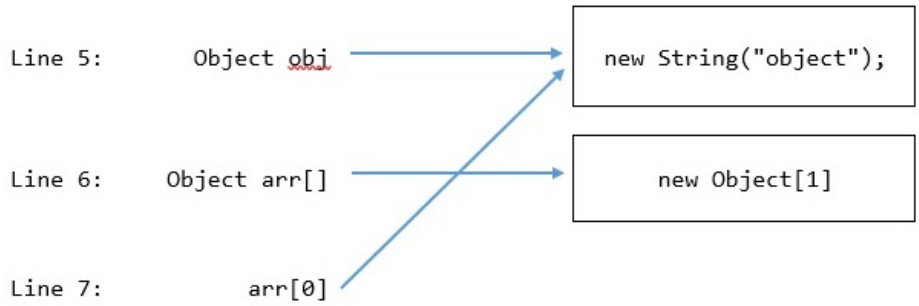
İndi isə kağız-qələm istifadə edərək necə rahat tapmaq olar baxaq. Mən adətən bunu aşağıdakı kimi bir cizgi cızıb tapıram:

Addım 1:

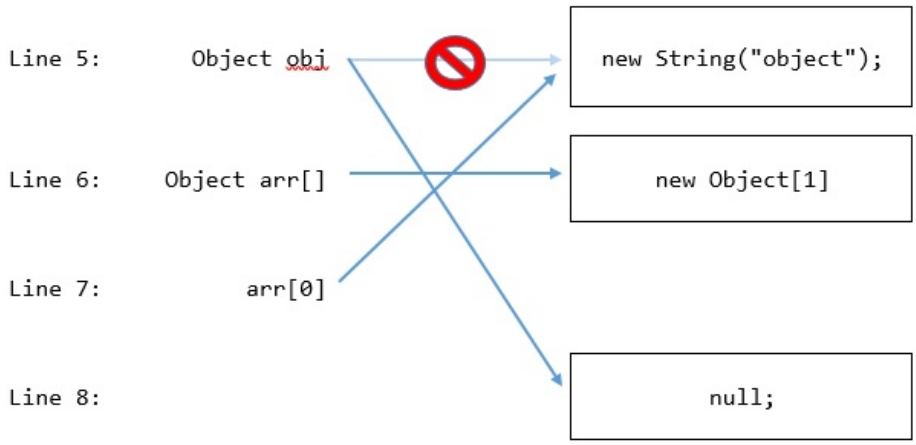
Addım 2:



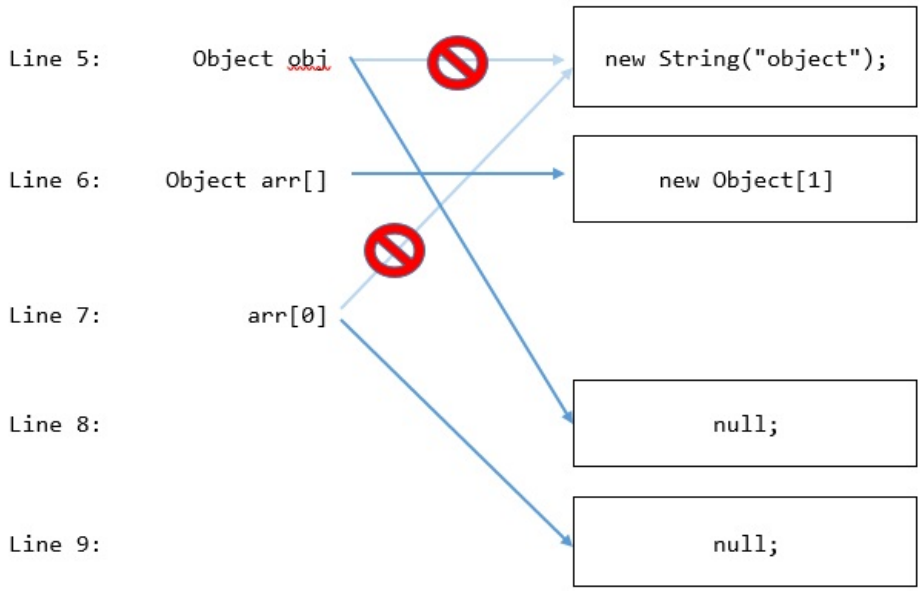
Addım 3:



Addım 4:



Addım 5:



Gördüyünüz kimi line 9`dan sonra line 5`dəki obyektə heç bir referans işarə etmir.

Başqa bir nümunəyə baxaq:

```
3:  public class MyClass {
4:     public static MyClass getResult() {
5:        MyClass name = new MyClass();
6:        return name;
7:     }
8:     public static void main(String[] args) {
9:        MyClass result;
10:       result = getResult();
11:       System.out.println(result);  // MyClass@4e25154f
12:       result = null;
13:    }
14: }
```

5-ci sətirdə yaradılmış obyekt GC üçün nə vaxt eligible olacaq?

Yuxarıda "3-cü hal"-da qeyd etmişdik ki, metod daxilində yaradılan bütün lokal dəyişənlərin/obyektlərin həyat dövrü metod bitdikdə bitir. Bu qayda ilə yanaşdıqda artıq 7-ci sətirdə, 5-ci sətirdə yaradılmış `new MyClass();` obyekti eligible olmalı idi. Amma burada maraqlı istisna var. Sətir 6-da `name` referansının kopyası `getResult()` metodu vasitəsilə sətir 10-dakı `result` referansına mənimsədilir. Bu isə dolayı yolla o deməkdir ki, `result` referansı sətir 5-də yaradılmış `MyClass` obyektinə işarə edir. Bu səbəbdən də həmin obyekt sətir 7-də eligible olmur. Amma sətir 12-də artıq `result` referansına `null` mənimsədilir və onun 5-ci

sətirdə yaradılmış obyektlə əlaqəsi kəsilir. Buna görə də sətir 12-dən sonra 5-ci sətirdə yaradılmış obyekt eligible olur.

İndi isə `System.gc()` metodu ilə bağlı situasiyalara baxaq. Əvvəla qeyd edim ki, biz Garbage Collector`u birbaşa çağıra bilmərik, `System.gc()` metodunu çağırmaq o demək deyil ki, GC mütləq işə düşəcək. Biz bu metodu çağırmaqla sadəcə JVM`ə bununla bağlı istək göndəririk, ancaq bu bizə heç cür zəmanət vermir ki, GC işə düşəcək. GC`nin özünün xüsusi alqoritmləri var, həmin vaxt özü qərar verir ki, işə düşsün ya yox. Ona görə də `System.gc()` metodunun aşkar şəkildə proqramçı tərəfindən çağırılması məsləhət görülmür.

İmtahanla bağlı bilməli olduğunuz digər metod isə `finalize()` metodudur. Bu metod `Object` classına aiddir və onun necə işləməsini görmək üçün override etməyiniz lazımdır. Bəs bu metod necə və nə zaman çağırılır? `finalize()` metodu GC tərəfindən eligible obyektlər mövcud olarsa çağırılır. Sonuncu kod nümunəsi üzərində bir az dəyişiklik edək və bu metodun necə işləməsinə baxaq:

```
public class MyClass {

   String name;

   public MyClass(String name){
      this.name = name;
   }

   public static MyClass getResult() {
      MyClass name = new MyClass("Mushfiq");
      return name;
   }

   public static void main(String[] args) {
      MyClass result;
      result = getResult();
      System.out.println(result);
      result = new MyClass("Alixan");
      System.out.println(result);
      result = null;
      System.gc();
   }

    public String toString() {
       return name;
    }

    @Override
    protected void finalize() throws Throwable {
       System.out.println(this + " object is garbage collected.");
    }
```

```
}
```

Output:

```
Mushfiq
Alixan
Alixan object is garbage collected.
Mushfiq object is garbage collected.
```

Əgər biz main metodun daxilində `System.gc();` metodunu aşkar çağırmasaq o zaman `finalize()` metodu da çağırılmayacaq. Koddan da göründüyü kimi bizim iki eligible obyektimiz var və buna görə də `finalize()` metodu iki dəfə çağırılır. Bir vacib məqamı da qeyd edim, **`finalize()` metodu eyni obyekt üçün ancaq və ancaq bir dəfə çağırılır.** Amma eyni class üçün bir neçə dəfə çağırıla bilər. `finalize()` ilə bağlı başqa bir çaşdırıcı suala da baxaq:

```
public class Test {
   public static void main(String[] args) {
      Test t1 = new Test();
      Test t2 = new Test();
      t1 = t2; t2 = null;
      // garbage collection runs
      t1 = null;
   }
   protected void finalize(Object obj){
      System.out.println("finalize method runs");
   }
}
```

Komment sətirində garbage collection işə düşdükdə nə baş verəcək?

A. `"finalize method runs"` heç vaxt çap edilməyəcək
B. `"finalize method runs"` bir dəfə çap ediləcək
C. `"finalize method runs"` iki dəfə çap ediləcək
D. Kod kompayl olunmur, xəta var

Ağıla ilk **B** variantı gəlir, çünki komment sətirinədək cəmi bir eligible obyekt var (bu arada bunu kağız-qələmlə çox rahat təyin edə bilərsiniz). Amma kodda "trick" var, bu `finalize()` metodunun override olunmuş versiyası deyil, `finalize()` metodu parametr qəbul etmir. Bu overload metoddur, buna da icazə verilir, yəni kod kompayl xətası vermir. Ona görə də cavab **A** variantıdır. Əgər `finalize()` metodu düzgün override olunsa idi cavab **B** variantı olacaqdı.

GC ilə bağlı çox qarışıq, "beyin yandıran" suallar da mövcuddur, amma inanmıram bu tip suallar imtahana salınsın. Həmin sual nümunələrindən bir neçəsi ilə aşağıdakı linklərdən tanış ola bilərsiniz:

- https://coderanch.com/t/644245/certification/CH-objects-eligible-garbage-collection
- https://coderanch.com/t/648140/certification/test-chapter-book-cd

Bu arada qeyd edim ki, `static` dəyişənlərin həyat dövrü proqram başlayanda başlanır və proqram bitəndə bitir. Ona görə də onlar eligible olmur.

Əlavələr

Nümunə 1:

```
1:  class Lost {
2:     public static void main(String args[]) {
3:        Lost e1 = new Lost();
4:        Lost e2 = new Lost();
5:        Lost e3 = new Lost();
6:        e3.e = e2;
7:        e1.e = e3;
8:        e2.e = e1;
9:        e3 = null;
10:       e2 = null;
11:       e1 = null;
12:    }
13:    Lost e;
14: }
```

At what point is only a single object eligible for garbage collection?

A. After line 8 runs
B. After line 9 runs
C. After line 10 runs
D. After line 11 runs
E. Compilation fails
F. Never in this program
G. An exception is thrown at runtime

Bu sual Kathy Sierra və Bert Bates`in "OCA/OCP Java SE 7 Programmer I & II Study Guide" kitabındandır və GC ilə bağlı gördüyüm ən mürəkkəb suallardan biridir. Obyektlərin bu nümunədəki kimi yazılışına "island of objects" və ya "island of isolation" deyilir, zəncirvari şəkildə bir-biri ilə əlaqəlidirlər. Sualda nə vaxt ancaq bir obyektin eligible olacağı soruşulur, cavab isə **F** bəndidir – heç vaxt. Çünki obyektlər bir-biri ilə əlaqəli olduğu üçün 9 və 10-cu sətirlərdə `e3` və `e2` referanslarına `null` mənimsədilməsinə baxmayaraq heç bir obyekt eligible olmur. `e1` referansı vasitəsi ilə hələ də bütün obyektlərə müraciət etmək mümkündür:

- `e1` – 3-cü sətirdə yaradılmış obyektə işarə edir;
- `e1.e` – 5-ci sətirdə yaradılmış obyektə işarə edir;
- `e1.e.e` – 4-cü sətirdə yaradılmış obyektə işarə edir.

Amma 11-ci sətirdə `e1` referansına `null` mənimsədildikdən sonra hər 3 obyekt eyni vaxtda eligible olur. Bu sualın izahını ətraflı şəkildə aşağıdakı linkdən oxuya bilərsiniz:

https://coderanch.com/t/648140/certification/test-chapter-book-cd#2987924

Chapter 2. Operators and Statements

Understanding Java Operators

Java`da 3 tip operator mövcuddur:

- unary
- binary
- ternary

Java`da operatorlar öncəlik sırasına *(operator precedence)* görə dəyərləndirilir və əgər eyni öncəlik sırasına məxsus iki operator bir sətirdə yer alarsa, o zaman hesablanma qaydası adətən *soldan-sağa* doğru icra edilir.

Cədvəl 2.1 Order of operator precedence

Operator	Symbols and examples
Post-unary operators	`expression++, expression--`
Pre-unary operators	`++expression, --expression`
Other unary operators	`+, -, !`
Multiplication/Division/Modulus	`*, /, %`
Addition/Subtraction	`+, -`
Shift operators	`<<, >>, >>>`
Relational operators	`<, >, <=, >=, instanceof`
Equal to/not equal to	`==, !=`
Logical operators	`&, ^, |` *(operate on integral and booleans)*
Short-circuit logical operators	`&&, ||` *(operate only on booleans)*
Ternary operators	`boolean expression ? expression1 : expression2`
Assignment operators	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`

`assignment` operatoru digər operatorlardan fərqli olaraq soldan-sağa doğru deyil, sağdan-sola doğru hesablanır. Eyni bir sətirdə zəncirvari şəkildə bir neçə dəyişənə dəyər mənimsətmək

mümkündür, bir şərtlə ki, həmin dəyişənlər əvvəlcədən elan edilmiş olsunlar. Məsələn, aşağıdakı kod kompayl olunmayacaq:

```
int a = b = c = 100;  // does not compile
```

Əgər `b` və `c` dəyişənini əvvəlcədən elan etsək kod kompayl olunacaq:

```
int b = 0, c = 0;
int a = b = c = 100;
```

Aşağıdakı formada yazılış da səhv deyil:

```
int b, c;            // Not initializing b and c here
int a = b = c = 100; // declaring a and initializing c, b, and a at the same time
```

Sonuncu sətirdə ilk öncə `c` dəyişəninə dəyər mənimsədilir, daha sonra `c`nin dəyəri `b` dəyişəninə mənimsədilir. Ən sonda isə `a` dəyişəninə dəyər mənimsədilir.

Bütün aritmetik/riyazi operatorlar (arithmetic operators → `*`, `/`, `%`, `+`, `-`) `boolean` və `String` tiplərindən başqa istənilən primitiv java tipinə tətbiq edilə bilər. Ancaq `+` və `+=` iki String dəyərini birləşdirmək üçün (*concatenation*) tətbiq edilə bilər.

```
String str = null + "k";
System.out.println(str);    // output: nullk
String str2 = null + null;  // DOES NOT COMPILE
```

Modul əməliyyatlarında (`%`), tutaq ki, bölünən (dividend) – `y` və bölən (divisor) - `x` olarsa:

- `y` – müsbət ədəd olarsa: nəticə həmişə `0` ilə `(x-1)` aralığında olur;
- `y` – mənfi ədəd olarsa: nəticə həmişə `0` ilə `(-x+1)` aralığında olur.

```
System.out.println(8%3);     // output:  2
System.out.println(-8%3);    // output: -2
System.out.println(8%-3);    // output:  2
System.out.println(-8%-3);   // output: -2
```

Nümunədən belə qərara gəlmək olar ki, bölənin (`x`) mənfi olub olmaması nəticəyə heç bir təsir göstərmir, ancaq bölünənin (`y`) qiymətini yoxlayır.

Numeric Promotion

İmtahanda hansı tipin nə qədər yaddaş saxladığını bilməyə ehtiyac yoxdur, amma hansı tipin digərindən daha böyük olduğunu bilmək lazımdır.

Operatorlar dəyişənlərə tətbiq edilərkən aşağıdakı qaydalar mütləq yadda saxlanılmalıdır,

Numeric Promotion Rules:

1. Əgər verilmiş iki dəyər fərqli tiplərdədirsə, java bu iki tipdən birini bunlardan ən yüksək olan digərinin tipinə çevirəcək/yüksəldəcək (promote);
   (ex, short & int → int & int)

   ```
   int x = 1;
   long y = 33;
   ```

   *x*y* → result is *long*

2. Əgər dəyərlərdən biri tam (integral), digəri onluq (floating-point) ədəddirsə, java avtomatik olaraq tam ədədi onluq ədəd tipinə çevirəcək;
   (ex, double & int → double & double)

   ```
   double x = 39.21;
   int y = 2;
   ```

   x+y → result is *double*

3. `byte`, `short` və `char` tiplərində olan dəyişənlər java binary aritmetik operatorlarla istifadə edilərkən əvvəlcə (əməliyyatdan öncə) `int` tipinə çevrilirlər, hətta həmin verilənlərdən heç biri `int` tipində olmasa belə;
   (ex, short + short → int)

   ```
   short x = 10;
   short y = 3;
   ```

   x/y → result is *int*

4. Əgər verilənlər hamısı eyni tipdədirsə, nəticə də eyni tipdə olacaq (istisnalardan başqa);
   (ex, int + int → int)

   ```
   short x = 14;
   float y = 13;
   double z = 30;
   ```

   *x*y/z* → result is *double*

4-cü punktda bütün qaydalar tətbiq olunur. Aritmetik binary operator istifadə edildiyindən `x` avtomatik olaraq `int` tipinə çevrilir. Sonra `y` dəyişəninə vurulduğu üçün `x` avtomatik olaraq `float` tipinə çevrilir. Sonra `x*y` ifadəsi `z` dəyişəninə bölünə bilməsi üçün `double` tipinə çevirilir və bölündükdən sonra son nəticə `double` olur.

Bu mövzu ilə bağlı aşağıdakı linkdən maraqlı kod nümunələrinə baxa bilərsiniz:

http://www.coderanch.com/t/655528/ocajp/certification/Numeric-promotion-rules

Logical Complement and Negation Operators

İnkar operatorunu (negation operator → -) *boolean* ifadəyə, logical complement operatorunu (`!`) isə *numeric* ifadəyə tətbiq etmək mümkün deyil, kompayl xətası baş verir:

```
int x = !5;          // does not compile
boolean y = -true;   // does not compile
boolean z = !0;      // does not compile
```

Digər proqramlaşdırma dillərindən fərqli olaraq Java`da `1` ilə `true` və `0` ilə `false` bir-biri ilə əlaqələnmir.

Increment and Decrement Operators

Əgər operator dəyişəndən (operand) əvvəl gəlirsə, pre-increment (`++a`) və pre-decrement (`--a`) operator adlanır. Bu zaman operator əvvəlcə tətbiq/icra edilir və dəyişənin yeni dəyəri geri qaytarılır. Müvafiq olaraq, əgər operator dəyişəndən (operand) sonra gəlirsə, post-increment (`a++`) və post-decrement (`a--`) operator adlanır. Bu zaman isə əvvəlcə dəyişənin orijinal dəyəri geri qaytarılır və operator dəyər qaytarıldıqdan sonra icra edilir.

```
int x = 3;
int y = ++x * 5 / x-- + --x;
System.out.println("x is " + x);  // x is 2
System.out.println("y is " + y);  // y is 7
```

Addım-addım baxaq:

```
int y = 4 * 5 / x-- + --x;   // x assigned value of 4
int y = 4 * 5 / 4 + --x;     // x assigned value of 3
int y = 4 * 5 / 4 + 2;       // x assigned value of 2
```

Bu sualın izahı əksər oxucular tərəfindən qarışdırılır, hətta Coderanch forumunda da bu nümunənin izahı ilə bağlı 10-dan çox post var. Onlardan 2 ən maraqlısının linkini aşağıda qeyd etmişəm:

- https://coderanch.com/t/649218/certification/wiley-oca-book-error-unary
- https://coderanch.com/t/669054/certification/Priority-Post-Pre-Unary-Operators

`char` ilə bağlı bir nümunəyə baxaq:

```
char c = 'c';
System.out.println(++c);      // output: d
System.out.println(++'c');    // DOES NOT COMPILE
```

Casting Primitive Values

Daha böyük tipdən (numerical) daha kiçik tipə və yaxud onluq saydan tam ədədə keçmək üçün casting istifadə olunur.

```
int x = (int)1.0;
short y = (short)1921222;  // Stored as 20678
int z = (int)9f;
long t = 192301398193810323L;
```

Overflow:

```
int i = 2147483647+1;       // -2147483648
int i2 = 2147483648;        // DOES NOT COMPILE
byte b = (byte) (128+128);  // 0
System.out.println(i+b);    // -2147483648
```

Bir sıra digər maraqlı nümunələrə baxaq:

```
short x = 10;
short y = 3;
short z1 = x * y;           // does NOT compile
short z2 = (short) x * y;   // does NOT compile
short z3 = (short)(x * y);  // does compile

byte b1 = (byte)1000 + (byte)1000;  // output: -48
byte b2 = (byte)100 + (byte)100;    // DOES NOT COMPILE
byte b3 = (byte)100 + (byte)27;     // output: 127
```

Compound Assignment Operators

Compound Assignment Operators dedikdə `+=, -=, *=, /=` və s. operatorlar nəzərdə tutulur.

```
int x = 2, z = 3;
x = x * z;   // Simple assignment operator
x *= z;      // Compound assignment operator

x += 2;
System.out.println(x);  // output: 20
x =+ 2;
System.out.println(x);  // output: 2
```

Compound assignment operatorunun sol tərəfində dayanan dəyişən (x) həmin sətrə kimi artıq elan olunmuş dəyişən olmalıdır, əks halda kompayl xətası verəcəkdir:

```
int y += 2;  // does not compile
```

Əgər bu operator çaşdırıcı formada tərsinə yazılarsa, kompayl xətası verməyəcək!

```
int k =+ 2;    // positive number
```

```
int m =- 2;    // negative number
System.out.printf("k = %d and m = %d %n", k, m);  // k = 2 and m = -2
```

Compound operator həm də ona görə faydalıdır ki, açıq-aşkar cast etməkdən bizi xilas edir:

```
long x = 10;
int y = 5;
y = y * x;          // does not compile
y = (int) y * x;    // does not compile; mötərizə olmadığına görə
y = (int) (y * x);  // line1
y *= x;             // line2
```

line1 və *line2* mahiyyətcə bir-biri ilə eyni koddur. *line2* sətrində `y` əvvəlcə `long` tipinə çevrilir, `long` tipində olan `x` dəyişəninə vurulur və alınan nəticə sonda yenidən `int` tipinə cast olunur.

Daha bir nümunə:

```
double d = 10.8;
int i = 5;
i += d;
System.out.println(i);  // output: 15 not 15.8

short s = 3;
s += 4.6;               // is equivalent to: (short)(s + 4.6);
System.out.println(s);  // output: 7

long x = 5;
long y = (x=3);
System.out.println(x);  // Outputs 3
System.out.println(y);  // Also, outputs 3
```

Logical Operators

Şəkil 2.1 The logical true tables for &, |, and ^

x & y (AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y (INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

Daha yaxşı yadda saxlamaq üçün qaydalar:

- AND (**&**) ancaq o zaman `true` dəyərini ala bilər ki, müqayisə olunan hər iki tərəf `true` olsun.
- Inclusive OR (**|**) ancaq o zaman `false` ola bilər ki, müqayisə olunan hər iki tərəf `false` olsun.
- Exclusive OR (^) ancaq o zaman `true` ola bilər ki, müqayisə olunan tərəflər müxtəlif olsun.

Şərt operatorları **&&** və **||** həm də *short-circuit operators* adlanır, müvafiq olaraq **&** və **|** operatorları ilə eyni işi görürlər, ancaq aralarında çox vacib bir fərq var:

- **&&** və **||** istifadə olunan zaman əgər yekun nəticəni operatorun solundakı (left hand side) ifadəyə əsasən təyin etmək mümkündürsə, o zaman sağdakı (right hand side) ifadə icra edilmir (never evaluate);

```
if(x != null && x.length() < 5){
  // if x is null x.length() doesn't execute, it prevents a NullPointerException
}
```

- Lakin **&** və **|** operatorları ilə iş zamanı soldakı ifadənin nəticəsindən asılı olmayaraq sağdakı ifadə həmişə icra edilir.

```
if(x != null & x.length() < 5){
  // if x is null x.length() throws NullPointerException
}
```

Nümunə:

```
int y = 6;
boolean z = (y >= 6) || (++y <= 7);
System.out.println(y);  // y is 6, because ++y doesn't execute
```

Enthuware test bankında bu operatorla bağlı maraqlı bir sual nümunəsi var və həmin sual nümunəsi coderanch forumunda ətraflı şərh edilib. Müzakirələrə aşağıdakı linkdən baxa bilərsiniz:

```
boolean b1, b2, b3;
b1= b2 = b3 = false;
boolean result = (b1 = true) || (b2 = true) && (b3 = true);
System.out.printf("b1:%b  b2:%b  b3:%b", b1, b2, b3);
```

http://www.coderanch.com/t/640520/ocajp/certification/Evaluating-boolean-expression-assignments-short

Equality Operators

Müqayisə operatoru (`==` and `!=`) əsas bu 3 hal üçün istifadə edilir:

- İki numeric primitiv tipi müqayisə etmək üçün; əgər müqayisə olunan dəyərlər fərqli tipdə olarsa, yüksək olanının tipinə çevirilib sonra müqayisə olunurlar.
  `5 == 5.0` returns true since the left side is promoted to a `double`.
- İki boolean dəyəri müqayisə etmək üçün;
- `null` və `String` dəyərlər də daxil olmaqla iki obyekti müqayisə etmək üçün.

```
boolean x = true == 3;          // does not compile
boolean y = false != "true";    // does not compile
boolean z = 3 == "3";           // does not compile

boolean k = false;
boolean m = (k = true);
System.out.println(m);     // Outputs true
```

İki obyekti müqayisə etdikdə isə obyektlərin özləri deyil onların referansları müqayisə olunur. İki referans ancaq o zaman bərabər olur ki, onlar ya eyni obyektə işarə edir ya da `null`a:

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y);     // false
System.out.println(x == z);     // true

String a = null;
String b = null;
System.out.println(a==b);       // true
```

The if Statement

`if` operatorunun nə üçün istifadə olunduğu çox güman hamıya məlumdur. Ona görə də sadəcə `if` ilə bağlı bir neçə maraqlı nümunəyə baxacağıq:

```
// nümunə 1
int i;
if(true) i = 20;
int k = i;

// nümunə 2
int i;
if(false) i = 20;
int k = i;         // DOES NOT COMPILE

// nümunə 3
int i;
```

```
if(5 > 3) i = 20;
int k = i;

// nümunə 4
int i;
if(5 > 6) i = 20;
int k = i;        // DOES NOT COMPILE

// nümunə 5
boolean condition = true;
int i;
if(condition) i = 20;
int k = i;         // DOES NOT COMPILE

// nümunə 6
final boolean condition;
condition = true;
int i;
if(condition) i = 20;
int k = i;         // DOES NOT COMPILE

// nümunə 7
final boolean condition = true;
int i;
if(condition) i = 20;
int k = i;
```

Əlavə olaraq qeyd edək ki, əgər `if` ifadəsindən sonra fiqurlu mötərizə açılmazsa, `if``dən sonra gələn ancaq bir ifadə ona aid edilir.

`if` fiqurlu mötərizə ilə, ona aid `else` isə mötərizəsiz işlənə bilər:

```
int i = 1;
if(i > 2) { i = 3; }
else i = 2;
```

İmtahanda mən rast gəlməmişəm, amma hər ehtimala qarşı aşağıdakı kod yazılışlarına da baxa bilərsiniz:

```
if(true) else;        // illegal
if(true) if(false);   // valid
```

The if-then Statement (dangling else)

Bəzən imtahanda `if` ifadəsi ilə bağlı çaşdırıcı formatda olan suallarla rastlaşa bilərsiniz. Aşağıdakı nümunəyə baxaq:

```
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = true;

   if(a==true)
   if(b==true)
   if(c==true) System.out.println(1);
   else System.out.println(2);
   else if(a&&(b=c)) System.out.println(3);
   else System.out.println(4);
}
```

Burada hansı `else` ifadəsinin hansı `if` ifadəsinə aid olmasını təyin etmək çətindir. Bu formada verilmiş nümunə *"dangling else"* problemi adlandırılır. Bura qədər biz belə öyrənmişdik ki, `if` ifadəsindən sonra fiqurlu mötərizə açılmazsa, `if`-dən sonra gələn ancaq bir ifadə ona aid edilir. Bu məntiqlə yanaşsaq 1-ci `if`-in tərkibinə ancaq 2-ci `if` ifadəsi aiddir. Eyni məntiqlə, 2-ci `if`-in tərkibinə 3-cü `if` ifadəsi və 3-cü `if`-in tərkibinə də `System.out.println(1);` ifadəsi daxildir. Eyni ardıcıllıqla davam etsək, 1-ci `else` ifadəsi onda 1-ci `if` ifadəsinə məxsus olmalıdır və belə olanda halda da kod kompayl xətası verməlidir, çünki `else if` ifadəsi `else` ifadəsindən sonra gələ bilməz. Amma kod xəta vermədən normal kompayl olunur və console-da alınan nəticə də `3` olur.

Kodun necə işlədiyini daha aydın başa düşmək üçün IDE-də kodu normal formata salıb (Netbeans-də `Alt-Shift-F`) yenidən baxaq, bu zaman hər şey aydın olacaq:

```
public static void main(String[] args) {
   boolean a = true;
   boolean b = false;
   boolean c = true;

   if (a == true) {
      if (b == true) {
         if (c == true) {
            System.out.println(1);
         } else {
            System.out.println(2);
         }
      } else if (a && (b = c)) {
         System.out.println(3);
      } else {
         System.out.println(4);
      }
   }
}
```

Formatdan sonra kodun hansı məntiqlə işlədiyi artıq aydın göründü, qaldı səbəbini bilmək. Səbəbi JLS`də açıq-aşkar qeyd olunub:

"..an else clause belongs to the innermost if to which it might possibly belong."

Yəni *əgər fiqurlu mötərizə yoxdursa, else şərti ən içdəki (innermost) if`ə aiddir*. Əlavə məlumat üçün aşağıdakı linklərə baxa bilərsiniz:

- http://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.5
- http://www.coderanch.com/t/654229/ocajp/certification/Complex-statement

Ternary Operator

`booleanExpression ? expression1 : expression2`

Birinci tərəf mütləq `boolean` ifadə olmalıdır, ikinci və üçüncü tərəflər isə mənimsədilən dəyərin tipinə uyğun olmalıdır. Short-circuit operatorlar kimi ternary operatorda da icra vaxtı "?"-dan sonra ancaq iki şərtdən biri icra edilir:

```
int y = 1;
int z = 1;
final int x = y < 10 ? y++ : z++;
System.out.println(y + "," + z);      // output: 2,1
```

Aşağıda birinci sətirdə istənilən halda ancaq `true` şərtinin yerinə yetirilməsinə baxmayaraq hər iki dəyər `x` dəyişəninin tipinə uyğun olmalıdır. Əgər mənimsətmə yoxdursa, `?` simvolundan sağdakı dəyərlərin bir-biri ilə eyni tipdə olması isə tələb deyil:

```
int x = 6 > 5 ? 6 : "5";               // does not compile
System.out.println(6 > 5 ? 6 : "5");   // output is 6
```

Ternary operatorlarda operatorların yerinə yetirilmə ardıcıllığına diqqət etmək lazımdır. Bir nümunəyə baxaq:

```
String a = "a";
String b = "b";
final String letter = 213 > 321 ? a : b = "c";  // DOES NOT COMPILE
```

Baş verən xəta operatorların icra ardıcıllığından qaynaqlanır. Assignment operator (`=`) ternary operatordan (`?:`) sonra icra olunur. Daha aydın olması üçün sonuncu sətri bir az formatlaşdırılmış şəklə salaq:

```
final String letter = (213 > 321 ? a : b) = "c";
final String letter = "b" = "c";
```

Dəyəri dəyərə mənimsətmək mümkün olmadığından kod kompayl olunmur. Əgər "`b`" əvəzinə `b` olsa və yaxud ilk kod nümunəsindəki (`b = "c"`) mötərizə içində yazılsa kod kompayl olunar:

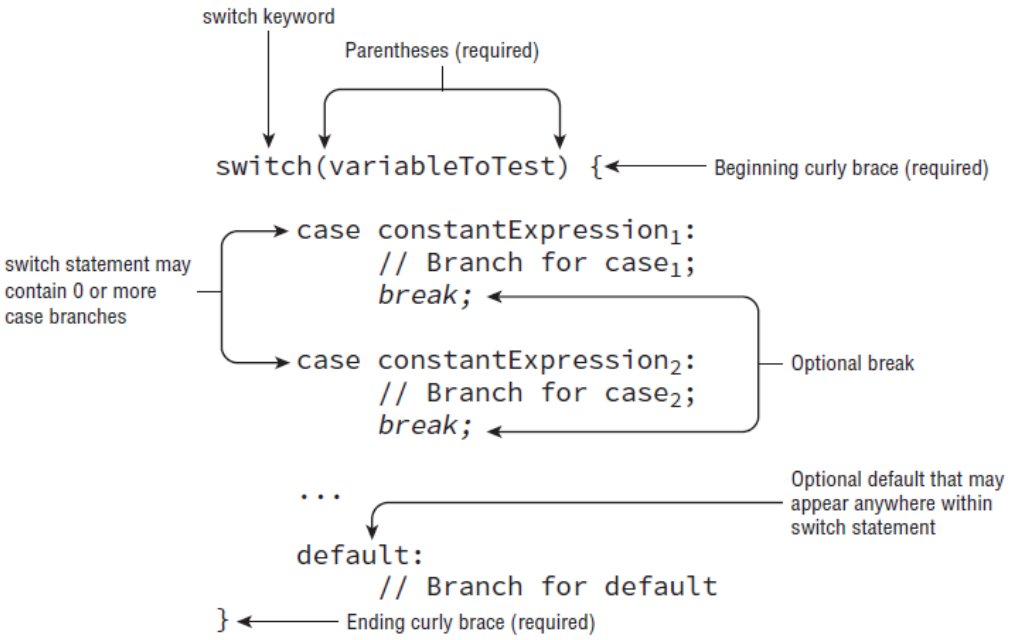
```
final String letter = b = "c";
final String letter = 213 > 321 ? a : (b = "c");
```

`void` tipində metodların ternary operator ilə istifadəsinə icazə verilmir:

```
class TestTernaryOperator {

   public static void main(String[] args) {
      System.out.println(6>5 ? return1() : return2()); // output: I am String
      System.out.println(6>5 ? void1() : void2());     // does NOT compile
   }

   static String return1(){ return "I am String"; }
   static int return2(){ return 7; }

   static void void1(){ System.out.println("I am void 1"); }
   static void void2(){ System.out.println("I am void 2"); }
}
```

The switch statement

Şəkil 2.2 switch ifadəsinin quruluşu



Switch ifadələr ancaq aşağıdakı tipdə ola bilər:

- `int` və `Integer`
- `byte` və `Byte`

- `short` və `Short`
- `char` və `Character`
- `String` (java 7`dən bəri)
- `enum` values

`float, double, boolean` və `long` tipləri və həmçinin onların wrapper classları (`Float, Double, Boolean, Long`) switch ifadələr tərəfindən dəstəklənmir.

Mahiyyət baxımından bir iş görməsə belə aşağıdakı nümunədə verilmiş sintaksis səhv deyildir, normal kompayl olunur və icra edilir:

```
public static void main(String[] args) {
   switch (1) {
      default: break;
   }
   switch (2) { }
}
```

Compile-time Constant Values

Hər bir `case` ifadəsinin dəyəri `switch` ifadəsinin dəyəri ilə **eyni tipdə olan** (əgər `switch` dəyişəni `String` tipindədirsə, `case` ifadəsinin də dəyəri `String` tipində olmalıdır və ya `switch` ifadə `char` tipindədirsə `case` dəyərlər 65535-dən yuxarı ola bilməz və s.) *compile-time constant* dəyərlər olmalıdır. Bunlar ola bilər:

- literals;
- enum constants;
- final constant variables (`final` modifier ilə təyin olunmalı və elan olunduğu sətirdə literal dəyər mənimsədilməlidir; **primitiv** və `String` tiplər ola bilər, Wrapper classlar constant hesab edilmir).

Sonuncu bəndə əsasən wrapper classlar (`Integer, Byte, Short, Character`) `switch` ifadədə işlədilə bilər, amma case label kimi işlədilə bilməz. Suallarda buna diqqət etmək lazımdır.

Qısaca *compile-time constant* olması üçün dəyişən aşağıdakı şərtləri ödəməlidir:

- `final` olmalıdır;
- primitiv və ya `String` tipində olmalıdır;
- elan olunduğu sətirdə dəyər mənimsədilmiş olmalıdır;
- dəyişənə "compile-time constant" dəyərlər mənimsədilməlidir (be assigned to a compile time constant expression).

Misal üçün `private final int x = getX();` ifadəsində `x` compile-time constant hesab edilmir, çünki compile-time constant dəyər mənimsədilmir, metod çağırılır.

Başqa bir nümunəyə baxaq:

```
private int getSortOrder(String firstName, final String lastName) {
   String middleName = "MBM";
   final String suffix = "MM";
   int id = 0;

   switch(firstName) {
      case "Test":             // String literal
         return 52;
      case suffix:             // final constant variable
         id = 7;
         break;
      case middleName:         // does not compile, beacuse it is not final
         id = 5;
         break;
      case lastName:           // does not compile, final but not constant
         id = 8;
         break;
      case 5:                  // does not compile, not String
         id = 5;
         break;
      case 'J':                // does not compile, not String
         id = 10;
         break;
      case java.time.DayOfWeek.SUNDAY.toString(): // does not compile, enum
         id = 15;
         break;
      case "string".toUpperCase():  // does not compile
         break;
      case "string".toString():     // does not compile
         break;
      case "string".trim():         // does not compile
         break;
   }
   return id;
}
```

Bir sıra incə məqamlara da nəzər salaq. Misal üçün aşağıdakı nümunə kompayl xətası verəcək, çünki `char` birbaşa `Integer` tipində olan dəyişənə mənimsədilə bilməz:

```
Integer x = 1;  // int x = 1; is valid.
switch (x) {
   case 'a':    // does not compile
      System.out.println("a");
}
```

Bildiyimiz kimi `char` dəyişəninə rəqəm (numeric) tipli dəyər mənimsədə bilərik. Bu məntiqlə yanaşsaq deyə bilərik ki, `switch char` tipindədirsə `case byte` tipində ola bilər. Amma bu hər

zaman doğru deyildir, çünki `byte` mənfi ədədlər də ala bilər, `char` isə mənfi dəyər qəbul edə bilməz:

```
char c = 'a';
switch (c) {
   case -1: System.out.println("-1"); // Doesn't compile: "possible loss of precision"
}
```

Əgər `case` ifadədə `-1` əvəzinə `(char)-1` yazsaq kompayl olunacaq.

Həmçinin də `switch` ifadə əgər `byte` tipindədirsə, `case` dəyərlər `-128 < case_label < 127` aralığında dəyər ala bilərlər.

```
byte b = 100;
switch (b) {
   case 100:
   case 'a': default: case 'b':
   case 150:  // does not compile
   case 200:  // does not compile
}
```

İmtahan üçün `char` dəyərlərin rəqəm qarşılığını bilmək tələb olunmur, amma ümumi olaraq yadda saxlaya bilərsiniz ki, `0-9` aralığındakı rəqəmlərin və eləcə də ingilis əlifbasının bütün böyük və kiçik hərflərinin hamısının rəqəm qarşılığı 127-dən kiçikdir. Yəni əgər `switch` ifadə `byte` tipindədirsə, `case` label olaraq qeyd etdiyimiz bu `char` dəyərlərin hər birini rahatlıqla istifadə edə bilərsiniz. ASCII kod siyahısına aşağıdakı linkdən baxa bilərsiniz:

http://www.asciitable.com/

Case dəyərlər təkrarlana bilməz, əks halda təkrarlanan sətir xəta verəcək:

```
switch (new Integer(2)) {
   case 1: System.out.println("1");
   case 2: System.out.println("2");
   case 3: System.out.println("3");
   case 2: System.out.println("duplicate 2");    // does not compile
   case 1: System.out.println("duplicate 1");    // does not compile
}
```

Nümunə bir az qarışıq formada da verilə bilər:

```
switch (0) {
   case 97: System.out.println("97");
   case 16: System.out.println("2");
   case (int)'b': System.out.println("98");
   case (int)'a': System.out.println("duplicate 97");  // does not compile
   case 8*2: System.out.println("duplicate 16");       // does not compile
}
```

Yaxşı olar ki, təsəvvürünüzə gələn bütün mümkün variantları yazıb özünüz üçün test edəsiniz.

Bir switch blokunda `default` ifadəsi ancaq bir dəfə istifadə oluna bilər, əks halda kod xəta verəcək. Default ifadəsi switch`də istənilən yerdə yazıla bilər; əvvəldə, ortada, axırda, case ifadələri ilə yerləşmə ardıcıllığında heç bir məhdudiyyət yoxdur. Amma icra olunma ardıcıllığı fərqlidir, default əvvəldə və ya ortada yerləşməsindən asılı olmayaraq həmişə sonda icra olunur. Əvvəlcə case ifadələri bir-bir yoxlanılır və əgər heç bir uyğunluq olmasa o zaman default ifadəsi axtarılır və əgər varsa o zaman default icra olunur. Və ən vacib məqamlardan biri: əgər uyğunluq hansı sətirdə tapılsa, **həmin sətirdən başlayaraq switch ifadəsinin sonuna kimi** `break` ifadəsi olmazsa bütün `case` və `default` ifadələrinə daxil olur.

```
int dayOfWeek = 5;
switch(dayOfWeek) {
   case 0:
      System.out.print(" Sunday");
   default:
      System.out.print(" Weekday");
   case 6:
      System.out.println(" Saturday");
}
```

Output:

```
Weekday Saturday
```

Başqa bir maraqlı nümunəyə baxaq:

```
public void displayTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
   String typeOfDay;
   switch (dayOfWeekArg) {
      case "Monday":
         typeOfDay = "Start of work week";
         break;
      case "Tuesday":
      case "Wednesday":
      case "Thursday":
         typeOfDay = "Midweek";
         break;
      case "Friday":
         typeOfDay = "End of work week";
         break;
      case "Saturday":
      case "Sunday":
         typeOfDay = "Weekend";
         break;
      default:
```

```
            throw new IllegalArgumentException("Invalid day of the week");
        }
        System.out.println(typeOfDay);  // line A
    }
```

Əgər bu metodda istənilən `case` ifadəsinin birində `typeOfDay` dəyişəninə dəyər mənimsədilməsini və ya `default`da `throw` ifadəsini kommentə atsaq *`line A`* kompayl olunmayacaq. Çünki kompilyator yüz faiz əmin olmalıdır ki, istənilən hər bir situasiyada `typeOfDay` dəyişəninə dəyər mənimsədiləcək, əks halda lokal dəyişən olduğu üçün onu çap edə bilməz.

Switch ifadələrə göndərilən `String` obyektlərin case labela uyğun olub olmadığını yoxlamaq üçün Java `equals()` metodundan istifadə edir. Ümumiyyətlə, əgər müqayisə ediləcək şərtlərin sayı çoxdursa, o zaman `if-then-else` ifadəsi ilə müqayisədə `switch` ifadəsinin istifadə edilməsi daha məqsədəuyğundur. Çünki `switch` ifadəsi həm `if-then-else` ifadəsinə nisbətən daha oxunaqlıdır, həm də performans baxımından daha yaxşıdır. Çünki dərləmə (*compilation*) zamanı `switch` üçün *"jump table"* yaradılır və artıq icra (*execution*) zamanı hansı case label`ın uyğun gəldiyi yoxlanılmır, sadəcə hansı case`in icra olunacağına qərar verilir. Əlavə məlumat üçün aşağıdakı linkə baxa bilərsiniz:

https://goo.gl/iFjldb

Bəzən `switch` sintaksislərində *"case else"* label`ına rast gələ bilərsiniz. Switch ifadələrində bu cür sintaksis mövcud deyil, `if`dəki `else` ilə bunu çaşdırmayın. *"case else"* əvəzinə *"default"* label istifadə olunur.

Switch ifadələrində `enum` da istifadə etmək mümkündür. Baxmayaraq ki, enum OCP imtahanı mövzularına daxildir, hər ehtimala qarşı tanış olmaqda fayda var. Təsəvvür üçün bir nümunəyə baxaq:

```
class TestEnumWithSwitch {
    public static void main(String[] args) {
        int i = 1;
        switch (Seasons.SUMMER) {
            default:     i++;
            case WINTER: i+=1;
            case SPRING: i+=2;
            case SUMMER: i+=3;
            case AUTUMN: i+=4;
        }
        System.out.println(i);  // output: 8
    }
}
enum Seasons { WINTER, SPRING, SUMMER, AUTUMN }
```

Bəzi suallarda maksimum diqqətli olmaq lazımdır, *"sağı göstərib sol ilə də vura bilərlər"*. Ani bir diqqətsizlik üzündən çox sadə bir sualı səhv də cavablandıra bilərsiniz. Aşağıdakı nümunəyə baxaq:

```
4.  int i = 8 / 3 - 1;
5.  switch(i){
6.     case: 0 System.out.print("0"); break;
7.     case: 1 System.out.print("1"); break;
8.     default: System.out.println("default");
9.  }
```

Bu testdə ilkin olaraq diqqət 4-cü sətirə yönəldilir. `switch` ifadə `double` tipində dəyər qəbul edə bilməz, adətən bu prizmadan yanaşıb qərar qəbul edirik:

1) ya 4-cü sətir kompayl olunmur;
2) yaxud da 4-cü sətir kompayl olunur və i`nin dəyərinə əsasən hansı case label çap olunur onu təyin edirik.

Əslində isə səhv tamam başqa yerdədir: **6 və 7-ci sətirlərdə**. case label`larda sintaksisə görə iki nöqtə case`in dəyərindən sonra yazılır, case`dən sonra yox. Bu səbəbdən kod kompayl olunmur.

Son olaraq `switch` ilə bağlı *"Java sertifikat sualları"* facebook qrupumuzda müzakirə edilmiş daha bir maraqlı sualı qeyd edirəm, cavabını özünüz təxmin etməyə çalışın:

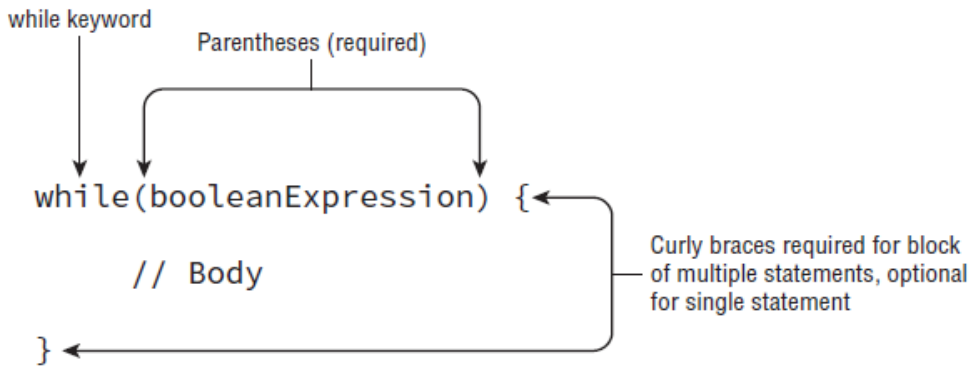
```
class SwitchExample {
   public static void main(String[] args) {
      String str = null;
      switch (str) {
         case "null":
            System.out.println("1");
            break;
         case "":
            System.out.println("2");
            break;
         default:
            System.out.println("3");
      }
   }
}
```

Əgər təxminləriniz olmazsa, doğru cavab və izahı görmək üçün aşağıdakı linkə daxil ola bilərsiniz:

https://www.facebook.com/groups/javacertification/permalink/917643694951436/

The while Statement

Şəkil 2.3 while ifadəsinin quruluşu



Struktur olaraq `while` ifadəsi ancaq geriyə *"boolean"* dəyər qaytaran ifadələri qəbul edir. `for` dövründən fərqli olaraq siz `while` dövründə mötərizə (parentheses) içərisində dəyişən elan edə, dəyişənə dəyər mənimsədə (*instantiation*) və ya dəyişənin dəyərini artıra (*increment*) bilməzsiniz:

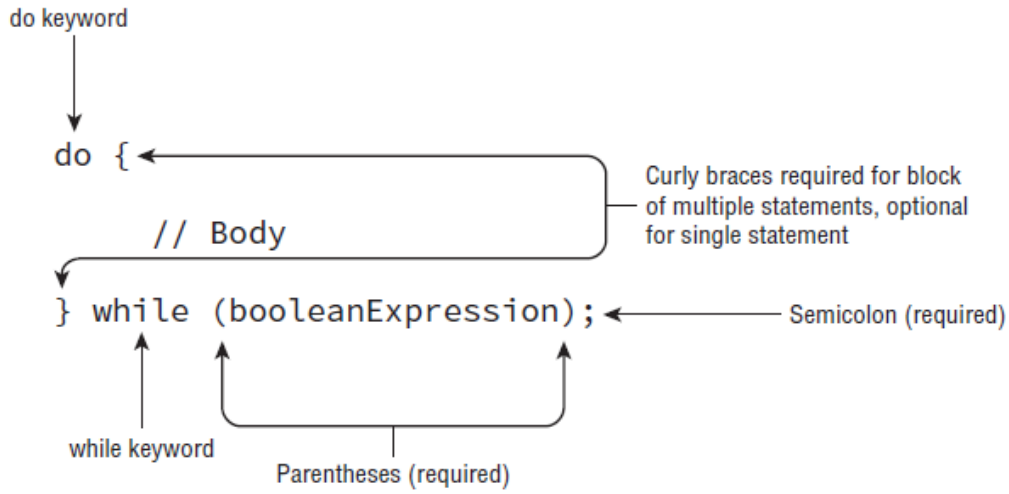
```
for(int k=5; k<7;)     // is valid
while(int k=5; k<7;)   // is invalid
```

Əgər şərt düzgün təyin edilməzsə, sonsuz dövr baş verə bilər:

```
int x = 2;
int y = 5;
while (x < 10)  // infinite loop
  y++;
```

The do-while Statement

Şəkil 2.4 do-while ifadəsinin quruluşu

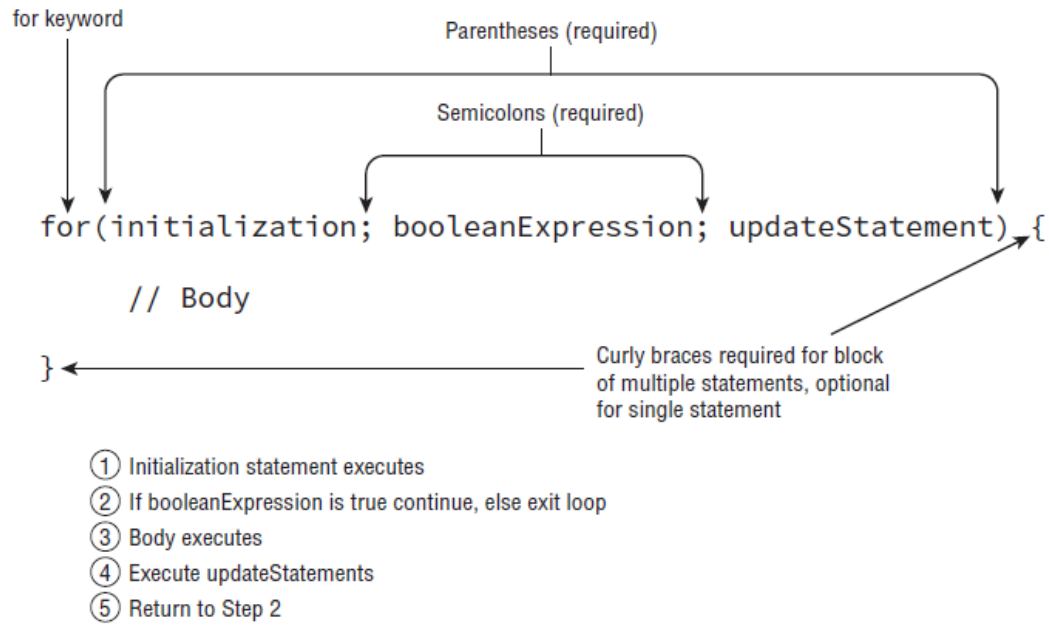


While dövründən fərqli olaraq *do-while* dövrü sizi sığortalayır ki, ifadə və ya blok ən azı bir dəfə icra olunacaq, çünki java birinci body hissəni icra edir və sonra şərti yoxlayır.

```
if (z > 10)
   do ;   // nöqtəli-vergülə görə kod kompayl olunur, do`dan sonra mütləq bir ifadə olmalıdır
   while (z > 10);
```

The for Statement

Şəkil 2.5 sadə for dövrünün quruluşu



Göründüyü kimi `for` ifadəsi 3 hissədən ibarətdir:

1. initialization;
2. booleanExpression;
3. updateStatement.

1-ci və 3-cü hissə bir və ya bir neçə ifadədən (multiple statements) ibarət ola bilər və bu zaman həmin ifadələr bir-birindən vergül ilə ayrılır.

Əgər dəyişən initalization blokda (1-ci hissə) elan edilibsə, həmin dəyişənə ancaq `for` dövrünün içində müraciət edilə bilər. Əgər dəyişən `for` dövründən əvvəl elan edilib, initalization blokda dəyər mənimsədilibsə, həmin dəyişənə `for` dövründən kənarda da müraciət edilə bilər.

Yuxarıda icra olunma ardıcıllığı qeyd olunub, updateStatement ++a yaxud a++ olmasından asılı olmayaraq body hissə icra olunduqdan sonra icra olunur.

İmtahanda rastlaşa biləcəyimiz nümunələr:

1. Creating an Infinite Loop

```
for( ; ; ){
   System.out.println("This is infinite loop");
}
System.out.println("This line is unreachable because of infinite loop");
```

Nümunədən də göründüyü kimi `for` dövrünün bütün komponentləri optional`dı, for (;) və for ( ) şəklində yazılsa kompayl xətası verəcək. Bundan əlavə sonsuz dövrdən sonra gələn bütün ifadələr də kompayl xətası verir.

2. Adding Multiple Terms to the for Statement

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
   System.out.print(y + " ");
}
System.out.println(x);
```

3. Redeclaring a Variable in the Initalization Block

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {  // does not compile
   System.out.println(x + " ");
}
```

Aşağıdakı kod isə kompayl olunur:

```
int x = 0;
long y = 10;
```

```
for(y = 0, x = 4; x < 5 && y < 10; x++, y++) {
   System.out.println(x + " ");
}
```

4. Using Incompatible Data Types in the Initialization Block

```
for(long y=0, int x=4; x<5 && y<10; x++, y++) {   // does not compile
   System.out.println(x + " ");
}
```

Initalization blokda elan olunan dəyişənlər hamısı eyni tipdə olmalıdır.

5. Using Loop Variables Outside the Loop

```
for(long y=0, x=4; x<5 && y<10; x++, y++) {
   System.out.println(y + " ");
}
System.out.println(x);   // does not compile
```

`for` dövrünün 3-cü hissəsində *(update part)* hər ifadəni yazmağa icazə verilmir. Ancaq aşağıdakı ifadələri istifadə etmək mümkündür:

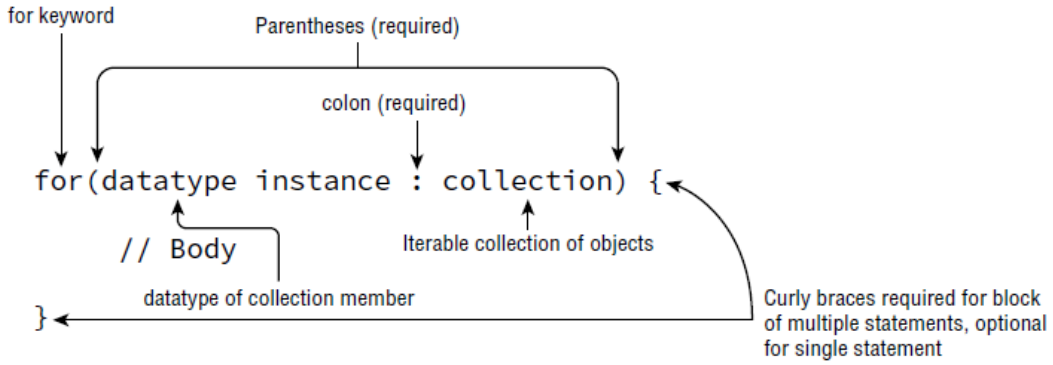
- ✓ Assignment;
- ✓ PreIncrementExpression;
- ✓ PreDecrementExpression;
- ✓ PostIncrementExpression;
- ✓ PostDecrementExpression;
- ✓ MethodInvocation;
- ✓ ClassInstanceCreationExpression.

Nümunələrə baxaq:

```
for( ; Math.random() < .07; ){ }
for( ; ; Math.random() < .07){ }  // does NOT compile
for( ; ; Math.random()){ }
for( ; Math.random() < .05 ? true : false; ){ }
for ( ; ; ) {
   Math.random() < .05 ? break : continue;  // does NOT compile
}
```

The for-each statement

Şəkil 2.6 for-each (enhancement for) dövrünün quruluşu



Sağ tərəf (collection) massiv tipində və ya `java.lang.Iterable` interfeysindən implements olunmuş classlar (xüsusilə Java Collections) olmalıdır. Sol tərəf (instance) isə sağ tərəfdəki collection`un tipində olan dəyişən. Məsələn;

collection → `String [] arr, ArrayList<Integer> list`

instance → `String s, Integer i`

`for` dövründən fərqli olaraq `forEach` dövründə **instance dəyişəni dövrdən əvvəl təyin etmək olmur**, ancaq içəridə elan olunmalıdır.

```
final String[] names = new String[3];
names[0] = "Murad";
names[1] = "İlkin";
names[2] = "Elmar";
for(String name: names){
   System.out.print(name + ", ");    // Murad, İlkin, Elmar,
}
String name2;
for (name2 : names) {            // does not compile, identifier expected
   System.out.println(name2);
}
names = new String[5];           // does not compile, because of final
```

`for`&`forEach` qarşılaşdırması:

```
String[] names = {"Ceyhun", "Elnur", "Hasil", "Adil"};
List<Integer> values = (List<Integer>) Arrays.asList(new Integer[]{6, 5, 4});
for (String name: names) {
   System.out.print(name + ", ");
}
for (int i = 0; i < names.length; i++) {
   String name = names[i];
   System.out.print(name + ", ");
}
```

```
for (int value: values) {
   System.out.print(value + " ");
}
for (java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {
   int value = i.next();
   System.out.print(value + " ");
}
```

`java.util.Iterator` classı istifadə edildikdə *updateStatement* bölməsinə ehtiyac yoxdur.

Digər nümunələr:

```
Integer[] i = new Integer[3];
for (int i: i) {             // does not compile, eyni dəyişən adına görə
   System.out.println(i);
}
for (int a: i) {             // does compile, but throw NullPointerException
   System.out.println(a);
}
for (Integer a: i) {         // does compile and output: null null null
   System.out.print(a + " ");
}
```

forEach massiv və ya `Collection` tipində dəyərlər (başqa sözlə `java.lang.Iterable` interfeysini implements etmiş obyektlər) qəbul edir. Map`ı forEach dövründə birbaşa istifadə etmək mümkün deyil, çünki Map `Iterable` interfeysinin varisi deyil. Amma Map`ın `keySet()` və ya `values()` metodlarından istifadə etməklə onu forEach dövründə işlətmək olar, çünki bu metodlar geriyə `Collection` qaytarır.

Modifier`lərdən ancaq `final` açar sözü forEach dövründə istifadə edilə bilər.

```
String[] banks = {"CBAR", "IBAR", "Unibank"};
String s1;
for(s1: banks){}    // does NOT compile
for(final String s2: banks){}
```

Adding Optional Labels

Label adətən if, switch ifadələri və dövrlərlə istifadə olunur. Oxunaqlı olması üçün və adi dəyişənlərdən fərqlənsin deyə label adətən böyük hərflərlə yazılır və sözlərin arasında altxətt (underscore) istifadə olunur:

```
int[][] complexArray = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
OUTER_LOOP: for (int[] simpleArray : complexArray) {
   INNER_LOOP: for (int i = 0; i < simpleArray.length; i++) {
```

```
            System.out.print(simpleArray[i] + "\t");
        }
        System.out.println();
    }
```

İstənilən blok ifadələri ilə label istifadə etməyə icazə verilir, amma "declaration"larda icazə verilmir:

```
INSTANCE_INITIALIZER_BLOCK: {
    System.out.println("");
}
DECLARATION: int i = '1';  // does NOT compile
```

Dövr və ya hər hansı bir kod bloku daxilində labeli sonlandırmaq üçün `break` açar sözündən istifadə edilir. Amma burada bir məqama diqqət etmək lazımdır: hansı labeli sonlandırmaq istəyiriksə, *"`break label`"* ifadəsi mütləq həmin kod blokunun daxilində olmalıdır, əks halda kompayl xətası verəcəkdir:

```
void testLabelBreak() {
    int c = 0;
    LABEL_1: while (c < 8) {
        LABEL_2: System.out.println(c);
        if (c > 3) break LABEL_2;  // does NOT compile
        else c++;
    }
}
```

Əgər `break LABEL_2;` əvəzinə `break LABEL_1;` və yaxud təkcə `break;` olsa idi, kompayl olardı. Həmçinin aşağıdakı formada da yazsaq kompayl olacaq:

```
LABEL_1: { System.out.println(c); if(c>3) break LABEL_1; }
LABEL_1: { if(c > 3) break LABEL_1; }
```

Açar söz label kimi istifadə oluna bilməz:

```
for: { ; } // does NOT compile, because of "for" is keyword, but "For" is ok
```

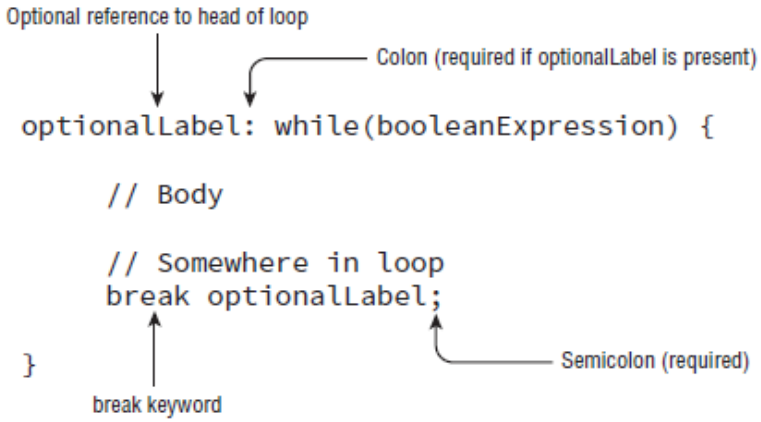
Enthuware test bankında belə bir maraqlı kod nümunəsi çıxacaq qarşınıza:

```
String String = "";  // This is valid.
String: for (int i = 0; i < 10; i++){  // This is valid too!
    for (int j = 0; j < 10; j++) {
        if (i + j > 10) {
            break String;
        }
    }
    System.out.println("ok");
}
```

`String` açar söz (keyword) hesab edilmir. Java`da açar sözlər kiçik hərflərlə başlayır.

The break statement

Şəkil 2.7 break ifadəsinin quruluşu



`break` ümumilikdə dövrlərdə (loop) və switch ifadələrində işlədilir. Onu təkcə `if`in içində işlətdikdə kompayl xətası verir, `if`in içində işlətsək və həmin `if` özü də dövrün içində olsa, o zaman normal kompayl olunacaq:

```
for(;;)
   if(true){
      System.out.println("Infinite loop break");
      break;
   }
```

Nümunə:

```
public static void main(String[] args) {
   int[][] array = { {1,13,5}, {1,2,5}, {2,7,2} };
   int searchValue = 2;
   int px = -1;
   int py = -1;
   PARENT_LOOP: for(int x=0; x < array.length; x++){
      for(int y=0; y < array[x].length; y++){
         if(array[x][y] == searchValue){
            px = x;
            py = y;
            /*Insert code*/      // line1
          }
       }
    }
```

```
        if(px == -1 || py == -1)
            System.out.println("Value "+searchValue+" not found!");
        else
            System.out.println("Value "+searchValue+" found at: ("+px+", "+py+")");
    }
```

Əgər line1`ə aşağıdakı ifadələr əlavə olunarsa nəticə belə dəyişəcək:

- `break PARENT_LOOP;` → Value 2 found at: `(1,1);`
- `break;` → Value 2 found at: `(2,0);`
- remove `break` → Value 2 found at: `(2,2);`

Aşağıdakı nümunəyə Coderanch forumunda rast gəldim, deməli `if` ifadəsi əgər dövrün (loop) içində deyilsə, `break label;` işləyəcək, amma tək `break;` işləməyəcək:

```
void test1(){
   int value = 2;
   label:
   if (value > 1) {
      if (value <= 2) {
         break label;   // əgər "label" yazılmasa tək break; kompayl olunmayacaq
      }
      System.out.println("More than two");
   }
   System.out.println("Done");
}

void test2(){
   int value = 2;
   if (value > 1) {
      if (value <= 2) {
         break;   // DOES NOT COMPILE
      }
      System.out.println("More than two");
   }
   System.out.println("Done");
}
```

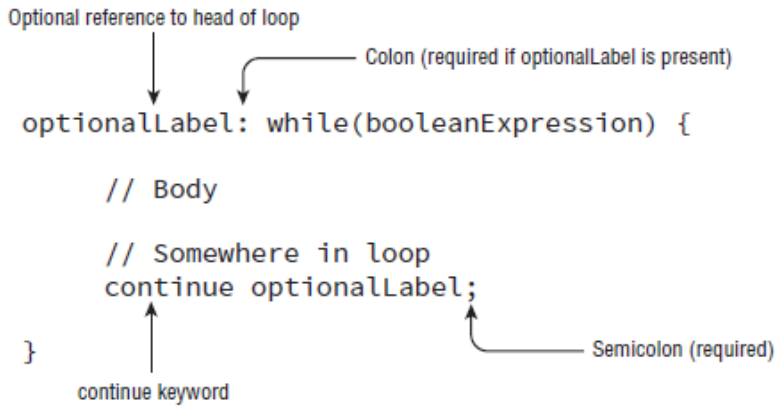
Yadda saxlamaq lazımdır ki, "`break label;`" və ya "`continue label;`" ifadələri mütləq label elan olunan dövrün (loop) daxilində olmalıdır. Label`in xaricindən həmin label`a müraciət etmək mümkün deyil:

```
public void method1(int i) {
   int j = (i * 30 - 2) / 100;
   POINT1: for (; j < 10; j++) {
      boolean flag = false;
      while (!flag) {
```

```
            if (Math.random() > 0.5) break POINT1;
         }
      }
      while (j > 0) {
         System.out.println(j--);
         if (j == 4) break POINT1;    // does NOT compile
      }
   }
```

The continue statement

Şəkil 2.8 continue ifadəsinin quruluşu



Nümunə:

```
FIRST_LOOP: for (int i = 1; i <= 4; i++) {
   for (char j = 'a'; j <= 'c'; j++) {
      if (i == 2 || j == 'b') {
         /*Insert code*/       // line1
      }
      System.out.print(" " + i + j);
   }
}
```

Əgər line1`ə aşağıdakı ifadələr əlavə olunarsa nəticə belə dəyişəcək:

- `continue FIRST_LOOP;` → `1a 3a 4a`
- `continue;` → `1a 1c 3a 3c 4a 4c`
- remove `continue` → `1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c`

Yuxarıda qeyd etdiklərimizi yekunlaşdırsaq *label*, `break` və `continue` ifadələrinin istifadəsi ilə bağlı aşağıda görünən mənzərə yaranacaq:

	Label istifadə etməyə icazə verilirmi?	`break` istifadə etməyə icazə verilirmi?	`continue` istifadə etməyə icazə verilirmi?
`if`	Bəli*	Xeyr	Xeyr
`while`	Bəli	Bəli	Bəli
`do while`	Bəli	Bəli	Bəli
`for`	Bəli	Bəli	Bəli
`switch`	Bəli	Bəli	Xeyr

* Labels are allowed for any block statement, including those that are preceded with an if-then statement.

Unreachable and dead code

"Unreachable" və *"dead code"* ilə bağlı Coderanch forumunda maraqlı kod nümunələri mövcuddur, aşağıdakı linkdən baxa bilərsiniz:

https://coderanch.com/t/649689/certification/continue-break-loop-generate-compilation#2998168

Qısaca `while(false){i=7;}` "*unreachable*" kod hesab olunur və kompayl olunmur, amma `if(false){i=7;}` normal kompayl olunur.

Əlavələr

Bəzi qarışıq testlərin izahı

Nümunə 1:

```
1: import static java.lang.System.*;
2: class _ {
3:    static public void main(String[] __A_V_) {
4:       String $ = "";
5:       for (int x = 0; ++x < __A_V_.length;)
6:          $ += __A_V_[x];
7:       out.println($);
8:    }
9: }
```

And the command line:

```
java _ - A .
```

What is the result?

A. `-A`
B. `A.`
C. `-A.`
D. `_A.`
E. `_-A.`
F. Compilation fails
G. An exception is thrown at runtime

İzahı: Başlayaq 1-ci sətirdən; `java.lang` paketi Java tərəfindən bütün classlara default olaraq import olunur, onu ayrıca aşkar şəkildə import etməyə ehtiyac yoxdur. Bu nümunədə adi importdan fərqli olaraq **static import**'dur, yəni `System` classının içində mövcud olan bütün static üzvlər (members) import olunur. Bu səbəbdən də 7-ci sətirdə dəyişəni print etmək üçün `System` class adını istifadə etmədik. Əgər import sətrini kommentə atsaq, o zaman 7-ci sətir kompayl xətası verəcək.

İkinci diqqət çəkən məsələ identifier adlarıdır. Java`da identifier adları hərflə və yaxud da **$** və ya _ (underscore) simvollarından biri ilə başlamalıdır. Ona görə də `_` , `$` və `__A_V_` adları məsləhət görülən adlar olmasa da qaydalar çərçivəsində elan edilmiş adlardır.

3-cü diqqət çəkən məsələ `for` dövrünün quruluşudur. Standart for dövrü adətən 3 hissədən ibarət olur (mötərizənin içi nəzərdə tutulur):

```
for(initialization; booleanExpression; updateStatement) {  // body  }
```

Və bütün bu hissələr optional`dı, yəni istəyə bağlı olaraq istifadə oluna da bilər, buraxıla da bilər. Aşağıdakı formada yazsaq normal kompayl olunacaq və şərt buraxıldığından sonsuz dövrə girəcək:

```
for( ; ; ) {  // infinite loop  }
```

Bu nümunədə 3-cü hissə (updateStatement) buraxılıb. Dövrün şərtinə görə əgər `x`, `main` metoduna göndərilən parametrlərin sayından kiçik olarsa dövr işləyəcəkdir. Azacıq da açıq söyləmiş olsaq command line`dan classı icra (run) etmək üçün bu sintaksisi istifadə edirik:

```
java ClassAdi parametr1 parametr2 parametr3 ...
```

Bu nümunədə verilmiş command line`a görə `main` metoduna göndərilən parametrlər aşağıdakı massiv şəklini almış olur:

```
String[] __A_V_ = {"-", "A", "."};
```

Bu elementlər `for` dövrü vasitəsilə şərtə uyğun olaraq `$` dəyişəninə concatenation vasitəsilə mənimsədilir və sonda print olunur. `for`dan sonra blok mötərizələr olmadığına görə onun tərkibinə özündən sonrakı ancaq bir statement daxildir, yəni `out.println($);` `for`un gövdəsinə daxil deyil və bir dəfə icra edilir.

İndi `for`un 3 halına baxaq:

1. `for(int x = 0; ++x < __A_V_.length;)`

`++x` - pre-increment operatoru olduğundan əvvəlcə operator icra edilir və qaytarılan yeni dəyər massivin ölçüsü ilə müqayisə edilir:

```
1 < 3 → $ += __A_V_[1]; → A
2 < 3 → $ += __A_V_[2]; → A.
3 < 3 → false
```

Output:

```
A.
```

2. `for(int x = 0; x++ < __A_V_.length;)`

`x++` - post-increment operatoru olduğundan əvvəlcə müqayisə aparılır, sonra operator icra edilir:

```
0 < 3 → $ += __A_V_[1]; → A
1 < 3 → $ += __A_V_[2]; → A.
2 < 3 → $ += __A_V_[3]; → ArrayIndexOutOfBoundsException: 3
```

3. `for(int x = 0; x < __A_V_.length; ++x)`

Bu standart qaydada işlətdiyimiz `for` dövrüdür və *updateStatement* mövcuddur. Burada bir vacib məqamı yadda saxlamaq lazımdır ki, updateStatement həmişə gövdədən sonra icra edilir, ona görə də bu halda `++x` yaxud `x++` olmağı heç bir fərq etmir:

```
0 < 3 → $ += __A_V_[0]; → -
1 < 3 → $ += __A_V_[1]; → -A
2 < 3 → $ += __A_V_[2]; → -A.
3 < 3 → false
```

Output:

```
-A.
```

Nümunə 2:

```
3:  int m = 9, n = 1, k = 0;
4:  while (m > n) {
5:     m--;
6:     n += 2;
7:     k += m + n;
8:  }
9:  System.out.println(k);
```

Həlli: Bu sualdakı əməliyyatları şifahi hesablayıb k-nın cavabını tapmaq nisbətən çətindir. Çünki 2 ya 3 addımdan sonra bütün dəyişənlərin son dəyərini yadda saxlamaq mümkün olmur yaxud səhv yadda saxlama ehtimalı çox böyükdür, düzgün hesabladığına əmin olmaq üçün sualı yenidən gözdən keçirməyə başlayırsan. Və bütün bunlar da müəyən vaxt itkisinə səbəb olur. Nəzərə alsaq ki, imtahanda hər suala ortalama 1.5-2 dəqiqə vaxt düşür, o zaman bu sualların optimal həll variantlarını axtarıb tapmaq lazım gəlir. Bu həll variantlarından biri də cədvəl şəklində həll variantıdır. Dəyişənlərin adlarını sütunlara yazırıq və hər növbəti sətirdə də dəyişənlərin müvafiq əməliyyatlardan sonra dəyişilmiş dəyərini qeyd edirik:

m	n	k
9	1	0
8	3	11
7	5	23
6	7	**36**

Nümunə 3:

```
3:  int count = 0;
4:  ROW_LOOP: for (int row = 1; row <= 3; row++)
5:     for (int col = 1; col <= 2; col++) {
6:        if (row * col % 2 == 0) continue ROW_LOOP;
7:        count++;
8:     }
```

```
9:  System.out.println(count);
```

Həlli:

row	col	count
1	1	1
1	2	1
2	1	1
2	2	1
3	1	2
3	2	**2**

Nümunə 4:

```
public static void main(String[] args) {
   int x = 5;
   while(x >= 0){
      int y = 3;
      while(y > 0){
         if(x < 2) continue;
         x--; y--;
         System.out.println(x*y + " ");
      }
   }
}
```

A. `43 31 20 12`

B. `6 4 2 3`

C. `8 3 0 2`

D. The code will not compile because of line 6.

E. The code will not compile because of line 7.

F. The code contains an infinite loop and does not terminate.

İzahı: Bu suala "Sybex Practise Exams"-in birində rast gələcəksiniz, mənim ən çox bəyəndiyim suallardan biridir. Sualı tapmaq üçün xüsusi bir texnika tələb olunmur. Doğru cavablandırmaq üçün sadəcə güclü diqqət və səbr tələb olunur. Özünüz cavablandırmağa çalışın, sualın sonuna çatdığınıza əmin olmadan, cavabı doğru tapdığınıza əmin olmayın.

Nümunə 5:

```
public class Question44 {
```

```
  String tiger = "Tiger";
  String lion = "Lion";
  static String leopard = "Leopard";

  final String  statement1 = tiger;
  static final String  statement2 = tiger;     // doesn't compile
  static final String  statement3 = leopard;

  public static void main(String[] args) {
    Question44 q = new Question44();
    q.tiger = null;
    System.out.println(q.statement1 + " <> " + q.tiger);   // Tiger <> null

    String tiger = "Tiger";
    String lion = "Lion";
    final String statement = 250>338 ? lion : tiger = " cub"; // doesn't compile
  }
}
```

Chapter 3. Core Java APIs

Creating and Manipulating Strings

`String` dəyişəninə iki formada dəyər mənimsədə bilərik:

```
String name = "Cavid";              // String poolda saxlanılır
String name = new String("Cavid");  // Yeni obyekt yaradılır
```

Concatenation

Yadda saxlanılmalı qaydalar:

1. Əgər hər iki dəyişən (operand) numeric olarsa, `+` toplama əməliyyatını nəzərdə tutur.
2. Əgər dəyişənlərdən biri `String` olarsa, `+` concatenation əməliyyatını nəzərdə tutur.
3. İfadə soldan sağa doğru hesablanır.

```
System.out.println(1 + 2);              // 3
System.out.println("a" + "b");          // ab
System.out.println("a" + "b" + 3);      // ab3
System.out.println(1 + 2 + "c");        // 3c
System.out.println(1 + 2 + "c"+ 1 + 2); // 3c12

int three = 3;
String four = "4";
System.out.println(1 + 2 + three + four);  // 64

String s = "1";
s += "2";
s += 3;
System.out.println(s);   // 123

StringBuilder sb = new StringBuilder("Hello ");
String s = "java!";
String welcome = sb + s;
System.out.println(welcome);   // Hello java!
```

Əgər `+` operatorunun sağ və ya sol tərəfindəki dəyərlərdən biri `String` tipindədirsə, icra vaxtı digər dəyər də String`ə çevrilir. Aşağıdakı nümunəyə baxaq:

```
public static void main(String[] args) {
   System.out.print(null + true);         // line1
   System.out.print(true + null);         // line2
   System.out.print(null + null);         // line3

   System.out.print(getString() + true);  // line4
}
static String getString() {
   return null;
}
```

`line1, line2, line3` – kompayl olunmur, çünki aydındır ki, `true` String tip deyil, `null` isə burada konflikt yaradır, konkret `String` tipə aid etmək olmaz. Amma `line4` kompayl olunur, çünki `null` `String` tipə mənimsədilib və icra edildikdə ekrana `nulltrue` çap edilir.

Immutability

`String` *immutable* dəyişəndir, yəni `String` obyekti bir dəfə yaradıldısa, sonradan dəyişdirilməyinə icazə verilmir, *genişlənə* və *kiçilə* bilməz. Həmçinin Java`da immutable classlar final`dır.

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);    // 12
```

The String Pool

Ümumiyyətlə, `String` dəyişənləri ən çox istifadə edilən dəyişənlərdən biridir və ümumi proqramda yaddaşın təxminən 25-40 faizini mənimsəyir.

String pool (intern pool) – bütün stringləri toplamaq üçün JVM`də ayrılmış yerdir. String pool özündə ancaq literal dəyərləri tutur (contain).

Important String Methods

Java`da indekslər sıfırdan başlayaraq sayılır.

Nümunələr aşağıdakı dəyişən üzərindən aparılacaqdır:

```
String animals = "animals";
```

length()

Quruluşu (signature):

```
int length()
```

```
System.out.println(animals.length());    // 7
```

charAt()

Axtarılan indeksdə olan simvolu (character) geri qaytarır.

Quruluşu (signature):

```
char charAt(int index)
```

```
System.out.println(animals.charAt(0));  // a
System.out.println(animals.charAt(6));  // s
System.out.println(animals.charAt(7));  // throw StringIndexOutOfBoundsException
```

Dokumentasiyaya (javadoc) əsasən isə `IndexOutOfBoundsException` fırladır.

`charAt()` metodu geriyə `char` dəyər qaytardığından onu `String`ə mənimsətmək mümkün deyil:

```
String str = "str";
String s = str.charAt(0);  // incompatible types: char cannot be converted to String
```

indexOf()

Hər hansı bir simvolun (character) və ya ifadənin verilmiş stringin daxilində olub olmadığını axtarır və nəticə müsbət olarsa, axtarılan ifadənin ilk tapıldığı indeksi geri qaytarır. Axtarışa istənilən indeksdən başlamaq olar.

Quruluşu (signature):

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)
```

```
System.out.println(animals.indexOf('a'));            // 0
System.out.println(animals.indexOf("al"));           // 4
System.out.println(animals.indexOf('a', 4));         // 4
System.out.println(animals.indexOf("al", 5));        // -1
System.out.println(animals.indexOf("al", 9));        // -1
System.out.println(animals.indexOf(''));  // doesn't compile, empty character literal
```

`charAt()` metodundan fərqli olaraq `indexOf()` metodu axtarılan dəyəri tapmadıqda exception fırlatmır, **`-1`** dəyərini geri qaytarır.

substring()

Quruluşu (signature):

```
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)

System.out.println(animals.substring(3));                    //mals
System.out.println(animals.substring(animals.indexOf('m')));  //mals
System.out.println(animals.substring(3, 4));                 //m
System.out.println(animals.substring(3, 7));                 //mals

System.out.println(animals.substring(3, 3)); //empty string
System.out.println(animals.substring(3, 2)); //throws StringIndexOutOfBoundsException
System.out.println(animals.substring(3, 8)); //throws StringIndexOutOfBoundsException
System.out.println(animals.substring(7));    //empty string
```

`beginIndex` qayıdan nəticəyə daxildir, amma `endIndex` daxil deyil. Sonuncu nümunəyə diqqət edək, 7-ci indeks mövcud olmamasına baxmayaraq exception baş vermədi. Əgər bu nümunədə `(length-beginIndex)` mənfi olmazsa, exception baş vermir, sıfıra bərabər olarsa, boş string qaytarır.

toLowerCase() and toUpperCase()

Quruluşu (signature):

```
String toLowerCase()
String toUpperCase()

System.out.println(animals.toUpperCase());        // ANIMALS
System.out.println("Abc123".toLowerCase());       // abc123
```

equals() and equalsIgnoreCase()

Quruluşu (signature):

```
boolean equals(Object str)
boolean equalsIgnoreCase(String str)

System.out.println("abc".equals("ABC"));             // false
System.out.println("ABC".equals("ABC"));             // true
System.out.println("abc".equalsIgnoreCase("ABC"));   // true

Object o1 = "str";
Object o2 = "Str";
System.out.println(o1.equals(o2));    // false
System.out.println(o1.equalsIgnoreCase(o2)); //DOES NOT COMPILE, only String..
```

```
System.out.println(((String)o1).equalsIgnoreCase((String)o2));  // true
```

startsWith() and endsWith()

Quruluşu (signature):

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)

System.out.println("abc".startsWith("a"));     // true
System.out.println("abc".startsWith("A"));     // false
System.out.println("abc".endsWith("c"));       // true
System.out.println("abc".endsWith("a"));       // false
```

contains()

Quruluşu (signature):

```
boolean contains(CharSequence s)

System.out.println("abc".contains("b"));        // true
System.out.println("abc".contains("B"));        // false
```

Bu metod çıxdıqdan sonra artıq `indexOf()` metodu ilə şərt yoxlamağa ehtiyac qalmadı.

replace()

Quruluşu (signature):

```
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)

System.out.println("abcabc".replace('a', 'A'));      // AbcAbc
System.out.println("abcabc".replace("a", "A"));      // AbcAbc
System.out.println("abcabc".replace("ab", "AB"));    // ABcABc
System.out.println("abcabc".replace("d", "D"));      // abcabc, no exception

StringBuilder sb = new StringBuilder("bc");
System.out.println("abcabc".replace(sb, "BC"));      // aBCaBC
```

2 faktı yadda saxlamaq lazımdır:

1. `replace()` metodu yeni `String` obyekti yaradır;
2. əgər `replace()` metoduna göndərilən hər iki parametr eynidirsə, yəni heç bir dəyişiklik baş verməyəcəksə, eyni `String` obyekti geri qaytarılır.

```
"String".replace('g','g') == "String"                   // true
"String".replace('g','g') == "String".replace('S','S')  // true
"String".replace('S','s') == "String".replace('S','s')  // false
"String".replace('g','G') == "StrinG"                   // false
```

trim()

String`in əvvəlindəki və sonundakı boşluqları, həmçinin `\t` (tab), `\n` (newline) və `\r` (carriage return) simvollarını da silir. Amma ortadakı boşluqlara toxunmur.

Quruluşu (signature):

```
String trim()
```

```
System.out.println("abc".trim());              // abc
System.out.println("abc\r !".trim());          //  !
System.out.println("\t   a b c \n".trim());    // a b c
```

intern()

Quruluşu (signature):

```
public String intern()
```

Bu metod böyük ehtimalla imtahan suallarına daxil edilmir, hazırlaşdığım sertifikat kitabında da bu metodla bağlı hər hansı bir izah yox idi. Amma Enthuware testlərində bu metodla bağlı sual ilə rastlaşa bilərsiniz. İlkin olaraq qeyd edək ki, String Pool`da saxlanılan bütün dəyərlər *intern* özəlliyinə sahibdir. Bu nə deməkdir? Tutaq ki, bizim `String` tipində bir listimiz var, özündə ad və soyadları saxlayır. Adlar və ya soyadlar bir neçə dəfə təkrarlana bilər, amma intern özəlliyi sayəsində təkrarlanan ad və ya soyadlar pool`da cəmi bircə dəfə saxlanılır. Yəni String Pool`a əlavə edilən bütün dəyərlər avtomatik olaraq intern edilir və təkrarlanan dəyərlər olarsa, bu təkrarlanma aradan qaldırılaraq yaddaş sahəsinə qənaət edilir. Bununla da eyni dəyərə malik fərqli referanslar eyni yaddaş sahəsinə müraciət edir. Əgər bizim `String name1 = "Mushfiq";` adlı dəyərimiz varsa, `String name2 = "Mushfiq";` dəyərini yaratdıqda `"Mushfiq"` dəyəri String Pool`da yenidən yaradılmayacaq, artıq mövcud olduğundan `name1` və `name2` hər ikisi eyni yaddaş sahəsinə müraciət edəcək. Bəs `intern()` metodu hər hansı `String` dəyəri və ya referansı üzərindən çağırıldıqda nə baş verir? Əgər həmin `String` obyekti pool`da mövcuddursa, həmin obyekt geri qaytarılır, yox əgər mövcud deyilsə, həmin obyekt pool`a əlavə edilir və onun referansı geri qaytarılır. `intern()` metodu ilə çağırılmış 2 String`in müqayisəsinin `true` qaytarması üçün mütləq onların dəyərləri bir-birinə bərabər olmalıdır, yəni `s1.intern() == s2.intern()` o zaman `true` olar ki, `s1.equals(s2)` `true` olsun. Nümunələr üzərindən davam edək:

```
3.  String s1 = new String("java");
4.  String s2 = "java";
5.  String s3 = s1.intern();   //returns string from pool, now it will be same as s2
6.  System.out.println(s1 == s2);  // false
7.  System.out.println(s2 == s3);  // true
```

`new` açar sözü ilə yaradılan `String` dəyərlər bildiyimiz kimi String Pool`da saxlanılmır, ona görə də `s1` və `s2` dəyişənlərinin dəyərləri eyni olsa da onlar fərqli yaddaş sahələrində saxlanılan obyektlərə müraciət edirlər. Bu səbəbdən onların referanslarını müqayisə etdikdə geriyə `false` dəyər qaytarır. `s3` dəyişəni ilə isə yuxarıda qeyd etdiyimiz qayda baş verir. `s1` dəyişəni üzərindən `intern()` metodu çağırıldığına görə ilk öncə `s1` dəyərinin String Poolda olub olmaması yoxlanılır. `s1` dəyəri artıq poolda mövcuddur və bu dəyərə eyni zamanda `s2` referansı da müraciət edir. Və yekun nəticə odur ki, `s2` və `s3` pool`da mövcud olan eyni obyektə, yəni eyni yaddaş sahəsinə müraciət edirlər. Bu səbəbdən onların referanslarını müqayisə etdikdə geriyə `true` dəyər qaytarır.

Method Chaining

Eyni bir `String` dəyişəni üçün bir neçə metodu zəncirvari şəkildə bir sətirdə çağırmaq mümkündür:

```
String result = "AniMaL    ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);   // AnimAl
```

Yuxarıda birinci sətirdə 4 obyekt yaradılır.

```
String a = "abc";
String b = a.toUpperCase();
b = b.replace("B", "2").replace('C', '3');
System.out.println("a="+a);     // abc
System.out.println("b="+b);     // A23
```

StringBuilder Class

Nümunə 1:

```
String s = "";
for(char current='a'; current <= 'z'; current++)
   s += current;
System.out.println(s);
```

Nümunə 2:

```
StringBuilder sb = new StringBuilder();
for(char current='a'; current <= 'z'; current++)
   sb.append(current);
System.out.println(sb);
```

Nümunə 1 və Nümunə 2 mahiyyət baxımından ikisi də demək olar ki, eyni işi görür. Amma fərq ondadır ki, `String` dəyişəni immutable dəyişəndir və ona görə də 1-ci nümunədə hər dəfə concatenation əməliyyatı yerinə yetirildikdə yeni dəyər yaradılır və köhnə dəyər artıq garbage

collection üçün "eligible" obyekt hesab olunur (təxminən 27 obyekt yaradılır və əksəri eligible olur).

`StringBuilder` mutable`dir və ona görə də ikinci nümunədə cəmi bir obyekt yaradılır, hər `append()` metodundan sonra sadəcə dəyəri dəyişir.

`String` dəyərini `StringBuilder`ə mənimsətmək olmaz:

```
StringBuilder sb = "StringBuilder";       // DOES NOT COMPILE
```

Mutability and Chaining

Biz `String` metodlarını zəncirvari şəkildə çağırdıqda (chaining) nəticə olaraq geriyə **yeni String** qaytarır. Amma `StringBuilder` chaining olunan referansın öz dəyərini dəyişir və geriyə həmin referansı qaytarır, yəni yeni `StringBuilder` obyekti yaradılmır:

```
StringBuilder a = new StringBuilder("abc");
StringBuilder b = a.append("de");
b = b.append("f").append('g');
System.out.println("a = "+a);   // abcdefg
System.out.println("b = "+b);   // abcdefg
```

Burada cəmi bir `StringBuilder` obyekti var, `a` və `b` referansları hər ikisi eyni obyektə işarə edirlər (refer).

Creating a StringBuilder

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
StringBuilder sb4 = new StringBuilder(-5);   // throws NegativeArraySizeException
```

`sb3` referansında `10` capacity`ni göstərir, length`ı deyil. *Capacity* obyektin özündə nə qədər simvol tutub saxlaya biləcəyini göstərir. `String` immutable olduğundan onun üçün *length* və *capacity* eynidir. Amma `StringBuilder`ə yeni simvollar əlavə olunduqca capacity avtomatik olaraq genişlənir. `StringBuilder` yaradılarkən capacity təyin edilməyibsə, default olaraq 16`dan başlayır.

```
System.out.println(sb3.length());     // output: 0
System.out.println(sb3.capacity());   // output: 10
```

Capacity`ni artırmaq üçün `ensureCapacity(int minimumCapacity)` metodundan istifadə edilir. Göndərilən parametrdən asılı olaraq fərqli nəticələr meydana çıxır:

1) əgər `minimumCapacity` mənfi olarsa, heç bir dəyişiklik baş vermir və `currentCapacity` yerində qalır *(default olaraq 16)*;

2) əgər `currentCapacity > minimumCapacity` olarsa, heç bir dəyişiklik baş vermir və `currentCapacity` yerində qalır;

3) əgər `currentCapacity < minimumCapacity` olarsa, yeni capacity yaradılır:
   a) `minimumCapacity < currentCapacity*2+2` olarsa, yeni capacity `currentCapacity*2+2` olacaq;
   b) `minimumCapacity > currentCapacity*2+2` olarsa, yeni capacity `minimumCapacity` olacaq.

```
StringBuilder sb = new StringBuilder();
System.out.println(sb.capacity());  // 16
sb.ensureCapacity(-10);             // 16
sb.ensureCapacity(10);              // 16
sb.ensureCapacity(20);              // 34
sb.ensureCapacity(71);              // 71
```

Important StringBuilder Methods

charAt(), indexOf(), length() və substring()

Bu 4 metod `String`in eyniadlı metodları ilə demək olar ki, eyni işi görür.

```
StringBuilder sb = new StringBuilder("animals");
String sub = sb.substring(sb.indexOf("a"), sb.indexOf("al"));
int len = sb.length();
char ch = sb.charAt(6);
System.out.println(sub+" "+len+" "+ch);   // anim 7 s
```

`substring()` metodu geriyə `StringBuilder` yox `String` qaytarır, ona görə də `sb` referansının dəyəri **dəyişmir.**

```
StringBuilder sb = new StringBuilder("animals");
sb.substring(sb.indexOf("a"), sb.indexOf("al"));
sb.append("-dog");
System.out.println(sb);   // output is animals-dog, not anim-dog

sb.append("a").substring(0, 4).insert(2, "asdf");  // DOES NOT COMPILE
```

Sonuncu sətir kompayl olunmur, çünki `substring()` metodu geriyə `String` qaytarır və `String` classının da `insert()` metodu yoxdur.

append()

`StringBuilder`ə yeni parametrlər əlavə edir və geriyə həmin obyektin referansını qaytarır. `append()` metodu 10`dan çox tipdə parametr qəbul edir. Əsas `String` tipli parametrlə olan nümunələrə baxılır.

Quruluşu (signature):

```
StringBuilder append(String str)
```

```
StringBuilder sb = new StringBuilder(50).append(1).append('c');
sb.append("-").append(true);
System.out.println(sb);       // output: 1c-true
```

İmtahanda mənə düşən və SYBEX kitabında olmayan yeganə sual 3 parametrli append() metodu ilə bağlı idi.

Quruluşu (signature):

```
public StringBuilder append(CharSequence s, int start, int end)
```

```
String s = new StringBuilder("sb").append("0123456789", 4, 8).toString();
System.out.println(s);    // sb4567
new StringBuilder("sb").append("0123456789",4,11); // throws IndexOutOfBoundsException
```

insert()

`StringBuilder`də qeyd olunan indeksə yeni parametrlər əlavə edir və geriyə həmin obyektin referansını qaytarır.

Quruluşu (signature):

```
StringBuilder insert(int offset, String str)
StringBuilder insert(int dstOffset, CharSequence s, int start, int end)
```

```
StringBuilder sb = new StringBuilder("animals");
sb.insert(7, "-");          // animals-
sb.insert(0, "-");          // -animals-
sb.insert(4, "-");          // -ani-mals-
System.out.println(sb);
```

```
String s2 = new StringBuilder("sb").insert(0, "0123456789", 4, 8).toString();
System.out.println(s2);    // 4567sb
new StringBuilder("sb").insert(0, "0123456789", 4, 11);  // throws exception
```

delete() and deleteCharAt()

`delete()` metodu `insert()` metodunun əksidir. Ardıcıllıqdan (sequence) simvolları silir və geriyə həmin obyektin referansını qaytarır. Əgər ancaq bir simvol silmək istəyiriksə, o zaman `deleteCharAt()` metodu daha uyğundur.

Quruluşu (signature):

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

Nümunə:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3);          // adef
sb.deleteCharAt(5);       // throws StringIndexOutOfBoundsException
sb.delete(1);             // DOES NOT COMPILE
sb.delete(1, 45);         // a
```

`delete()` metodu aşağıdakı hallarda `StringIndexOutOfBoundsException` verir:

- `start` parametri mənfi dəyər olarsa;
- `start String`in ölçüsündən (`length`) böyük olarsa;
- `start` parametri `end` parametrindən böyük olarsa.

`deleteCharAt()` metodu aşağıdakı hallarda `StringIndexOutOfBoundsException` verir:

- `index` mənfi dəyər alarsa;
- `index` String`in ölçüsünə (length) bərabər və ya ondan böyük olarsa.

reverse()

Ardıcıllıqdakı (sequence) simvolları tərsinə çevirir və geriyə həmin obyektin referansını qaytarır.

Quruluşu (signature):

```
StringBuilder reverse()
```

Nümunə:

```
StringBuilder sb = new StringBuilder("Ismayil");
sb.reverse();
System.out.println(sb);     // liyamsI
```

toString()

`StringBuilder`i `String`ə çevirir.

Quruluşu (signature):

```
String toString()
```

Nümunə:

```
String s = sb.toString();
```

setLength()

`StringBuilder`ə yeni uzunluq vermək üçün istifadə edilir.

Quruluşu (signature):

```
public void setLength(int newLength)
```

Əgər `newLength` cari uzunluqdan kiçik olarsa, o zaman `newLength` `StringBuilder`in yeni uzunluğu olur və geriyə qalan digər simvollar atılır. Yox əgər `newLength` cari uzunluqdan böyük və ya bərabər olarsa, `newLength` yenə də yeni uzunluq olur və çatışmayan simvolların yeri `'\u0000'` simvolu ilə doldurulur. `setLength()` metodu mənfi parametr qəbul edə bilməz.

Nümunə:

```
StringBuilder sb = new StringBuilder("12345678");
System.out.println(sb + ", " + sb.length());   // 12345678, 8
sb.setLength(5);
System.out.println(sb + ", " + sb.length());   // 12345, 5
sb.setLength(10);
System.out.println(sb + ", " + sb.length());   // 12345     , 10
sb.setLength(-5);   // throws StringIndexOutOfBoundsException
```

StringBuilder vs. StringBuffer

`StringBuilder` `StringBuffer`ə nisbətən java`ya sonradan əlavə olunub. Demək olar ki, eyni işi görürlər, amma `StringBuffer` "thread safe" olduğundan daha yavaş işləyir.

`StringBuffer` imtahan mövzusuna daxil deyil, sadəcə yadınızda saxlaya bilərsiniz ki, `StringBuilder` kimi eyni metodlara malikdir. `String`, `StringBuilder` və `StringBuffer` hər üçü `final` classdır. Ümumiyyətlə, final classlar ilə bağlı əlavə olaraq aşağıdakıları bilməyinizdə fayda var:

1. Wrapper classlar (`Boolean`, `Integer`, `Long`, `Short`, `Byte`, `Character`, `Float`, `Double`) `final` classlardır və onlardan varis almaq mümkün deyil;
2. `Number` classı `final` class deyil və `Integer`, `Long`, `Double` və s. wrapper classlar `Number` classının varisidirlər ("extends" edirlər);
3. `java.lang.System` classı da həmçinin `final` classdır.

Understanding Equality

== operatoru ilə iki referansı müqayisə etdikdə hər iki referansın da eyni obyektə işarə edib etmədiyini müqayisə edir. `equals()` metodu ilə müqayisə etdikdə isə referansların dəyərlərini bir-biri ilə müqayisə edir. `String` classı `equals()` metodunu override edir deyə eyni dəyərə malik iki Stringi `equals()` ilə müqayisə etdikdə nəticə `true` verir. Ancaq `StringBuilder` classı `equals()` metodunu override etmir, ona görə də eyni dəyərə malik `StringBuilder` referanslarını `equals()` ilə müqayisə etdikdə nəticə `false` verir. Ümumiyyətlə, əgər hər hansı bir class `equals()` metodunu override etmirsə, həmin classın referansı üzərindən `equals()` metodu çağırılsa, bu zaman `Object` classının `equals()` metodu çağırılır. Çünki bütün classlar aşkar şəkildə qeyd olunmasa da `Object` classından törəmədirlər və onun metodlarını varis alırlar. Bu mövzu Chapter 5'də ayrıca izah ediləcəkdir, hələlik isə bilməniz yetərlidir ki, `Object` classının `equals()` metodu dəyərləri müqayisə etmir, referansları müqayisə edir. Əgər referanslar eyni obyektə işarə edirsə, `true` qaytarır, digər hallarda isə `false` qaytarır.

```
public class Book {
   public static void main(String[] args) {
      Book b1 = new Book ();
      Book b2 = new Book ();
      Book b3 = b1;
      System.out.println(b1 == b3);       // true
      System.out.println(b1 == b2);       // false
      System.out.println(b1.equals(b2));  // false
   }
}
```

`Book` classı göründüyü kimi `equals()` metodunu override etmir, belə olan halda bu class üçün == və `equals()` eyni cür işləyir və eyni nəticəni verir.

İndi isə `String` və `StringBuilder` ilə bağlı nümunələrə baxaq.

`StringBuilder` ilə bağlı nümunə:

```
StringBuilder one = new StringBuilder("java");
StringBuilder two = new StringBuilder("java");
System.out.println(one == two);                      // false
System.out.println(one.equals(two));                 // false

StringBuilder three = one.append(two);

System.out.println(one == three);                    // true
System.out.println(one.equals(three));               // true
System.out.printf("%s, %s, %s%n", one, two, three);  // javajava, java, javajava
```

`String` ilə bağlı nümunə:

```
String x = "Hello World";
String y = "Hello World";
```

```
System.out.println(x == y);              // true
System.out.println(x.equals(y));         // true

String z = " Hello World".trim();
System.out.println(x == z);              // false
System.out.println(x.equals(z));         // true
```

`String` immutable`dır və String literal dəyərləri String Pool'da saxlanılır. JVM yaddaşda eyni literaldan ancaq bir ədəd yaradır. Yuxarıdakı nümunədə `"Hello World"` bir dəfə yaradılır və pool`da saxlanılır. `x` və `y` referanslarının hər ikisi də həmin o obyektə işarə edir.

`" Hello World".trim();` literal dəyər olmadığından pool`da saxlanılmır, ona görə də `x` və `z` eyni obyektə işarə etmirlər, amma eyni dəyərə malikdirlər. Burada da incə bir məqam var, aşağıdakı kod nümunəsinə baxaq:

```
String x = "Hello World";
String y = "Hello World".trim();
String z = "Hello World".toString();
System.out.println(x == y);          // true
System.out.println(x == z);          // true
```

Bu mövzu ətraflı şəkildə coderanch forumunda müzakirə edilib:

http://www.coderanch.com/t/652234/ocajp/certification/Understanding-Equality-page-Java-OCA

`String` metodlar üzərindən bəzi əlavə nümunələrə baxaq:

```
String s1 = "test";
String s2 = s1.substring(0);
String s3 = s1.replace("e","e");
String s3_ = s1.replace('e','e');
String s4 = "test".trim();
String s5 = "test ".trim();
String s6 = s1.toString();
System.out.printf("substring: %s==%s returns %b%n", s1, s2, s1==s2);
System.out.printf("replace1:  %s==%s returns %b%n", s1, s3, s1==s3);
System.out.printf("replace2:  %s==%s returns %b%n", s1, s3_, s1==s3_);
System.out.printf("trim1:     %s==%s returns %b%n", s1, s4, s1==s4);
System.out.printf("trim2:     %s==%s returns %b%n", s1, s5, s1==s5);
System.out.printf("toString:  %s==%s returns %b%n", s1, s6, s1==s6);
```

Output:

```
substring:  test==test returns true
replace1:   test==test returns false
replace2:   test==test returns true
```

```
trim1:      test==test returns true
trim2:      test==test returns false
toString:   test==test returns true
```

P.S. replace() char parametr ilə olanda true qaytarır.

Primitiv wrapper classların (`Integer`, `Double`, `Float` və s.) `equals()` metodu ilə istifadəsi zamanı aşağıdakı 3 qayda keçərlidir:

1. *symmetric* => `a.equals(b)` ilə `b.equals(a)` eyni nəticəni qaytarır;
2. *transitive* => əgər `a.equals(b)` və `b.equals(c)` geriyə `true` qaytarırsa, o zaman `a.equals(c)` də `true` qaytaracaq;
3. *reflexive* => `a.equals(a)` geriyə `true` qaytarır.

Aşağıdakı suala Enthuware test bankında rast gələcəksiniz. İmtahanda əvvəllər "drag-drop" tipli suallar olsa da hazırda bu formatda suallar imtahan testlərindən çıxarılıb. Amma bu sualın içərisində maraqlı məqamlar var, faydalı olacağından qeyd etməyi zəruri bildim.

```
Integer i1 = 1;
Integer i2 = new Integer(1);
int i3 = 1;
Byte b1 = 1;
Long g1 = 1L;
```

1. i1 == i2	false
2. i1 == i3	true
3. i1 == b1	Will not compile
4. i1.equals(i2)	true
5. i1.equals(g1)	false
6. i1.equals(b1)	false

`i1 == i2` – geriyə `false` qaytarır, çünki referanslar fərqli obyektlərə işarə edir.

`i1 == i3` – geriyə `true` qaytarır, çünki `i3` primitiv tipdə olduğundan `i1` də "unboxed" edilərək primitiv tipə çevirilir və sonra müqayisə aparılır.

`i1 == b1` – kompayl olunmur, çünki `i1` və `b1` referansları eyni ierarxiyada olmayan classlara işarə edirlər və kompilyator kompayl zamanı bunu tutur və başa düşür ki, bu referanslar eyni obyektə işarə edə bilməzlər.

`i1.equals(i2)` – geriyə `true` qaytarır, çünki hər ikisi `Integer` obyektidir və dəyərləri də eynidir.

`i1.equals(b1)` və `i1.equals(g1)` - geriyə `false` qaytarır, çünki müqayisə olunan dəyərlər fərqli classlara aiddir. `==` ilə müqayisədən fərqli olaraq `equals()` ilə müxtəlif classlara işarə edən referansların müqayisəsi zamanı kompayl xətası baş vermir, çünki `equals()` metodu parametr olaraq `Object` tipi qəbul edir. Bütün wrapper classların `equals()` metodunun gövdəsində ilk olaraq müqayisə edilən obyektlərin eyni classa aid olub olmadığı yoxlanılır. Əgər eyni classa aid deyilsə, o zaman dəyərlərin yoxlanılması aparılmır, birbaşa false qaytarılır. `Integer` metodunun `equals()` metoduna baxsaq bu proses aydın olacaq:

```
public boolean equals(Object obj) {
   if (obj instanceof Integer) {
      return value == ((Integer)obj).intValue();
   }
   return false;
}
```

Wrapper classlar ilə bağlı digər maraqlı bir nümunəyə Coderanch forumunda rast gələ bilərsiniz:

```
public static void main(String[] args) {
     int myInt1 = 10;
     Integer myInteger1 = 10;
     Integer myInteger2 = 10;
     Integer myInteger3 = new Integer(10);
     Short myShort1 = 10;
     Number myNumber1 = 10;
     Double myDouble1 = 10.0;
     Double myDouble2 = 10;    // DOES NOT COMPILE

     System.out.println(myInt1 == myInteger1);   // true
     System.out.println(myInt1 == myInteger3);   // true
     System.out.println(myInt1 == myShort1);     // true
     System.out.println(myInt1 == myNumber1);    // DOES NOT COMPILE
     System.out.println(myInt1 == myDouble1);    // true

     System.out.println(myInteger1 == myInteger2);   // true
     System.out.println(myInteger1 == myInteger3);   // false
     System.out.println(myInteger1 == myShort1);     // DOES NOT COMPILE
     System.out.println(myInteger1 == myNumber1);    // true
     System.out.println(myInteger1 == myDouble1);    // DOES NOT COMPILE

     System.out.println(myShort1 == 10.0);           // true
     System.out.println(myShort1 == myNumber1);      // false
     System.out.println(myShort1 == myDouble1);      // DOES NOT COMPILE
}
```

Ətraflı:

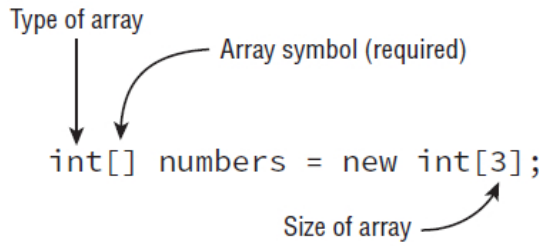
http://www.coderanch.com/t/643974/ocajp/certification/operator-wrapper-classes

Əlavə olaraq aşağıdakı nümunələrə baxmağınızda da fayda var:

```
String s1 = new String();
String s2 = new String();
System.out.println(s1.equals(s2));   //true, çünki equals metodunu implement edir

StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder();
System.out.println(sb1.equals(sb2)); //false, çünki equals metodunu implement etmir

ArrayList l1 = new ArrayList();
ArrayList l2 = new ArrayList();
System.out.println(l1.equals(l2));   //true, çünki equals metodunu implement edir

Object o1 = new Object();
Object o2 = new Object();
System.out.println(o1.equals(o2));      // false

String str1 = "one";
String str2 = "two";
System.out.println(str1.equals(str1 = str2));   // false
System.out.println(str1.equals(str2 = str1));   // true

String s1 = "s1";
String s = "s";
System.out.println(s1 == s + 1);    // false
System.out.println(s1 == "s" + 1);  // true
final String e1 = "e1";
final String e = "e";
System.out.println(e1 == e + 1);   // true

String s1 = "java";
StringBuilder s2 = new StringBuilder("java");
if(s1 == s2)                    // DOES NOT COMPILE
   System.out.println("1");
if(s1.equals(s2))               // false
   System.out.println("2");

String a = "";
String b = "2cfalse";
a += 2;
a += 'c';
a += false;
System.out.println(a == "2cfalse");         // false
System.out.println(a.equals("2cfalse"));    // true
System.out.println(b == "2cfalse");         // true
```

Understanding Java Arrays

Şəkil 3.1 Massivin (Array) quruluşu



Burada `numbers` referans dəyişəndir, primitiv tip deyil. Bu formada massiv yaratdıqda, massivin elementlərinə müvafiq tipin default dəyərləri mənimsədilir, yəni:

```
numbers[0] = 0;
numbers[1] = 0;
numbers[2] = 0;
```

Dəyərləri birbaşa həmin sətirdə də mənimsətmək mümkündür:

```
int[] numbers = new int[] {11, 22, 33};
```

və yaxud

```
int[] numbers = {11, 22, 33};
```

İkinci yanaşma anonim massiv (anonymous array) adlanır, çünki nə tip bildirilir, nə də ölçü. Əgər sol tərəfdə tip qeyd olunubsa, sağ tərəfdə yenidən qeyd etməyə ehtiyac yoxdur (redundant), java artıq tipi bilir. Amma burada incə bir məqam var, ona xüsusi diqqət yetirmək lazımdır. Qeyd etdik ki, anonim massivdə tipi qeyd etməyə ehtiyac yoxdur, amma massivi elan edib dəyəri növbəti sətirdə mənimsətsək, o zaman kod xəta verəcəkdir. Bu zaman sağ tərəfdə massivin tipi mütləq qeyd olunmalıdır:

```
int[] numbers;
numbers = {11, 22, 33};            // does not compile
numbers = new int[3]{11, 22, 33};  // does not compile
numbers = new int[]{11, 22, 33};
```

Bundan əlavə, əgər massivə elan olunduğu sətirdə dəyər mənimsədilməyibsə, instansı yaradılarkən mütləq ölçüsü qeyd edilməlidir, əks halda kompayl xətası verəcəkdir. Massivin ölçüsü mənfi ola bilməz:

```
int[] a = new int[];    // does not compile
int[] b = new int[-5];  // throws NegativeArraySizeException
```

Düzgün qaydada elan olunmuş massiv formaları:

```
int[] arr1;
int arr2[];
```

```
int[] names1, types1;   // both array
int names2[], types2;   // names2 is array, types2 is int
```

İstənilən java tipini massivin tipi olaraq seçmək mümkündür.

```
String[] games = {"football", "volleyball", "handball"};
String[] sports = games;
System.out.println(games.equals(sports));     // true
System.out.println(games.toString());         // [Ljava.lang.String;@19e0bfd
System.out.println(Arrays.toString(games));   // [football, volleyball, handball]
```

equals() metodu massivin elementlərinin dəyərinə baxmır, referansların eyni obyektə işarə edib etmədiyini yoxlayır.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
int[] arr3 = arr1;
System.out.println(arr1.equals(arr2));    // false
System.out.println(arr1 == arr2);         // false
System.out.println(arr1.equals(arr3));    // true
System.out.println(arr1 == arr3);         // true
```

Böyük tipdən kiçik tipə cast edərkən ehtiyatlı olmaq lazımdır:

```
String[] strings = {"stringValue"};
Object[] objects = strings;
String[] againStrings = (String[]) objects;
againStrings[0] = new StringBuilder();  // does not compile
objects[0] = new StringBuilder();  // does compile, but throw ArrayStoreException
```

Sub class tipində olan massivi super class tipində olan massivə mənimsətmək olar, qısaca polimorfizm, casting qaydaları burada da keçərlidir. Bənzər nümunə ilə Enthuware test bankında rastlaşacaqsınız:

```
public class NewClass {
   public static void main(String[] args) {
      A[] a1, a2;
      B[] b;
      a1 = new A[5];
      b = new B[5];
      a2 = a1;
      a1 = b;       // line1
      b =(B[]) a1;  // line2
      b =(B[]) a2;  // line3
   }
}

class A { }
class B extends A { }
```

Polimorfizm özəlliyinə görə line1 kompayl olunur, kiçik (sub) tipi böyük (super) tipə mənimsədəndə aşkar cast tələb edilmir. Amma line2 və line3'də line1'in tərsi baş verdiyindən aşkar cast tələb edilir, əks halda kompayl xətası verəcəkdir. Ümumiyyətlə, kodda kompayl xətası yoxdur, amma icra etdikdə line3 `ClassCastException` fırladacaq. Line2 isə exception fırlatmır, çünki `a1` referansı `B[]` obyektinə işarə edir. Əgər line1 kommentə salınsa, o zaman line2 də exception fırladacaq. Bu mövzular barədə növbəti fəsillərdə geniş məlumat veriləcək.

Bəzi oxucular bu mövzunu qarışdırır, ona görə də qeyd edək ki, massivlər ayrılıqda bir obyekt hesab edilir. Əgər siz `int` tipində bir massiv yaradırsınızsa, `int` primitiv tip olmasına baxmayaraq yekunda yaradılan massivdir və o bir obyektdir. Ona görə də `int` tipində olan bir massivi `Object` tipində olan bir referansa mənimsədə bilərik. Amma `int` primitiv tip olduğundan `int` tipində olan massivi `Object` tipində olan massivə mənimsədə bilmərik, nümunədən daha aydın olacaq:

```
Object obj = new int[]{-1, 0, 1};      // is valid
Object[] intArr = new int[7];          // is not valid
Object[] integerArr = new Integer[7];  // is valid
```

Son olaraq bir məqamı da qeyd edək. Hamımız bilirik ki, bir java proqramının çalışması üçün istifadə olunan əsas metod `public static void main(String[] args)` metodudur və bu metod `String` massiv tipində parametr qəbul edir. Sadəcə bilməniz gərəkdir ki, əgər `main` metoda parametr ötürülməyibsə, `args` referansının dəyəri ölçüsü sıfır olan bir massiv olur, yəni `null` olmur.

Aşağıdakı nümunə ilə də sizi çaşdıra bilərlər, diqqətli olun. Kodun qaydasında olub-olmadığını isə özünüz təxmin etməyə çalışın:

```
public static void main(String[] a) {
    Integer arr[] = {1, 2, 3, 4};
    arr[1] = null;
    for(Integer a: arr) System.out.println(a);
}
```

Sorting Arrays

Massivləri sort etmək üçün `Arrays.sort()` metodundan istifadə edilir. `Arrays` java tərəfindən təmin olunmuş birinci classdır ki, import tələb edir. Əgər import edilməzsə, hər dəfə tam formada – `java.util.Arrays` – çağırmaqla da istifadə etmək olar. Əgər kod nümunələri birinci sətirdən (line 1) başlamazsa, onda güman etməlisiniz ki, lazımı importlar artıq var.

```
int[] num = { 6, 9, 1 };
Arrays.sort(num);
for(int i = 0; i<num.length; i++)
    System.out.print(num[i] + " ");      // 1 6 9
```

```
String[] string = { "10", "9", "100" };
Arrays.sort(string);
for(String s: string)
    System.out.printf("%s ",s);          // 10 100 9
```

`String` əlifba sırası ilə sort edilir və `1 9`dan öncə gəldiyi üçün `100 9`dan öncə gəlib. `String` ilə sıralamada rəqəmlər hərflərdən və böyük hərflər (uppercase) kiçik hərflərdən (lowercase) öncə gəlir.

Searching

Massivdə axtarış apara bilmək üçün ilk öncə həmin massiv sort olunmalıdır. Axtarışla bağlı 3 hal mövcuddur:

1. Axtarılan element sort olunan massivdə tapılarsa: *elementin tapıldığı uyğun indeksi geri qaytarır;*

2. Axtarılan element sort olunan massivdə tapılmazsa: *axtarılan element əgər massivdə olsaydı hansı indeksdə olardı, onu təyin edir, mənfi ədədə çevirir və sonra 1 çıxır;*

3. Sort olunmamış massivdirsə: *nəticəni əvvəlcədən təxmin etmək mümkün deyil.*

```
int[] nums = {2,4,6,8};
System.out.println(Arrays.binarySearch(nums, 2));   // 0
System.out.println(Arrays.binarySearch(nums, 4));   // 1
System.out.println(Arrays.binarySearch(nums, 1));   // -1
System.out.println(Arrays.binarySearch(nums, 3));   // -2
System.out.println(Arrays.binarySearch(nums, 9));   // -5

int[] num2 = new int[] {3,2,1};                     // unsort
System.out.println(Arrays.binarySearch(num2, 2));   // 1
System.out.println(Arrays.binarySearch(num2, 3));   // -4
```

Creating a Multidimensional Array

```
int[][] arr1;                    // 2D array
int arr2[][];                    // 2D array
int[] arr3[];                    // 2D array
int[] arr4[], arr5[][];          // 2D array and 3D array
String[][] arr6 = new String[3][2];  // 3 arrays with 2 elements, [0][0]-[2][1]
```

Öncəki mövzuda qeyd etmişdik ki, təkölçülü massiv yaradılarkən ölçüsü mütləq qeyd edilməlidir. Amma çoxölçülü massivlərdə sonuncu ölçü buraxıla bilər, məsələn biz massivi aşağıdakı şəkildə yarada bilərik:

```
int[][] a = new int[2][];
```

Bu massivin əgər ikinci ölçüsü qeyd edilmirsə `null` hesab edilir. Yəni bu ikiölçülü massiv özü iki massivdən (`a[0]` və `a[1]`) ibarətdir və hər iki massiv `null`dur:

```
int[] a[] = new int[2][];
System.out.println(a[0]);     // output: null
System.out.println(a[0][1]);  // throws NullPointerException
```

Amma `null` olan massivlərə təkrar yeni dəyər mənimsədilə bilər:

```
int[][] arr = new int[4][];
arr[0] = new int[5];
arr[1] = new int[3];
arr[2] = new int[2];
arr[3] = new int[0];
arr[4] = new int[1];      // throws ArrayIndexOutOfBoundsException
arr[5] = new int[];       // does not compile
arr[1] = new Integer[2];  // does not compile
```

Biz yuxarıda ölçünün qeyd edilməməsi ilə bağlı 2 ölçülü massiv nümunəsinə baxdıq, amma çoxölçülü massivlərdə bir neçə ölçü buraxıla, qeyd edilməyə bilər. Bir şərtlə ki, birinci ölçü qeyd edilsin və buraxılan ilk ölçüdən sonra heç bir ölçü qeyd edilməsin (sağda). Nümunə üzərindən daha aydın olacaq:

```
int a[][][][] = new int[4][3][3][2];
int b[][][][] = new int[4][3][][];
int c[][][][] = new int[4][][][];
int d[][][][] = new int[4][][3][];  // does not compile
```

Qeyd etmişdik ki, hər massiv bir obyektdir. Biz boş massiv yaratmaq üçün `{}` - fiqurlu mötərizələrdən istifadə edirik. Bu boş massiv özü bir obyekt hesab edilsə də biz bu sintaksisi birbaşa `Object` tipində olan dəyişənə mənimsədə bilmərik. Nümunə üzərindən baxsaq daha anlaşıqlı olacaq:

```
String strArr[] = {};
Object objArr1[] = {};
Object obj2 = strArr;
Object obj3 = {};                    // does not compile
Object objArr2[] = {"str", 7, {} };  // does not compile
```

Sonuncu sətirdə `7` primitiv tip olsa da autoboxing sayəsində `Integer`'ə çevrilir və `Object` tip hesab edilir. Bu səbəbdən xəta vermir, xəta verən sonuncu elementdir.

Tutaq ki, 4 ölçülü massiv elan etmişik. Bu massivə mənimsədilən dəyərin də 4 ölçülü olub olmamasını təyin etmək üçün açılan mötərizələrin sayına fikir vermək lazımdır.

```
String[][][][] array4D =
        { { null, { {}, { "x" }, null }, { { "y" } }, { { "z", "p" }, {} } } };
```

```
 {  { null, {  {  }, { "x" }, null }, { { "y" }  }, {  { "z", "p" }, {  }  }  }  }
 1o 2o      3o 4o 4c 4o   4c      3c 3o4o   4c 3c 3o 4o         4c 4o 4c 3c 2c 1c
```

o - opening curly brace
c - closing curly brace

Ətraflı:

http://www.coderanch.com/t/648777/ocajp/certification/Review-Multidimensional-Array

Using a Multidimensional Array

```
int[][] twoD = new int[3][2];
for(int i=0; i < twoD.length; i++){
   for(int j=0; j < twoD[i].length; j++)
      System.out.print(twoD[i][j] + " ");
   System.out.println();
}
```

Output:

```
0 0
0 0
0 0
```

forEach dövrü ilə bunu daha sadə formada yazmaq olar:

```
for(int[] inner: twoD){
   for(int num: inner)
      System.out.print(num+" ");
   System.out.println();
}
```

Understanding an ArrayList

Massivin bir mənfi cəhəti var ki, biz massivi yaradanda onun neçə elementdən ibarət olacağını əvvəlcədən təyin etməliyik. Amma listin üstünlüyü odur ki, sonradan genişlənə bilər.

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10); //10-specific num of slots, but not to assign any
ArrayList list3 = new ArrayList(list2);
System.out.println(list2.size());    // 0
```

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
ArrayList<> list55 = new ArrayList<String>();  // DOES NOT COMPILE
ArrayList<String> list555 = new ArrayList();
ArrayList list6 = new ArrayList<String>();
ArrayList list7 = new ArrayList<>();
```

Son iki nümunə *"mixing generic with non-generic code"* ilə əlaqəlidir. Ətraflı:

http://www.coderanch.com/t/652334/ocajp/certification/difference-ArrayList-ArrayList

`ArrayList` `List` interfeysini implement edir. `List` referansına `ArrayList` obyekti mənimsətmək mümkündür, amma əksi mümkün deyil.

```
List<String> list8 = new ArrayList<>();
List<String> list9 = new ArrayList();
ArrayList<String> list10 = new List<>();       // doesn't compile
ArrayList<> list11 = new ArrayList<String>();  // doesn't compile
ArrayList list12 = new ArrayList<String>();
list12.add(12);                                // it is valid
```

ArrayList Methods

Aşağıdakı nümunələrdə istifadə olunacaq `E` class adıdır. `ArrayList` `toString()` metodunu "implement" edir, ona görə də massivlərdən fərqli olaraq listləri `toString()` metodundan istifadə etməklə print edib normal görünüşdə baxa bilərik.

add()

Quruluşu (signature):

```
boolean add(E element)
void add(int index, E element)
```

boolean `add()` geriyə həmişə `true` qaytarır.

Nümunə:

```
ArrayList<String> cars = new ArrayList<>();
cars.add("bmw");
cars.add(Boolean.TRUE);  //does not compile
```

Generic tipi `String` olduğundan başqa tipdə dəyişən qəbul etmir. Amma generic ilə non-generic`i mix edəndə kompayl səhvi vermir, sadəcə xəbərdarlıq (warning) verir.

```
ArrayList cars = new ArrayList<String>();
cars.add("bmw");
cars.add(Boolean.TRUE);
```

Başqa nümunəyə baxaq:

```
List<String> cars = new ArrayList();
cars.add("bmw");                    // [bmw]
cars.add(1, "mercedes");            // [bmw, mercedes]
cars.add(0, "kia motors");          // [kia motors, bmw, mercedes]
cars.add(1, "toyota");              // [kia motors, toyota, bmw, mercedes]
System.out.println(cars);           // [kia motors, toyota, bmw, mercedes]
```

Əgər hansısa indeksə yeni element əlavə etmək istəyirsinizsə, o zaman əlavə etmək istədiyiniz indeksin maksimum nömrəsi listin ölçüsünə (size) bərabər olmalıdır, əks halda `IndexOutOfBoundsException` verəcək. Yəni əlavə etmək istədiyiniz indeksdən əvvəlki indeksdə mütləq element olmalıdır.

```
cars.add(5, "moskvich");   // throws IndexOutOfBoundsException
```

Aşağıdakı nümunəni icra etsək, ekrana "`1 `" çap olunacaq, amma `break` ifadəsini kommentə salsaq `NullPointerException` baş verəcək. Exceptionun baş verməməsi üçün `x` dəyişəninin tipini `Integer` edə bilərik. Bu zaman ekrana çap olunan yazı belə olacaq: `1 null 3`

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(null);
list.add(3);
for (int x : list) {
   System.out.print(x + " "); break;
}
```

remove()

Quruluşu (signature):

```
boolean remove(Object object)
E remove(int index)
```

Birinci metod `ArrayList`də göndərilən `object`ə uyğun gələn **birinci** elementi silir və nəticə olaraq geriyə `true` qaytarır. Yox əgər həmin element listdə mövcud deyilsə, geriyə `false` qaytarır.

İkinci metod göndərilən indeksdə olan elementi silir və geriyə sildiyi indeksdə mövcud olan elementi qaytarır. Əgər həmin indeks listin ölçüsündən böyük olarsa exception baş verir.

```
List<String> cars = new ArrayList();
cars.add("bmw");                              // [bmw]
cars.add("mercedes");                         // [bmw, mercedes]
cars.add("bmw");                              // [bmw, mercedes, bmw]
System.out.println(cars.remove("toyota"));    // false
System.out.println(cars.remove("bmw"));       // true
```

```
System.out.println(cars.remove(0));          // mercedes
System.out.println(cars);                    // [bmw]
```

Bu metod ilə bağlı ən çaşdırıcı nümunələr adətən aşağıdakı formada olur. Sizcə nə çap olunacaq?

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.remove(1);
System.out.println(list);
```

Göründüyü kimi listə `1` və `2` rəqəmləri əlavə edilir və sonra `remove` metodu çağırılır. Bu proseslərdən sonra ağıla ilk gələn fikir o olur ki, `1` silinir və `2` qalır. Son olaraq da `2` çap olunur. Amma əslində elə deyil. `remove()` overload metod olduğundan, overload qaydalarına görə (bu barədə chapter 4'də məlumat veriləcək) "signature"də göstərilən 2-ci metod çağırılır. Ona görə də bu nümunədəki `remove` metodu elementin dəyərini deyil, onun indeksini göstərir. Bu səbəbdən 1-ci indeksdəki element (`2`) silinir. Əgər `1` dəyərinin silinməsini istəyiriksə, o zaman `remove` metodunu aşağıdakı formada yazmalıyıq:

```
list.remove(new Integer(1));
```

set()

Quruluşu (signature):

```
E set(int index, E newElement)
```

Bu metod `ArrayList`in qeyd olunmuş indeksdəki elementini `newElement` ilə əvəz edir və geriyə əvəz edilmiş yeni elementi qaytarır. Bu zaman listin ölçüsü dəyişilmir.

```
List<String> cars = new ArrayList();
cars.add("ford");                        // [ford]
System.out.println(cars.size());         // 1
cars.set(0, "honda");                    // [honda]
System.out.println(cars.size());         // 1
cars.set(1, "lada");                     // throws IndexOutOfBoundsException
```

isEmpty() and size()

Quruluşu (signature):

```
boolean isEmpty()
int size()
```

Nümunə:

```
System.out.println(cars.isEmpty());  // true
System.out.println(cars.size());     // 0
```

```
cars.add("nissan");
cars.add("ferrari");
System.out.println(cars.isEmpty());  // false
System.out.println(cars.size());     // 2
```

clear()

Quruluşu (signature):

```
void clear()
```

Nümunə:

```
ArrayList<String> cars = new ArrayList<>();
cars.add("infiniti");       // [infiniti]
cars.add("mazda");          // [infiniti, mazda]
cars.clear();               // []
```

contains()

Quruluşu (signature):

```
boolean contains(Object object)
```

Nümunə:

```
ArrayList<String> cars = new ArrayList<>();
cars.add("kamaz");                          // [kamaz]
System.out.println(cars.contains("kamaz")); // true
System.out.println(cars.contains("maz"));   // false
```

Bu metod uyğunluq olub olmadığını yoxladığına görə hər element üçün `equals()` metodunu çağırır.

equals()

Quruluşu (signature):

```
boolean equals(Object object)
```

`ArrayList equals()` metodunu implement/override edir, ona görə də iki listdə eyni elementlərin eyni ardıcıllıqda gəlib gəlmədiyini müqayisə etmək olar.

Nümunə:

```
List<String> one = new ArrayList<>();
List<String> two = new ArrayList<>();
System.out.println(one.equals(two));     // true
one.add("a");                            // [a]
System.out.println(one.equals(two));     // false
two.add("a");                            // [a]
```

```
System.out.println(one.equals(two));    // true
one.add("b");                           // [a, b]
two.add(0,"b");                         // [b, a]
System.out.println(one.equals(two));   // false, ardıcıllıq fərqli olduğuna görə
```

sublist()

Quruluşu (signature):

```
List<E> subList(int fromIndex, int toIndex)
```

Geriyə yeni list qaytarır. `fromIndex` daxildir, `toIndex` daxil deyil.

Nümunə:

```
List s1 = new ArrayList( );
s1.add("a");
s1.add("b");
s1.add(1, "c");
List s2 = new ArrayList(s1.subList(1, 2));
s1.addAll(s2);
System.out.println(s1);  // [a, c, b, c]
```

`String`, `StringBuilder` və `ArrayList` metodlarını suallarda tez-tez bir-biri ilə qarışdırırdım, hansı metodun hansı classa aid olduğunu həmin vaxt dəqiq xatırlaya bilmirdim və bu da səhv cavablarla nəticələnirdi. Bu hal bir neçə dəfə təkrarlandığı üçün həmin metodları cədvəl şəklində qruplaşdırdım. Cədvəldə əsasən imtahanda düşən metodlar qeyd olunub. Eyni sətirdə qeyd olunan metodlar mahiyyət baxımından təxminən bir-biri ilə eyni işi görürlər.

String	StringBuilder	ArrayList
`concat()`	`append(), insert()`	`add()`
`replace()` *(2 parametrli)*	`replace()` *(3 parametrli)*	`set()`
`length()`	`length()`	`size()`
	`deleteCharAt()`	`remove()`
	`delete()`	`clear()`
`equals(), equalsIgnoreCase()`		`equals()`
`contains()`		`contains()`
		`isEmpty()`
`charAt()`	`charAt()`	
`indexOf()`	`indexOf()`	

String	StringBuilder	ArrayList
substring()	substring()	
toLowerCase()		
toUpperCase()		
startWith()		
endWith()		
trim()		
	reverse()	
	toString()	

Wrapper Classes

Hər bir primitiv tipin özünə uyğun obyekt tipində Wrapper classı var.

```
// parametr olaraq long və String qəbul edir
Long L1 = new Long(1_000_000_000_000);   // doesn't compile
Long L2 = new Long(1_000_000_000_000L);
Long L3 = new Long("5");
Long L4 = 5;            // doesn't compile
Long L5 = (long) 5;
Long L6 = 5L;

// new Float() parametr olaraq float, double və String qəbul edir
Float f = 5;       // doesn't compile
Float f1 = 5.2;    // doesn't compile
Float f2 = 5.2f;
Float f3 = new Float(5.2);
Float f4 = new Float("5.2");
```

Aşağıdakı cədvəldə Wrapper classlar vasitəsi ilə konstruktor yaradılması nümunələri göstərilib:

Primitiv tip	Wrapper class	Konstruktor nümunəsi
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)

Primitiv tip	Wrapper class	Konstruktor nümunəsi
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

String`i wrapper class və ya primitiv tipə çevirmək üçün parseXXX() və valueOf() metodlarından istifadə olunur.

```
int primitive = Integer.parseInt("123");  // bu metod geriyə primitive tip qaytarır
Integer wrapper = Integer.valueOf("123"); // bu metod geriyə wrapper class qaytarır
```

Doğru parametr göndərilmədikdə exception verir.

```
int bad1 = Integer.parseInt("a");           // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45");   // throws NumberFormatException
int bad3 = Integer.parseInt(null);          // throws NumberFormatException
```

Character classından başqa hamısının parse/valueOf metodları var. Aşağıdakı cədvəldə bu metodlardan istifadə edərək String`dən primitiv tipə və wrapper classa çevrilmə nümunələri göstərilmişdir:

Wrapper class	String`dən primitiv tipə çevrilmə	String`dən Wrapper classa çevrilmə
Boolean	Boolean.parseBoolean("true");	Boolean.valueOf("TRUE");
Byte	Byte.parseByte("1");	Byte.valueOf("2");
Short	Short.parseShort("1");	Short.valueOf("2");
Integer	Integer.parseInt("1");	Integer.valueOf("2");
Long	Long.parseLong("1");	Long.valueOf("2");
Float	Float.parseFloat("1");	Float.valueOf("2.2");
Double	Double.parseDouble("1");	Double.valueOf("2.2");
Character	None	None

`Boolean` ilə bağlı müəyyən maraqlı qaydalar var, Enthuware testlərinin birinin izahında rastlaşacaqsınız. Baxaq:

1. `Boolean` classının 2 konstruktoru var - `Boolean(String )` və `Boolean(boolean )`. Əgər `String` parametri `null` deyilsə və dəyəri "true" (və ya "True" yaxud "tRuE", yəni böyük-kiçik hərf duyarlı deyil) olarsa, o zaman geriyə dəyər `true` olan `Boolean` obyekti qaytarılır. Əks halda `false` qaytarır.

2. `Boolean` classının `boolean` dəyər yaratmaq üçün 2 köməkçi `static` metodu var - `parseBoolean` and `valueOf`.
`Boolean.parseBoolean(String )` metodu geriyə obyekt deyil, primitiv tip qaytarır. Yuxarıda qeyd etdiyimiz qayda bunun üçün də keçərlidir, yəni `null` deyilsə və dəyəri "true" olarsa, o zaman geriyə `true` qaytarır.
`valueOf` metodu overload metoddur, həm `String`, həm də `boolean` tipində parametr qəbul edir. İşləmə qaydası `parseBoolean` ilə eynidir, amma ondan fərqli olaraq geriyə referans tip qaytarır, amma yeni obyekt yaratmır.

3. `==` operatoru ilə 2 boolean dəyəri müqayisə etdikdə, əgər tərəflərdən biri `Boolean` wrapper class olarsa, o zaman o "unboxed" vasitəsilə primitiv tipə çevirilir və sonra müqayisə aparılır. Yox əgər hər iki tərəf wrapper classdırsa, o zaman onların referansları müqayisə olunur.

```
System.out.println(new Boolean("true "));  // false
System.out.println(new Boolean("tRUe"));   // true

new Boolean("true") == new Boolean("true")            // false
new Boolean("true") == Boolean.valueOf("true")        // false
new Boolean("true") == Boolean.valueOf(true)          // false
new Boolean("true") == Boolean.parseBoolean("true")   // true
new Boolean(null)   == Boolean.parseBoolean(" true ") // true
new Boolean("true") == Boolean.parseBoolean(true)     // does not compile

Boolean.valueOf("true") == Boolean.TRUE                   // true
new Boolean("true") == Boolean.TRUE                       // false
Boolean.valueOf("truE") == Boolean.parseBoolean("True")   // true
```

Fərqli Wrapper classlara aid referansları `==` operatoru ilə müqayisə etmək mümkün deyil, kompayl xətası verəcək:

```
Short k = 9;
Integer w = 9;
System.out.println(k == w);   // does not compile

short s = 9;
int i = 9;
System.out.println(s == i);  // true
```

Aşağıdakı nümunəyə bənzər nümunə ilə Enthuware'də rastlaşacaqsınız. Ekrana nəyin çap olacağını tapmağa çalışın. Əgər cavabınız doğru olmasa kodu incələyin, output'dan artıq sizə çox şey məlum olacaq və bir yeni şey də öyrənmiş olacaqsınız.

```
public static void main(String[] args) {
   Float f = null;
   try {
      f = Float.valueOf("11");
      String s = f.toString();
      double d = Double.valueOf(s);
      int i = Integer.parseInt(s);
      System.out.println(d + " " + i);
   } catch (NumberFormatException e) {
      System.out.println(f);
   }
}
```

Autoboxing

autoboxing: primitive → wrapper → `Integer w = 100;`
unboxing: primitive ← wrapper → `int p = new Integer(100);`

```
Double d1 = 50;               // doesn't compile
Double d2 = new Double(50);
double d3 = 50;
Double d4 = 50.;

List<Double> weights = new ArrayList<>();
weights.add(50.5);                        // [50.5]
weights.add(new Double(60));              // [50.5, 60.0]
weights.remove(50.5);                     // [60.0]
double first = weights.get(0);            // 60.0  - this is unbox.

List<Integer> heights = new ArrayList<>();
heights.add(null);
int h = heights.get(0);   // NullPointerException
```

`Integer`ə autoboxing`lə bağlı suallara diqqətlə baxmaq lazımdır. Aşağıdakı nümunə çaşdırıcıdır:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);  // [1]
```

`remove()` metodu burada `1` dəyərini deyil, 1-ci indeksdə olan dəyəri silir, yəni `2`-ni. Əgər `1` dəyərinin silinməsini istəyiriksə, o zaman parametr olaraq `int` deyil, obyekt tipli dəyişən göndərməliyik, yəni:

```
numbers.remove(new Integer(1));
```

Aşağıdakı nümunə kompayl xətası vermir, amma icra vaxtı exception fırladır:

```
int j = new Integer(null);   // throws NumberFormatException
```

Converting Between array and List

`ArrayList`in massivə çevrilməsi üçün `List` interfeysinin `toArray()` metodundan istifadə edilir:

```
3:  List<String> list = new ArrayList<>();
4:  list.add("element1");
5:  list.add("element2");
6:  Object[] objectArray = list.toArray();
7:  System.out.println(objectArray.length);         // 2
8:  String[] stringArray = list.toArray(new String[0]);
9:  System.out.println(stringArray.length);         // 2
```

`toArray()` metodu geriyə `Object` tipli massiv qaytarır, **ona görə də default olaraq yaranan massivin tipi Object tipidir**. Əgər massivin tipini xüsusi olaraq təyin etmək istəyiriksə, o zaman 8-ci sətirdəki kimi həmin tipi `toArray()` metoduna parametr olaraq göndəririk. `new String[0]` belə yazdıqda 0-a görə mənimsədilən listin öz ölçüsünü massivə verir. Ancaq 0-ın əvəzinə yazılan rəqəm əgər mənimsədilən listin ölçüsündən kiçik olarsa o zaman listin ölçüsü massivin ölçüsü olur, əgər böyük olarsa qalan elementlərin yeri `null` ilə doldurulur.

```
// String[] stringArray1 = list.toArray();  // DOES NOT COMPILE
String[] stringArray2 = list.toArray(new String[4]);
System.out.println(Arrays.toString(stringArray2)); // [element1,element2,null,null]
```

Əgər massivin ölçüsü qeyd olunmasa kompayl xətası verəcək:

```
String[] stringArray = list.toArray(new String[]);  // DOES NOT COMPILE
```

Massivi listə çevirmək üçün `Arrays` classının `asList()` metodundan istifadə edilir. Orijinal massiv ilə bu massiv əsasında yaradılmış list bir-biri ilə əlaqəli olur. Yəni bunlardan birinin elementlərinin dəyişilməsi hər ikisinə təsir göstərir, çünki eyni yaddaş sahəsinə müraciət edirlər (point to the same data store). Bu qayda ilə yaradılmış list həm də **fixed-size list** hesab olunur, ölçüsünün dəyişdirilməsinə icazə verilmir.

```
String[] array = {"element1", "element2"};  // [element1, element2]
List<String> list = Arrays.asList(array);   // returns fixed size list
System.out.println(list.size());            // 2
```

```
list.set(1, "test");                // [element1, test]
array[0] = "new";                   // [new, test]
for(String b: array)
   System.out.print(b+" ");         // new test
list2.remove(1);                    // throws UnsupportOperationException

ArrayList<String> list3 = Arrays.asList(array);  // doesn't compile
```

`asList()` metodu parametr olaraq *varargs* qəbul edir. Bir sətirdə list yaradıb elementlərini mənimsətmək mümkündür:

```
List<String>  list4 = Arrays.asList("one", "two");
List<Integer> list5 = Arrays.asList(1, 2);
List<Double>  list6 = Arrays.asList(1., 2.);

List<Integer> list = Arrays.asList(10, 4, -1, 5);
Collections.sort(list);
Integer[] arr = list.toArray(new Integer[4]);
System.out.println(arr[0]);  // -1
```

Sorting list

Massivləri *sort* etmək üçün `java.util` paketinin `Arrays.sort()` metodundan istifadə edirdik, `ArrayList`i sort etmək üçün isə həmin paketin `Collections.sort()` metodu istifadə edilir.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
java.util.Collections.sort(numbers);
System.out.println(numbers);        // [5, 81, 99]
```

Sıralamada rəqəmlər həmişə hərflərdən, həmçinin böyük hərflər də kiçik hərflərdən əvvəl gəlir, və `String` dəyişənin sıralaması əlifba sırasına uyğun aparılır, ona görə də `30` `8`dən öncə gəlir.

```
List<String> hex = Arrays.asList("30", "8", "3A", "FF");
Collections.sort(hex);
System.out.println(hex);                    // 30 3A 8 FF
int k = Collections.binarySearch(hex, "8");
int m = Collections.binarySearch(hex, "3A");
int n = Collections.binarySearch(hex, "4F");
System.out.printf("%d %d %d%n", k, m, n);    // 2 1 -3
```

Working with Dates and Times

Bu mövzu java 8-ci versiya ilə gəlmiş yeni mövzulardan biridir. Köhnə versiyalardakı tarix və vaxtla bağlı classlardan istifadə edərək hələ də işləmək mümkündür, lakin onlar imtahan suallarına salınmayacaq, ancaq yeni mövzu ilə əlaqəli suallar olacaq. Tarix və vaxtla bağlı yeni classlardan istifadə edə bilmək üçün ilk öncə `java.time` paketi import olunmalıdır.

```
import java.time.*;
```

Bu paketə daxil olan və imtahanda istifadə olunan əsas classlar `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration` və başqalarıdır. Bu classlar hamısı **immutable** və thread safe`dir.

Creating Dates and Times

Bu mövzu ilə bağlı imtahanda əsas 3 seçim (class) verilir:

LocalDate – saat və saat qurşağı (time zone) daxil deyil, ancaq tarixi göstərir. Nümunə üçün sizin ad günü tarixiniz buna misal ola bilər.

LocalTime – tarix və saat qurşağı daxil deyil, ancaq vaxtı/saatı göstərir.

LocalDateTime – həm tarixi, həm də saatı göstərir, lakin saat qurşağı daxil deyil.

Oracle ehtiyac olmadıqca saat qurşağından istifadə etməməyi məsləhət görür. Cari tarix və ya vaxtı öyrənmək üçün hər üç classın static `now()` metodundan istifadə edilir.

```
System.out.println(LocalDate.now());         // 2015-08-12
System.out.println(LocalTime.now());         // 12:40:37.095
System.out.println(LocalDateTime.now());     // 2015-08-12T12:40:37.095
```

`LocalDateTime`ı `String`ə çevirən zaman tarix və saatı bir-birindən ayırmaq üçün java `T` simvolu istifadə edir. Bir də yadda saxlamaq lazımdır ki, tarix formatında ay həmişə gündən əvvəl gəlir.

Hər hansı bir spesifik tarix və ya vaxt yaratmaq istəyiriksə, `of()` metodundan istifadə edirik.

LocalDate of() method signature:

```
public static LocalDate of(int year, Month month, int dayOfMonth)
public static LocalDate of(int year, int month, int dayOfMonth)

LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 10);   // 2015-01-10
LocalDate date2 = LocalDate.of(2015, 1, 10);               // 2015-01-10
```

`of()` metoduna 2-ci parametr olaraq həm `enum`, həm də `int` tipində dəyişən göndərmək mümkündür. `int` tipində parametr göndərərkən diqqət etmək lazımdır ki, **say 0-dan deyil 1-dən başlamalıdır.**

LocalTime of() method signature:

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanoOfSecond)

LocalTime time1 = LocalTime.of(6, 15);            // 06:15
LocalTime time2 = LocalTime.of(6, 15, 30);        // 06:15:30
LocalTime time3 = LocalTime.of(6, 15, 30, 200);   // 06:15:30.000000200
```

LocalDateTime of() method signature:

```
public static LocalDateTime of(int year, Month month, int day, int hour, int min)
public static LocalDateTime of(int year, Month month, int day, int hour, int min, int sec)
public static LocalDateTime of(int y, Month m, int day, int hour, int min, int sec, int nanos)
public static LocalDateTime of(int year, int month, int day, int hour, int minute)
public static LocalDateTime of(int year, int month, int day, int hour, int min, int sec)
public static LocalDateTime of(int year, int m, int day, int hour, int min, int sec, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime time)

LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 10, 6, 15);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

`LocalDate`, `LocalTime` və `LocalDateTime` classlarının konstruktoru `private` olduğundan `new` açar sözü ilə birbaşa obyektini yaratmaq mümkün deyil:

```
LocalDate d = new LocalDate();    // does NOT compile
```

Digər çaşdırıcı məqam isə `of()` metoduna parametr kimi doğru olmayan dəyərlərin göndərilməsidir:

```
LocalDate.of(2015, Month.JANUARY, 32); // DateTimeException: Invalid value for DayOfMonth
                                       // (valid values 1 - 28/31): 32
```

Yeni üsulla köhnə üsulun fərqi:

	Old way	**New way (Java 8)**
Importing	`import java.util.*;`	`import java.time.*;`
Creating an object with the current date	`Date d = new Date();`	`LocalDate d = LocalDate.now();`

	Old way	New way (Java 8)
Creating an object with the current date and time	`Date d = new Date();`	`LocalDateTime dt = LocalDateTime.now();`

Creating an object representing January 1, 2015	`Calendar c = Calendar.getInstance();`
	`c.set(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
	or
	`Calendar c = new GregorianCalendar(2015, Calendar.JANUARY, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, Month.JANUARY, 1);`

Creating January 1, 2015 without the constant	`Calendar c = Calendar.getInstance();`
	`c.set(2015, 0, 1);`
	`Date jan = c.getTime();`
		`LocalDate jan = LocalDate.of(2015, 1, 1);`

Manipulating Dates and Times

Date və time classları `String` kimi immutable`dir. Ona görə də edilmiş hər hansı bir dəyişikliyin yadda saxlanılmasını istəyiriksə, həmin dəyişikliyin nəticəsini yenidən referans dəyişənə mənimsətmək lazımdır.

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date);                    // 2014-01-20
date = date.plusDays(2);
System.out.println(date);                    // 2014-01-22
date = date.plusWeeks(1);
System.out.println(date);                    // 2014-01-29
date = date.plusMonths(1);
System.out.println(date);                    // 2014-02-28
date = date.plusYears(5);
System.out.println(date);                    // 2019-02-28
date = date.minusMonths(1);
System.out.println(date);                    // 2019-01-28
date = date.plusMonths(1).plusDays(2);
System.out.println(date);                    // 2019-03-02
```

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(dateTime);                 // 2020-01-20T05:15
dateTime = dateTime.minusDays(1);
System.out.println(dateTime);                 // 2020-01-19T05:15
dateTime = dateTime.minusHours(10);
System.out.println(dateTime);                 // 2020-01-18T19:15
dateTime = dateTime.minusSeconds(30);
System.out.println(dateTime);                 // 2020-01-18T19:14:30
```

Date və time classlarını zəncirvari şəkildə də işlətmək mümkündür. Yuxarıdakı nümunəni zəncirvari şəkildə aşağıdakı kimi yaza bilərik:

```
LocalDateTime dateTime = LocalDateTime.of(date,time)
                              .minusDays(1).minusHours(10).minusSeconds(30);
System.out.println(dateTime);  // 2020-01-18T19:14:30
```

İmtahanda sizi əsas 2 yolla çaşdıra bilərlər:

1. Date və time classlarının **immutable** olduğunu yaddan çıxarmayın!

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date.plusDays(10);
System.out.println(date);    // 2020-01-20
```

2. `LocalDate` saat ilə bağlı, `LocalTime` isə tarix ilə bağlı metodları dəstəkləmir!

```
LocalDate date = LocalDate.of(2017, Month.OCTOBER, 20);
date = date.plusDays(3);
date.plusMonths(-1);    // It is the same date.minusMonths(1);
date.plusHours(11);     // DOES NOT COMPILE
```

`LocalDate`, `LocalTime` və `LocalDateTime` ilə bağlı əsas metodlar aşağıdakılardır:

	`LocalDate` ilə istifadə edilə bilərmi?	**`LocalTime` ilə istifadə edilə bilərmi?**	**`LocalDateTime` ilə istifadə edilə bilərmi?**
`plusYears/minusYears`	Bəli	Xeyr	Bəli
`plusMonths/minusMonths`	Bəli	Xeyr	Bəli
`plusWeeks/minusWeeks`	Bəli	Xeyr	Bəli
`plusDays/minusDays`	Bəli	Xeyr	Bəli

	`LocalDate` ilə istifadə edilə bilərmi?	`LocalTime` ilə istifadə edilə bilərmi?	`LocalDateTime` ilə istifadə edilə bilərmi?
`plusHours/minusHours`	Xeyr	Bəli	Bəli
`plusMinutes/minusMinutes`	Xeyr	Bəli	Bəli
`plusSeconds/minusSeconds`	Xeyr	Bəli	Bəli
`plusNanos/minusNanos`	Xeyr	Bəli	Bəli

Working with Periods

Period yaratmağın 5 yolu var:

```
Period annually = Period.ofYears(1);              // every 1 year
Period quarterly = Period.ofMonths(3);            // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3);       // every 3 weeks
Period everyOtherDay = Period.ofDays(2);          // every 2 days
Period everyYearAndWeek = Period.of(1, 0, 7);     // every year and 7 days
```

Burada bir istisna var. Date və time classlarından fərqli olaraq `Period` classı üçün biz zəncirvari metodlardan istifadə edə bilmərik:

```
Period wrong = Period.ofYears(1).ofWeeks(1);  // P7D - every week
```

Biz fikirləşə bilərik ki, `wrong`un nəticəsi yuxarıda qeyd etdiyimiz `everyYearAndWeek`in nəticəsi ilə eyni olacaq, amma elə deyil. `Period.ofXXX` metodları `static` olduğundan zəncirvari şəkildə işlədildikdə ancaq sonuncu metodun nəticəsi götürülür. Aşağıdakı şəkildə yazıldıqda belə yenə nəticə "every week" olur, imtahanda çaşdırmaq üçün bu cür yazılışdan istifadə oluna bilər:

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(1);          // P7D - every week
```

Amma `of()` metodu vasitəsilə biz bir perioda həm il, həm ay və həm də gün göndərə bilərik. Signature:

```
public static Period of(int years, int months, int days)
```

`Period` adətən bir gün və ondan böyük müddətlər üçün istifadə olunur. Daha qısa müddətlər (saat, dəqiqə, saniyə və s.) üçün isə `Duration` istifadə olunur. `Duration`un da işləmə prinsipi `Period` ilə təxminən eynidir, lakin `Duration` adətən imtahanda düşmür.

```
Duration duration = Duration.ofSeconds(10);    // PT10S
```

Periodla bağlı sonuncu diqqət etməli olduğumuz məqam periodun hansı obyektlərlə işlənib işlənə bilməməsinə diqqət etməkdir.

```
LocalDate date = LocalDate.of(2015, 1, 20);
LocalTime time = LocalTime.of(6, 15);
LocalDateTime dateTime = LocalDateTime.of(date, time);
Period period = Period.ofMonths(1);
System.out.println(date.plus(period));       // 2015-02-20
System.out.println(dateTime.plus(period));   // 2015-02-20T06:15
System.out.println(time.plus(period));       // UnsupportedTemporalTypeException
Duration duration = Duration.ofSeconds(10);
System.out.println(time.plus(duration));     // 06:15:10
```

Formatting Dates and Times

Hər hansı tarix və ya vaxtdan müəyyən məlumatların əldə edilməsi üçün bir sıra metodlar mövcuddur:

```
LocalDate date = LocalDate.now();       // 2015-08-14
System.out.printf("%s, %s, %d, %d%n",
                                    date.getDayOfWeek(),    /* FRIDAY */
                                    date.getMonth(),        /* AUGUST */
                                    date.getYear(),         /* 2015   */
                                    date.getDayOfYear() );  /* 226    */
```

Tarix və ya vaxt ilə bağlı məlumatları istədiyin formatda görüntüləyə bilmək üçün `java.time.format` paketində mövcud olan `DateTimeFormatter` classından istifadə olunur. Standart ISO forması aşağıdakı kimidir:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
// date.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));  // throws Exception
```

Output:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

Lakin əvvəlcədən yaradılmış bir neçə faydalı format da mövcuddur. İmtahanda əsasən `SHORT` və `MEDIUM` formatları düşür:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
```

```
DateTimeFormatter shortDate = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
DateTimeFormatter shortTime = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
DateTimeFormatter shortDT = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter medDT = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);

System.out.println(shortDate.format(dateTime));     // 1/20/20
System.out.println(shortDate.format(date));         // 1/20/20
System.out.println(shortDate.format(time));   //UnsupportedTemporalTypeException

/* Yerlərini dəyişmək də olar */
System.out.println(dateTime.format(shortDate));     // 1/20/20
System.out.println(date.format(shortDate));         // 1/20/20
System.out.println(time.format(shortDate));   //UnsupportedTemporalTypeException

System.out.println(shortDT.format(dateTime));   // 1/20/20 11:12 AM
System.out.println(medDT.format(dateTime));     // Jan 20, 2020 11:12:34 AM
```

Əgər hazır formatları istifadə etmək istəməsəniz, öz istədiyiniz formatı da yarada bilərsiniz.

```
DateTimeFormatter f1 = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
DateTimeFormatter f2 = DateTimeFormatter.ofPattern("d/MMM/yy, hh:mm:ss");
System.out.println(dateTime.format(f1));     // January 20, 2020, 11:12
System.out.println(dateTime.format(f2));     // 20/Jan/20, 11:12:34

DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");
f.format(dateTime); // 11:12
f.format(date);     // UnsupportedTemporalTypeException
f.format(time);     // 11:12
```

MMMM – ayı göstərir. Məsələçün, `M - 1, MM - 01, MMM - Jan, MMMM - January.`
dd – ayın gününü göstərir (day of month). `d - 1, dd - 01.`
yyyy – ili göstərir. `yy - 15, yyyy - 2015.`
hh – saatı göstərir.
: – ayrıcı kimi istifadə olunur.
mm – dəqiqəni göstərir.

Format nümunələrində pattern`ə diqqətlə baxmaq lazımdır. Aşağıdakı nümunədə böyük yaxud kiçik hərf duyarlılığına görə nəticələr də fərqli olur:

```
LocalDateTime dateTime = LocalDateTime.of(2017, 6, 23, 10, 40);
DateTimeFormatter f1 = DateTimeFormatter.ofPattern("hh:mm");
DateTimeFormatter f2 = DateTimeFormatter.ofPattern("hh:MM");
f1.format(dateTime);   // 10:40
f2.format(dateTime);   // 10:06
```

`parse()` metodundan istifadə etməklə də tarix və ya vaxt obyektləri yaratmaq mümkündür, bunun üçün `parse()` metoduna lazımı formatda dəyərlər göndərmək lazımdır:

```
LocalDate dt = LocalDate.parse("2015-01-01").minusMonths(1)
                                            .minusDays(1).plusYears(1);
System.out.println(dt);  // 2015-11-30
```

`parse()` metodundan sonra yuxarıda gördüyünüz nümunədəki kimi zəncirvari şəkildə digər `LocalDate` metodlarını çağırmaq olar, bu metodların hər biri geriyə yeni `LocalDate` obyekti qaytarır. Amma `parse()` metodunu istifadə edən zaman göndərilən parametrə diqqət etmək lazımdır. Məsələn, aşağıdakı nümunədə `ldt` dəyişəni `LocalDateTime`, amma pattern `LocalDate` tipindədir. Pattern`də saatla bağlı məlumat qeyd olunmadığına görə icra vaxtı `DateTimeParseException` baş verəcək. Amma pattern kommentdə qeyd olunan formada olsa exception baş verməyəcək:

```
CharSequence pattern = "2017-06-23";  // 2017-06-23T12:20 is valid
LocalDateTime ldt = LocalDateTime.parse(pattern);
System.out.println(ldt);
```

Əlavə olaraq qeyd edək ki, `DateTimeParseException RuntimeException`dan törədiyi üçün "handle" və ya "declare" etməyə ehtiyac yoxdur. Bu mövzu Chapter 6-da geniş şəkildə qeyd ediləcək.

Aşağıdakı cədvəldə formatlaşdırma ilə bağlı olan `ofLocalized` metodlarının qısaca təsviri verilib:

`DateTimeFormatter f = DateTimeFormatter.`_____`(FormatStyle.SHORT);`	**Calling `f.format(localDate)`**	**Calling `f.format(localDateTime)`**	**Calling `f.format(localTime)`**
`ofLocalizedDate`	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
`ofLocalizedDateTime`	Throws runtime exception	Legal – shows whole object	Throws runtime exception
`ofLocalizedTime`	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

Əlavələr

Nümunə 1:

```
String s = "";
StringBuilder sb = new StringBuilder();
int[] array = new int[2];
List list = new ArrayList();

s.length();
s.capacity();      // DOES NOT COMPILE
s.size();          // DOES NOT COMPILE

sb.length();
sb.capacity();     // ancaq StringBuilder`ə aiddir
sb.size();         // DOES NOT COMPILE

array.length();    // DOES NOT COMPILE
array.length;      // DOES NOT COMPILE --> not a statement
System.out.println(array.length);
array.capacity();  // DOES NOT COMPILE
array.size();      // DOES NOT COMPILE

list.length();     // DOES NOT COMPILE
list.capacity();   // DOES NOT COMPILE
list.size();
```

Nümunə 2:

Aşağıdakı kodu icra etdikdə ekrana nə çap ediləcək?

```
3:  int[] times[] = new int[3][3];
4:  for (int i = 0; i < times.length; i++)
5:     for (int j = 0; j < times.length; j++)
6:        times[i, j] = i * j;
7:     System.out.println(times[2, 2]);
```

A. `1`
B. `4`
C. `1` printed 4 times
D. `4` printed 3 times
E. An exception is thrown.
F. The code fails to compile because of line 3.
G. The code fails to compile for another reason.

Nəticəni hesablamağa çox vaxt sərf etdiniz? Və cavab olaraq **B** variantını tapdınız? Onda təkrar bir də cəhd edin. Yenə **4** cavabını aldınız? Bir daha cəhd edin!
Yenə alınmadı? O zaman kağız, qələm götürün və bu kodu vərəqdə yazmağa çalışın. Yazdığınız kod ilə burada gördüyünüz kodu müqayisə edin. İndi tapdınızmı?

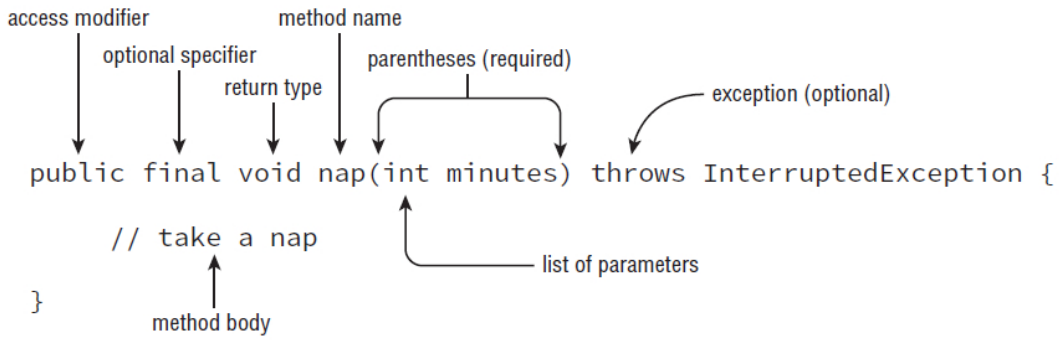
Nümunə 3:

```
StringBuilder s1 = new StringBuilder("meow");
StringBuilder s2 = new StringBuilder("meow");
System.out.println(s1 == s2);                  // false
System.out.println(s1.equals(s2));             // false
System.out.println(s1.toString() == "meow");   // false
System.out.println(s1 == "meow");              // DOES NOT COMPILE
```

Chapter 4. Methods and Encapsulation

Designing Methods

Şəkil 4.1 Metodun strukturu



Qısaca qeyd etsək:

Element	Value in nap() example	Required?
Access modifier	`public`	No
Optional specifier	`final`	No
Return type	`void`	Yes
Method name	`nap`	Yes
Parameter list	`(int minutes)`	Yes, but can be empty parentheses
Optional exception list	`throws InterruptedException`	No
Method body	`{ // take a nap }`	Yes, but can be empty braces

Ardıcıllıq sırasına diqqət etmək lazımdır. `()`-nin solunda metodun adı gəlməlidir, metodun adından əvvəl isə mütləq tipi qeyd olunmalıdır. Access modifier və optional specifier`lərin yerlərini dəyişmək mümkündür:

```
static final int max(int x, int y) {
```

```
    return (5);   // it is legal
}
```

Access modifiers

4 access modifier var:

public – metod `public` olarsa, istənilən classdan çağırıla bilər.

private – metod `private` olarsa, ancaq classın öz daxilində çağırıla bilər.

protected – metod `protected` olarsa, eyni paketdə olan classlar və subclasslar tərəfindən çağırıla bilər.

Default (Package Private) Access – metod default olarsa (yəni heç bir açar söz istifadə edilməsə), ancaq eyni paketdə olan classlar tərəfindən çağırıla bilər.

```
public class Test {
   public void test1() {}
   default void test2() {}    // DOES NOT COMPILE
   void public test3() {}     // DOES NOT COMPILE
   void test4() {}
}
```

Optional Specifiers

Access modifierdən fərqli olaraq bir metodda bir neçə "optional specifiers" istifadə etmək mümkündür. İmtahan suallarında istifadə olunan əsasları bunlardır:

static, abstract, final – imtahan suallarında əsasən bunlar istifadə olunur;

synchronized – OCA imtahanında yoxdu, ancaq OCP imtahanında istifadə olunur;

native, strictfp – nə OCA, nə də OCP imtahanında istifadə olunmur.

```
public class Test {
   public void test1() {}
   public final void test2() {}
   public static final void test3() {}
   public final static void test4() {}
   public modifier void test5() {}      // DOES NOT COMPILE
   public void final test6() {}         // DOES NOT COMPILE
   final public void test7() {}
}
```

Return Type

Metodun mütləq tipi olmalıdır, əgər metod geriyə heç bir dəyər qaytarmazsa, onun tipi mütləq `void` yazılmalıdır, buraxıla bilməz. Əgər tip `void` olarsa, metoddan qayıdan dəyər kimi `return;` ifadəsi yazılmasına icazə verilir, yaxud da ümumiyyətlə heç nə yazılmır.

```
public void test1() {}
public void test2() { return; }
public String test3() { return ""; }
public String test4() {}                        // DOES NOT COMPILE
public test5() {}                               // DOES NOT COMPILE
String test6(int a) { if(a==4) return ""; }     // DOES NOT COMPILE
```

Metoddan qayıdan dəyər metodun tipinə uyğun olmalıdır:

```
int getInt() {
   return 9;
}
int getLong() {
   return 9L;        // DOES NOT COMPILE
}
```

Method Name

Metod adları da identifiers kimi müəyyən qaydalara riayət etməklə elan edilməlidir, bu barədə birinci bölmədə ***Identifiers*** mövzusunda qeyd olunub. Həmçinin *Java Naming Conventions*`a görə metod adları kiçik hərflə başlamalıdır, amma yazılmasa səhv sayılmır.

(konvensiya ətraflı: http://www.oracle.com/technetwork/java/codeconventions-135099.html)

```
public void test1() {}
public void 2test() {}      // DOES NOT COMPILE
public test3 void() {}      // DOES NOT COMPILE
public void Test_$() {}
public void() {}            // DOES NOT COMPILE
```

Working with Varargs

Vararg – variable argument. Massivə oxşayır, azacıq fərqlidir. Vararg parametr metodun parametr listində ən sonda gəlməlidir və bir metodda ancaq bir vararg parametr istifadə edilə bilər.

```
public void test1(int... args) {}
public void test2(int count, int... args) {}
```

```
public void test3(int... args, int args) {}          // DOES NOT COMPILE
public void test4(int... count, int... args) {}      // DOES NOT COMPILE
public void test5(int count, int args...) {}         // DOES NOT COMPILE
public void test6(int count, int.. args) {}          // DOES NOT COMPILE
```

Vararg parametrli metodu çağırarkən bir neçə seçimimiz var:

1. Parametr olaraq massiv göndərə bilərik;
2. Massivin elementlərini vergüllə ayrılmış şəkildə göndərə bilərik;
3. Ümumiyyətlə parametr göndərməyə də bilərik. Bu zaman Java ölçüsü sıfır olan massiv yaradıb göndərir.

```
public static void test(int start, int... nums) {
   System.out.println(nums.length);
}
public static void main(String[] args) {
   test(1);                              // 0
   test(1, 2);                           // 1
   test(1, 2, 3);                        // 2
   test(1, new int[] {4, 5});            // 2
   test(1, new int[8]);                  // 8
   test(1, null);                        // throws NullPointerException
}
```

Vararg parametrlərinə müraciət massivdəki kimidir, indeksdən istifadə olunur:

```
public static void show(int... nums) {
   System.out.println(nums[1]);  // 2
}
public static void main(String[] args) {
   show(1, 2);
}
```

Applying Access Modifiers

Kimlər üçün əlçatandır:

- ***private*** – ancaq eyni class daxilində;
- ***default (package private) access*** – *private* + eyni paketdəki digər classlar;
- ***protected*** – *default access* + subclasslar;
- ***public*** – *protected* + digər paketlərdəki classlar.

Cədvəl şəklinə salsaq daha aydın görsənəcək:

Access Modifier	class daxilində	paket daxilində	Digər paketlərdən ancaq subclass ilə	digər paketlərdən
`private`	bəli	xeyr	xeyr	xeyr
`default`	bəli	bəli	xeyr	xeyr
`protected`	bəli	bəli	bəli	xeyr
`public`	bəli	bəli	bəli	bəli

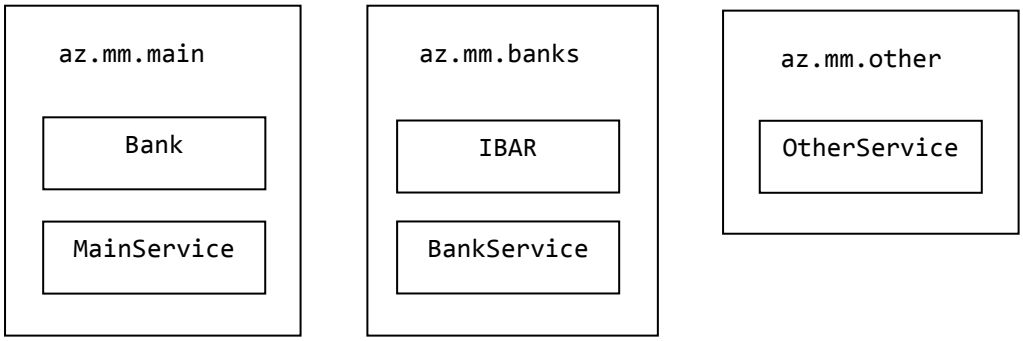
`private`, default və `public` açar sözlərinin istifadəsi `protected`ə nisbətən daha sadədir. Əvvəlcə bu modifier`lər ilə bağlı nümunələrə baxaq, sonra `protected`ə ayrıca baxarıq:

```
1:  package az.mm.A;
2:
3:  public class AccessModifiers {
4:
5:     private String secretText;
6:     String friendlyText;
7:     public String forEveryOneText;
8:
9:     public void displaySecretText() {
10:       System.out.println(secretText);     // it is ok
11:    }
12: }
13:
14: class Test {
15:    public static void main(String[] args) {
16:       AccessModifiers am = new AccessModifiers();
17:       System.out.println(am.forEveryOneText);
18:       System.out.println(am.friendlyText);
19:       System.out.println(am.secretText);     // DOES NOT COMPILE
20:    }
21: }
```

```
1:  package az.mm.B;
2:
3:  import az.mm.A.*;
4:
5:  public class Other {
6:     public void display() {
7:        AccessModifiers am = new AccessModifiers();
8:        System.out.println(am.forEveryOneText);      // it is ok
9:        System.out.println(am.friendlyText);         // DOES NOT COMPILE
10:       System.out.println(am.secretText);           // DOES NOT COMPILE
```

```
11:     }
12: }
```

`protected` ilə bağlı vəziyyət daha mürəkkəbdir. Nümunələrə baxaq:




```
1:  package az.mm.main;
2:
3:  public class Bank {
4:     protected String text = "This is not public information";
5:     protected void display(){
6:        System.out.println(text);
7:     }
8:  }
```

```
1:  package az.mm.main;
2:
3:  import az.mm.banks.IBAR;
4:
5:  public class MainService {
6:
7:     public String getInformation() {
8:        IBAR ibar = new IBAR();
9:        return ibar.text;
10:    }
11:
12:    public void showInformation() {
13:       Bank bank = new Bank();
14:       bank.display();
15:    }
16: }
```

```
1:  package az.mm.banks;
2:
3:  import az.mm.main.Bank;
4:
```

```
5:  public class IBAR extends Bank {
6:
7:     public void method1() {
8:        System.out.println(text);
9:        display();
10:    }
11:
12:    public void method2() {
13:       IBAR ibar = new IBAR();
14:       System.out.println(ibar.text);
15:       ibar.display();
16:    }
17:
18:    public void method3() {
19:       Bank bank = new Bank();
20:       System.out.println(bank.text);  // DOES NOT COMPILE
21:       bank.display();                 // DOES NOT COMPILE
22:    }
23:
24:    public void method4() {
25:       Bank ibar = new IBAR();
26:       System.out.println(ibar.text);  // DOES NOT COMPILE
27:       ibar.display();                 // DOES NOT COMPILE
28:    }
29: }
```

```
1:  package az.mm.banks;
2:
3:  public class BankService {
4:     public String getInformation(){
5:        IBAR ibar = new IBAR();
6:        return ibar.text;              // DOES NOT COMPILE
7:     }
8:  }
```

```
1:  package az.mm.other;
2:
3:  import az.mm.banks.IBAR;
4:
5:  public class OtherService {
6:     public String getInformation(){
7:        IBAR ibar = new IBAR();
8:        return ibar.text;              // DOES NOT COMPILE
9:     }
10: }
```

`MainService` classında gördüyünüz kimi eyni paketdəki digər class daxilində çağırıldıqda problem yoxdur. Amma fərqli paketdən ancaq subclass vasitəsi ilə çağırıla bilər, bu səbəbdən `IBAR` classında `method1()` və `method2()` xəta vermir. Amma subclassda `Bank` classının öz referansı vasitəsi ilə `protected` üzvlərə müraciət edilə bilməz, buna görə də `method3()` və `method4()` xəta verir. Bu nüans ilə bağlı Chapter 5`də varislik mövzusunda ayrıca təkrar baxacağıq. Amma subclassın referansı vasitəsilə fərqli paketdən çağırıldıqda yenə xəta verir (`BankService` və ya `OtherService` classlarında göstərildiyi kimi), subclassın referansı vasitəsi ilə ancaq subclassın öz daxilindən çağırmaq mümkündür.

Designing Static Methods and Fields

Static metodlar/dəyişənlər əsas iki məqsəd üçün istifadə olunur:

1. Classın obyektini yaratmağı tələb etmir, təkcə classın adını istifadə etməklə çağırmaq mümkündür.
2. Bir neçə fərqli instanslar (instance) üzərindən çağırılsa belə, onlar hamısı eyni `static` dəyişənə müraciət edir.

Calling a Static Variable or Method

Static dəyişən və metodlara qeyd etdiyimiz kimi class adını həmin dəyişən və ya metodun əvvəlində istifadə etməklə müraciət edirdik. Bununla yanaşı, obyektin instansı vasitəsilə də müraciət mümkündür:

```
class Tiger {
   public static int count = 5;
   public int age = 11;
}

class TigerTest {
   public static void main(String[] args) {
      Tiger t = new Tiger();

      System.out.println(Tiger.count);    // 5
  //  System.out.println(Tiger.age);      // DOES NOT COMPILE

      System.out.println(t.count);        // 5
      System.out.println(t.age);          // 11

      t = null;

      System.out.println(t.count);        // 5
      System.out.println(t.age);          // throws NullPointerException
}  }
```

Static dəyişən classa aiddir və ona görə də fərqli obyektlər yaradıb, onların referansları vasitəsilə bu dəyişənə dəyərlər versək, onlar hamısı eyni bir dəyişənin dəyərini dəyişəcək. Aşağıdakı kod nümunəsi yuxarıdakı nümunənin davamıdır və koddan da göründüyü kimi `count` dəyişəninin hansı obyekt üzərindən çağırılmasına baxmayaraq hamısı eyni bir dəyişənin dəyərini dəyişir:

```
Tiger.count = 4;
Tiger t1 = new Tiger();
Tiger t2 = new Tiger();
t1.count = 6;
t2.count = 5;
System.out.println(Tiger.count);  // 5
```

Static vs Instance

İmtahanda digər bir qrup çaşdırıcı suallar **static** və **instance** member`lər (*member = field + method*) ilə əlaqəli suallardır. Əsas bilinməsi vacib olan qaydalar aşağıdakılardır:

- static member instance member`i birbaşa çağıra bilməz;
- instance metod static metodun daxilində yalnız classın instansı istifadə olunaraq çağırıla bilər;
- static metodu class daxilində başqa bir static metodun içində class adı və ya instansı olmadan çağırmaq mümkündür;
- instance metod class daxilində başqa bir instance metodun içində referans dəyişəni istifadə olunmadan çağırıla bilər;
- static metod instance metodun daxilində çağırıla bilər.

```
public class Static {
   private String name = "Static class";
   public static void first() { }
   public static void second() { }
   public void third() { System.out.println(name); }    // line1

   public static void main(String[] args) {
      first();
      second();
      third();                // line2,  DOES NOT COMPILE
      new Static().third();   // line3
} }
```

Instance metod *line3*`də göstərilən qaydada çağırılmalıdır. Əgər `third()` metodunu `static` etsək, o zaman *line2* kompayl olunacaq, amma *line1*`də kompayl xətası çıxacaq. Çünki `name`

dəyişəni `static` deyil və `static` metodda ancaq classın instansı istifadə olunaraq çağırıla bilər ya da həmin dəyişən özü də `static` olmalıdır.

```
public class Gorilla {
   public static int count;
   public static void addGorilla(){ count++; }
   public void babyGorilla() { count++; }
   public void announceBabies() {
      addGorilla();
      babyGorilla();
   }
   public static void announceBabiesToEveryone(){
      addGorilla();
      babyGorilla();               // DOES NOT COMPILE
   }
   public int total;
   public static double average = total / count;      // DOES NOT COMPILE
}
```

Sonuncu sətirin kompayl olunması üçün `total` dəyişəninin də `static` olması lazımdır.

Static və instance dəyişənlərə dəyər mənimsədilməyibsə, avtomatik olaraq default dəyərlər mənimsədilir:

```
public class Counter {

   private static int count;
   private int count1;

   public Counter() {
      count++;
      count1++;
   }

   public static void main(String[] args) {
      Counter c1 = new Counter();
      c1.test();
   }

   public void test() {
      Counter c2 = new Counter();
      Counter c3 = new Counter();
      System.out.println(count);    // 3
      System.out.println(count1);   // 1
   }
}
```

Static Imports

Import ona görə əlverişlidir ki, biz hər dəfə kod daxilində hansısa classın adını istifadə edəndə onu paket adı ilə tam formada yazmayaq, paketi import edək, qısa class adından istifadə edək. Static import adi importdan fərqli olaraq classları deyil, ancaq classın static member`lərini import etmək üçündür.

```
import java.util.List;
import static java.util.Arrays.asList;             // static import
public class StaticImports {
   public static void main(String[] args) {
      List<String> list = asList("one", "two");    // without Arrays
   }
}
```

Əgər biz `StaticImports` classında `asList()` adlı yeni metod yaratsaq, import olunmuş metodla class daxilindəki metod adı eyni olacaq və bu zaman java öncəlik sırasını class daxilində yaradılmış metoda verəcək.

Bir başqa nümunəyə baxaq:

```
1:  import static java.util.Arrays;            // line1, DOES NOT COMPILE
2:  import static java.util.Arrays.asList;
3:  static import java.util.Arrays.*;          // line2, DOES NOT COMPILE
4:  class BadStaticImports {
5:     public static void main(String[] args) {
6:        Arrays.asList("one");                // line3, DOES NOT COMPILE
7:     }
8:  }
```

line1 – class `static` açar sözü ilə import edilə bilməz, static import ancaq static member`ləri import edir;

line2 – ardıcıllıq sırası pozulub, `static` sözü həmişə `import` sözündən sonra gəlməlidir;

line3 – biz line2`də ancaq `Arrays` classının `asList()` metodunu import etmişik, `Arrays` classının özünü yox.

Eyniadlı 2 static metodu və ya 2 static dəyişəni static import etdikdə kompayl xətası baş verir:

```
import static statics.A.TYPE;
import static statics.B.TYPE;         // DOES NOT COMPILE
```

Final Initialization

Final dəyişənlər elan olunarkən onlara mütləq dəyər mənimsədilməlidir və bu ancaq bir dəfə baş verə bilər, `final` dəyişənlərə 2-ci dəfə dəyər mənimsətmək mümkün deyil. Final static və final instance dəyişənlər üçün dəyər mənimsədilməsi bir az fərqlidir, baxaq.

- **final static dəyişənlərə dəyər mənimsədilə bilər:**
  1) elan olunduğu sətirdə;
  2) static initializer blokda.

- **final instance dəyişənlərə dəyər mənimsədilə bilər:**
  1) elan olunduğu sətirdə;
  2) instance initializer blokda;
  3) konstruktorda.

Konstruktor tamamlananda bütün final instance dəyişənlərin hamısına mütləq dəyər mənimsədilməlidir. Əgər qeyd olunan halların heç birində dəyər mənimsədilməsə kompayl xətası baş verəcək.

Lokal final dəyişənlərə dəyər mənimsədilməsi ilə bağlı isə mütləq məcburiyyət yoxdur. Lokal dəyişənlər üçün standart qayda var: istifadə olunmadan öncə mütləq "initialize" olunmalıdır. Lokal final dəyişənlər də sadəcə bu qaydaya riayət etməlidir, əgər istifadə olunacaqsa, dəyər mənimsədilməlidir, əks halda kompayl xətası vermir.

```
public class Finals {

   // static final variables - sf
   private static final int sf1;          // DOES NOT COMPILE
   private static final int sf2 = 10;
   private static final int sf3;
   private static int s4;
   static {
      System.out.println(s4);    // 0
      sf3 = 100;
   }

   // instance final variables - if
   private final int if1;  // kompayl olunmur, bunun xətasını əslində konstruktorda göstərir
   private final int if2 = 20;
   private final int if3;
   private final int if4;
   private int i5;

   {
      if4 = 200;
      if2 = 21;       // DOES NOT COMPILE
   }
```

```
    public Finals() {
       System.out.println(i5);      // 0
       if3 = 200;
    }

    public void method() {
       // local final variables
       final int a1;
       final int a2;
       final int a3;   // unused
       int b1;
       int b2;
       int b3;         // unused
       System.out.println(a1);      // DOES NOT COMPILE
       System.out.println(b1);      // DOES NOT COMPILE
       a2 = 30;
       System.out.println(a2);
       b2 = 3;
       System.out.println(b2);
       b2 = 300;
       System.out.println(b2);
       a2 = 15;                    // DOES NOT COMPILE
    }
}
```

Ətraflı bu linkdən baxa bilərsiniz:

http://www.coderanch.com/t/640116/ocajp/certification/Final-instance-variable-initialization-behaviour#2937716

İndi maraqlı bir kod nümunəsinə baxacağıq, yuxarıda qeyd edilən qaydalar ödənilmir, amma buna baxmayaraq kompayl xətası vermir:

```
class FinalKeywordExample {
   final boolean bool;  // line1
   static { System.out.println("static initializer"); }
   {  System.out.println("instance initializer"); }

   FinalKeywordExample() {
      throw new RuntimeException();   // line2
   }

   public static void main(String[] args) {
      FinalKeywordExample i = new FinalKeywordExample();
      System.out.println(i.bool);
   }
}
```

Koddan da gördüyünüz kimi line1`də final `bool` dəyişəni yaradılıb və nə elan olunduğu sətirdə, nə instance initializer blokda, nə də konstruktorda ona dəyər mənimsədilməyib. Amma buna baxmayaraq kompayl xətası vermir. Xəta verməməsinin səbəbi isə line2`də `RuntimeException` fırladılmasıdır. Əgər line2`ni kommentə salsaq, o zaman line1 kompayl xətası verəcək və `bool` dəyişəninə dəyər mənimsədilməsi tələb ediləcək.

Aşağıdakı nümunə sizi çaşdıra bilər. `list` və `array` referans dəyişənlərdir, yəni biz listə yeni dəyər əlavə etdikdə, yaxud massivin müvafiq indeksindəki dəyərini başqa dəyərlə əvəz etdikdə, bu `final` dəyişənə təkrar dəyər mənimsədilməsi hesab olunmur. Referans dəyişənə yeni obyekt mənimsətdikdə artıq kompayl xətası baş verir:

```
private static final ArrayList<String> list = new ArrayList<>();
private static final int array[] = {1, 2};
public static void main(String[] args) {
   list.add("changed");
   array[1] = 3;
   list = new ArrayList<>();      // DOES NOT COMPILE
   array = new int[]{4, 5};       // DOES NOT COMPILE
}
```

Aşağıdakı nümunə də maraqlı nümunədir, bənzər print ifadəsi `main` metodda xəta verir, amma `increaseX` metodunda xəta vermir.

```
public static void main(String[] args) {
   final int x = 15;
   increaseX(x);
   System.out.println(++x);  // DOES NOT COMPILE
}

public static void increaseX(int x){
   System.out.println(++x);  // prints 16
}
```

Səbəb Java`nın "pass by value" özəlliyinə görədir, yəni `x` dəyişəni `main` metodda `final` dəyişən olsa da, `increaseX` metodunda `final` dəyişən deyil. "pass by value" mövzusu ilə növbəti məqalədə tanış olacağıq.

Passing Data Among Methods

Java *"pass by value"* dildir. Yəni bu o deməkdir ki, dəyişənin kopyası yaradılır və metoda parametr kimi həmin kopya göndərilir. Metod daxilində göndərilmiş həmin parametrin dəyərinin dəyişdirilməsi (assignment) orijinal dəyərə (caller) təsir göstərmir.

```
public static void main(String[] args) {
   int age = 29;
   String name = "Mushfiq";
   changeValues(age, name);
```

```
    System.out.println(name + " " + age);    // Mushfiq 29
}
public static void changeValues(int age, String name) {
    age = 23;
    name = "Murad";
}
```

Dəyişən adı ilə metoda göndərilən parametr adı eyni olmaya bilər, sadəcə çaşdırmaq üçün imtahanda adətən eyni adlardan istifadə olunur. Əgər java "pass-by-reference" dil olsaydı, o zaman nəticə `Murad 23` olardı.

Mutable obyekt dəyişənlərini metoda parametr olaraq göndərərkən diqqətli olmaq lazımdır. `String` immutable dəyişəndir, ona görə də göndərilmiş metodda ona yeni dəyər mənimsədilməsi onun orijinal dəyərini dəyişmir. `StringBuilder` ilə isə vəziyyət bir az fərqlidir, nümunəyə baxaq:

```
public static void main(String[] args) {
    StringBuilder name = new StringBuilder("Mushfiq");
    changeName(name);
    System.out.println(name);    // Murad
}
public static void changeName(StringBuilder sb) {
    sb.delete(0, sb.length()).append("Murad");
}
```

Burada `name` referans dəyişəndir və `sb` də `name` dəyişəninin kopyasıdır və beləcə hər iki referans eyni `StringBuilder` obyektinə işarə edir. Buna görə də `sb` referansından istifadə edərək müvafiq metodları çağırmaqla həmin `StringBuilder` obyektinin dəyərini dəyişmək mümkündür. Burada yenidən dəyər mənimsətmədən (reassign) söz gedə bilməz. Yenidən dəyər mənimsətmə olarsa, o zaman orijinal dəyər yenə dəyişilməz olaraq qalır.

```
public static void main(String[] args) {
    int[] arr = {1, 3, 5, 7};
    changeArray(arr);
    System.out.println(Arrays.toString(arr));    // [0, 4, 5, 7]
}
public static void changeArray(int[] arr) {
    arr[0] = 0;
    arr[1]++;
    arr = new int[4];  // line1
    arr[2] = 9;
}
```

Koddan göründüyü kimi line1 sətrində `arr` dəyişəninə yeni massiv mənimsədilir və bu səbəbdən sonuncu sətirdə `arr` massivinin 2-ci indeksinə yeni dəyər mənimsədilsə də bu original `arr` massivinə təsir etmir. Bütün bu yuxarıda qeyd etdiklərimizi yekunlaşdırıb qısaca

belə deyə bilərik ki, əgər metoda göndərilən parametr Object olarsa, bu zaman obyektin referansının kopyası göndərilir və bu referans özündə obyektin heap yaddaşda mövcud olduğu ünvanı (the address of the memory location) saxlayır. Həmin ünvan üzərindən artıq eyni obyektə müraciət etmək mümkündür. Əgər metod daxilində referansa yeni dəyər mənimsədilərsə, artıq yaddaş ünvanı dəyişir və ondan sonrakı müraciətlər original obyektə təsir etmir.

Qeyd etdik ki, primitiv yaxud referans (immutable) dəyişənə parametr olaraq göndərdiyimiz metodda dəyər mənimsədilirsə (assigning) bu orijinal dəyərə təsir göstərmir. Əgər metodda baş vermiş dəyişikliyin orijinal dəyərdə yansımasını istəyiriksə, o zaman metoddan qayıdan yeni dəyəri həmin orijinal dəyişənə mənimsətmək lazımdır. Əks halda dəyişilmiş dəyərdən imtina edilir.

```
public class ReturningValues {
   public static void main(String[] args) {
      int number = 1;                          // 1
      String letters = "abc";                  // abc
      number(number);                          // 1
      letters = letters(letters);              // abcd
      System.out.println(number + letters);    // 1abcd
   }
   public static int number(int number){
      number++;
      return number;
   }
   public static String letters(String letters){
      letters += "d";
      return letters;
   }
}
```

Overloading Methods

Eyni ada, lakin müxtəlif parametr listinə (different method signature) sahib olan metodlar *overload* metodlar adlanır. Signature`nin tərkibinə daxildir:

1) metod adı;
2) parametr siyahısı.

Overload metodlarda metod adından başqa hər şey fərqli ola bilər (məsələn: access modifiers, specifiers (like static), return types və exception lists).

Overload metodlar əsasən parametr listinə görə təyin olunur, yəni bu metodlar üçün vacib qayda budur ki, **metod adı eyni olsun, parametr listi fərqli.** Fərqli dedikdə nə nəzərdə tutulur:

- parametrlərin sayı müxtəlif olmalıdır;
- parametrlərin sayı eyni ola bilər, o halda ki:
  - tipləri fərqli olmalıdır;
  - ardıcılıq fərqli olmalıdır.

Nümunə:

```
5:   public void fly(int i) {}
6:   public void fly(short s) {}
7:   public boolean fly() { return false; }
8:   void fly(int i, short s) {}
9:   public void fly(short s, int i) throws Exception {}
10:  int fly(int i, short s){ return 0; }            // doesn't compile, for line 8
11:  public static final void fly(short a, int b){}  // doesn't compile, for line 9
```

Overloading and Varargs

Java varargs`la massiv kimi davranır. Əgər biz parametr olaraq `int[]` göndərsək, hansı metod çalışacaq?

```
public void fly(int[] array) {}
public void fly(int... varargs) {}     // DOES NOT COMPILE
```

2-ci metod kompayl olunmayacaq, çünki java bunların hər ikisi üçün metod signature`ni eyni cür görür. Əgər ayrı-ayrılıqda kompayl olunsa, parametr olaraq massiv göndərilsə hər iki metod çalışacaq. Yalnız varargs`ın çalışmasını istəyiriksə, massiv elementlərini tək-tək göndəririk:

```
fly(new int[] {1, 2, 3} );    // both of them
fly(1, 2, 3);                 // only varargs
```

Autoboxing

```
public void fly(int i){}
public void fly(Integer i){}
```

Əgər bizim `int` və `Integer` parametr qəbul edən metodlarımız varsa, `fly(5);` olaraq çağırsaq 1-ci metod işləyəcək, 1-ci metod olmasa, o zaman 2-ci metod çağırılacaq. Çünki java həmişə ilk olaraq dəqiq uyğunluq olan parametri tapmağa çalışır. Əgər `int` parametrli metod mövcuddursa, heç bir səbəb yoxdur ki, java `int`i `Integer`ə autoboxing edərək əlavə iş görsün. Əgər `Integer i = 5;` elan edib `i` dəyişənini `fly` metoduna göndərsək o zaman 2-ci metod işləyəcək.

Primitive and Reference Types

Qeyd etdiyimiz kimi birinci dəqiq uyğunluq tapılır, sonra digər etaplar nəzərdən keçirilir. Aşağıdakı kod nümunəsində əgər bizim `int` parametr qəbul edən metodumuz olmasa, `long` parametrli metod çağırılır, çünki daha geniş tipə malik parametrli metodu çağırmaq java üçün problem deyil, amma əksi mümkün deyil, çünki java daha kiçik tipə avtomatik çevirmə etmir.

```
class OverloadingTest {
   public void fly(int i) {
      System.out.print("int ");
   }
   public void fly(long l){
      System.out.print("long ");
   }
   public void fly(Object o){
      System.out.print("object ");
   }
   public static void main(String[] args) {
      OverloadingTest obj = new OverloadingTest();
      obj.fly(123);
      obj.fly(123L);
      obj.fly((short)123);
      obj.fly(123.0);
   }
}
```

Output:

```
int long int object
```

Main metodda sonuncu sətirdə `double` tipli parametr göndəririk. İlk öncə `double` tipi axtarılır, dəqiq uyğunluq tapılmadığına görə `double` autoboxing olaraq `Double` tipinə çevirilir. Və yenə dəqiq uyğunluq tapılmadığına görə `Object` parametrli metod çağırılır.

Putting It All Together

Overload metodların işləmə ardıcıllığı aşağıdakı cədvəldə göstərilib:

Rule	Example of what will be chosen for `glide(1,2)`
Exact match by type	`public String glide(int i, int j) {}`
Larger primitive type	`public String glide(long i, long j) {}`

Rule	Example of what will be chosen for `glide(1,2)`
Autoboxed type	`public String glide(Integer i, Integer j) {}`
Varargs	`public String glide(int... nums) {}`

```
class TestOverloading {
   public static void main(String[] args) {
      System.out.print(glide());
      System.out.print(glide("a"));
      System.out.print(glide("a", "b"));
      System.out.print(glide("a", "b", "c"));
   }
   public static String glide(String s){
      return "1";
   }
   public static String glide(String... s){
      return "2";
   }
   public static String glide(Object o){
      return "3";
   }
   public static String glide(String s, String t){
      return "4";
   }
}
```

Output:

```
2142
```

Varargs parametrli metoda boş parametr də göndərə bilərik, bu zaman java boş massiv göndərir.

Java ən dəqiq uyğunluq olan parametri axtaran zaman ***ancaq bir çevirmə (one conversion) edir.***

```
public class TooManyConversions {
// public static void play(Object o) {}
   public static void play(Long l) {}
   public static void play(Long... l) {}
   public static void main(String[] args) {
      play(4);     // DOES NOT COMPILE
      play(4L);    // calls the Long version
   }
}
```

Yuxarıdakı nümunədə problem çevirmə ilə bağlıdır. Java `int` 4`ü `long` 4 və ya `Integer` 4`ə rahatlıqla çevirir. Amma bir addıma `int`i `Long`a çevirə bilmir, gərək əvvəlcə `int`i `long`a çevirsin, sonra da `long`u `Long`a, yəni iki addıma. İki çevirmə də mümkün deyil. Amma əgər 1-ci metodu kommentdən çıxarsaq, kod kompayl olunacaq. Çünki bu zaman ancaq bir çevrilmə yerinə yetiriləcək: `int`dən `Integer`ə. `Integer` də obyekt olduğundan birinci metod çağırılacaq.

`int` parametr qəbul edən metod həm `char`, həm də `int` tipini çağıra bilir. Amma `char` tipində parametr qəbul edən metod `int` tipini çağıra bilmir. Aşağıdakı nümunəyə Enthuware testlərində rast gələ bilərsiniz:

```
class Noobs {
   public void m(int a) { System.out.println("In int "); }      // line1
   public void m(char c) { System.out.println("In char "); }    // line2
   public static void main(String[] args) {
      Noobs n = new Noobs();
      int a = 'a';
      char c = 6;
      n.m(a);      // line3
      n.m(c);      // line4
   }
}
```

Əgər line2 kommentə alınsa kod normal işləyəcək, amma line1 kommentə alınsa line3 kompayl xətası verəcək.

Enthuware sual bankından daha bir maraqlı nümunə ilə tanış olaq və səbəbini öyrənək:

```
class TestClass {
   public void method(Object o){
      System.out.println("Object Version");
   }
   public void method(java.io.FileNotFoundException s){
      System.out.println("java.io.FileNotFoundException Version");
   }
   public void method(java.io.IOException s){
      System.out.println("IOException Version");
   }
   public static void main(String args[]){
      TestClass tc = new TestClass();
      tc.method(null);
   }
}
```

Sualdan da aydın göründüyü kimi hər üç metod `null` parametrini qəbul edir, bəs maraqlıdır ki, kodu icra etsək hansı metod çalışacaq? Əvvəlki yazılarımızda da qeyd etdik ki, Java birinci

ən dəqiq uyğunluğu (the most specific) axtarır. Bildiyimiz kimi `FileNotFoundException` classı `IOException` classının subclassıdır, ona görə də bu nümunədə ***"the most specific class"*** `FileNotFoundException` classı hesab edilir.

Əgər ***"most specific class"*** sayı bir deyil iki olarsa, onda nə baş verəcək? Yuxarıdakı kod nümunəsini biraz fərqli şəkildə yazaq və nəticəyə baxaq:

```
public class TestClass {
   public void method(Object o){
      System.out.println("Object Version");
   }
   public void method(String s){
      System.out.println("String Version");
   }
   public void method(StringBuilder s){
      System.out.println("StringBuilder Version");
   }
   public static void main(String args[]){
      TestClass tc = new TestClass();
      tc.method(null);    // DOES NOT COMPILE
   }
}
```

Aşağıdakı kompayl xətası ilə qarşılaşacağıq:

reference to method is ambiguous
both method method(String) in TestClass and method method(StringBuilder) in TestClass match

Məhz bu səbəbdən kod sətrinə `System.out.println(null);` yazdıqda kompayl xətası verir, çünki `println()` overload metoddur və parametr olaraq `null` göndərdikdə birdən çox uyğunluq tapılır.

Mövzunu tam dərk etdiyinizə əmin olmaq üçün sonda Enthuware`dən daha bir kod nümunəsi qeyd edəcəm, çox maraqlı sualdır. Kod nümunəsini dəyişərək ağlınıza gələn bir neçə kombinasiyada yoxlaya bilərsiniz. Belə olduqda sizə qaranlıq qalan məqamları özünüz üçün aydınlatmış olacaqsınız.

```
class TestClass {

   void probe(int... x) { System.out.println("In ..."); }

   void probe(Integer x) { System.out.println("In Integer"); }

   void probe(long x) { System.out.println("In long"); }

   void probe(Long x) { System.out.println("In LONG"); }

   public static void main(String[] args){
```

```
        Integer a = 4; new TestClass().probe(a);
        int b = 4; new TestClass().probe(b);
    }
}
```

Creating Constructors

Konstruktor class adı ilə eyni olan və geri dönüş tipi (return type) olmayan xüsusi metoddur.

```
class Book {
    public book() {}          // line1, DOES NOT COMPILE
    public void Book() {}     // line2
    public Book() {}          // line3
    public Book(Book b) {}    // line4
}
```

- line1 – java case sensitive dildir və class adı eyni olmadığından bu konstruktor ola bilməz. Java bunu normal metod kimi görür və geri dönüş tipi buraxıldığından kompayl olunmur.
- line2 – normal metoddur, dönüş tipi olduğundan konstruktor ola bilməz.
- line3 – düzgün konstruktordur.
- line4 – düzgün konstruktordur, parametr olaraq özü ilə eyni olan tip qəbul edə bilər.

Konstruktor yeni obyekt yaratmaq üçün istifadə edilir. Bu proses *instantiation* adlanır, çünki classın instansı yaradılır. Konstruktoru çağırmaq üçün `new` açar sözündən istifadə edilir. Java `new` sözünü görən kimi yeni yaradılacaq obyekt üçün yaddaş sahəsi ayırır, həmçinin konstruktoru axtarır və onu çağırır.

Konstruktor həmçinin instance dəyişənlərə dəyər mənimsətmək üçün istifadə edilir. Bəzən ola bilər ki, instance dəyişən adı ilə konstruktorun qəbul etdiyi parametrin adı eyni olsun. Parametr lokal dəyişən hesab edildiyindən ilkin olaraq ona müraciət edilir. Əgər instance dəyişəni lokal dəyişəndən fərqləndirmək istəyiriksə, o zaman instance dəyişənin adının önündə `this` açar sözünü istifadə edirik:

```
class Book {
    private String author;
    private int page;
    private int weight;
    public Book(int page, int theWeight) {
        page = this.page;         // dəyər dəyişilmir, yaxşı hal deyil
        weight = theWeight;       // normaldır, çünki fərqli adlardır
        this.author = "Roel";     // normaldır, amma this artıqdır
        // this.page = page;      // ən çox istifadə edilən forma
```

```
    }
    public static void main(String[] args) {
        Book b = new Book(7, 2);
        System.out.printf("%d %d %s", b.page, b.weight, b.author);  // 0 2 Roel
    }
}
```

Konstruktorlara access modifier (`public`, `private`, `protected`) tətbiq edilə bilər. Amma `static`, `final`, `synchronized`, `native` və `abstract` açar sözləri konstruktorlarla istifadə edilə bilməz.

Default Constructor

Sizin kodlayıb kodlamamağınızdan asılı olmayaraq hər bir classın konstruktoru var. Əgər siz classda hər hansı bir konstruktor yaratmasanız Java sizin üçün parametrsiz konstruktor yaradacaq və bu konstruktor *default constructor* adlanır. Bəzən ona *default no-arguments constructor* da deyilir. Default konstruktorun parametr listi və gövdəsi boş olur.

Default konstruktor ancaq o halda təchiz (supply) edilir ki, cari classda heç bir konstruktor olmasın. Əgər hər hansı bir konstruktor olarsa, o zaman default konstruktor buraxılır (omit).

```
class Book1 {
}
class Book2 {
    public Book2() {}
}
class Book3 {
    public Book3(int i){}
}
class Book4 {
    private Book4(){}
}

class BuyBook {
    public static void main(String[] args) {
        Book1 b1 = new Book1();
        Book2 b2 = new Book2();
        Book3 b3 = new Book3(5);
        Book3 b33 = new Book3();    // line1, DOES NOT COMPILE
        Book4 b4 = new Book4();     // line2, DOES NOT COMPILE
    }
}
```

Ancaq `Book1` classının default konstruktoru var. line1 ona görə kompayl olunmur ki, `Book3` classının arqumentsiz konstruktoru yoxdur, manual olaraq parametrli konstruktor

yaradıldığından default no-argument konstruktor buraxılıb. line2 – konstruktor `private` olduğundan kompayl olunmur. Private konstruktor yaratmaqda məqsəd odur ki, kompilyator classı default konstruktorla təmin etməsin. Belə olduqda başqa classlar tərəfindən bu classın obyektinin yaradılmasının qarşısı alınır. Bu o zaman faydalı olur ki, ya class ancaq `static` metodlara sahib olur, ya da class yeni instanslarının yaradılmasına özü nəzarət etmək istəyir.

Default konstruktorun access modifier`i classın access modifier`i ilə eyni olur. Əgər class `public` olarsa, default konstruktor da `public` olacaq.

Overloading Constructors

Konstruktorların adı class adı ilə eyni olduğuna görə overload olması üçün ancaq parametr listi fərqli olmalıdır:

```
class Cars {
   String color;
   int speed;
   public Cars(int speed){
      this.speed = speed;
      color = "green";
   }
   public Cars(int speed, String color){
      this.speed = speed;
      this.color = color;
   }
}
```

Bu kodda azacıq təkrarlanma var, hər iki konstruktorda da demək olar ki, eyni işlər görülür. Bu təkrarı aradan qaldırmaq üçün biz bir parametrli konstruktorda dəyişiklik edə və onun daxilində digər konstruktoru çağıra bilərik:

```
public Cars(int speed){
   // Cars(speed, "green");      // DOES NOT COMPILE
   new Cars(speed, "green");     // Kompayl olunur, amma bizə lazım olan bu deyil
}
```

`new` yazmayanda kod kompayl olunmur, `new` yazanda isə artıq nəticə biz istədiyimiz kimi olmur. Bir parametrli konstruktor çağırılanda classın obyekti yaradılır və `color, speed` dəyişənlərinə default dəyərlər mənimsədilir. Sonra biz həmin konstruktorun daxilində ikiparametrli konstruktoru `new` ilə çağıranda artıq yeni bir obyekt yaradılır və set olunan dəyərlər birinci obyektə aid olmur və onlardan imtina edilir (ignore).

```
class CarsDetail {
   public static void main(String[] args) {
      Cars c = new Cars(240);
```

```
        System.out.println(c.speed + " " + c.color);    // 0 null
    }
}
```

Əgər eyni obyektin başqa konstruktorunu çağırmaq istəyiriksə, o zaman `this()` ifadəsindən istifadə edirik. Və bu zaman set olunan yeni dəyərlər həmin obyektin dəyişənlərinə mənimsədilir. `Cars` classının təkparametrli konstruktorunun daxilini aşağıdakı kimi dəyişsək onda nəticə biz istədiyimiz kimi olacaq:

```
public Cars(int speed){
    // System.out.println("this() must be in the first line");
    this(speed, "green");
    }

class CarsDetail {
    public static void main(String[] args) {
        Cars c = new Cars(240);
        System.out.println(c.speed + " " + c.color);    // 240 green
    }
}
```

`this()` ifadəsi həmişə mütləq birinci sətirdə olmalıdır, əgər `println` ifadəsini kommentdən çıxarsaq kod kompayl olunmayacaq.

Order of Initialization

Ardıcıllıq:

1. Superclass (əgər varsa);

2. Static variable declarations and static initializers (faylda yerləşdiyi ardıcıllıqla);

3. Instance variable declarations and instance initializers (faylda yerləşdiyi ardıcıllıqla);

4. Constructor.

```
public class LoadTest {
    public static void main(String[] args) {
        System.out.println("START");
        new Child();
        System.out.println("END");
    }
}

class Grandparent {
    static { System.out.println("static - grandparent"); }
    { System.out.println("instance - grandparent"); }
```

```
   public Grandparent() { System.out.println("constructor - grandparent"); }
}

class Parent extends Grandparent {
   { System.out.println("instance - parent"); }
   public Parent() { System.out.println("constructor - parent"); }
   static { System.out.println("static - parent"); }
}

class Child extends Parent {
   public Child() { System.out.println("constructor - child"); }
   static { System.out.println("static - child"); }
   { System.out.println("instance - child"); }
}
```

Output:

```
START
static - grandparent
static - parent
static - child
instance - grandparent
constructor - grandparent
instance - parent
constructor - parent
instance - child
constructor - child
END
```

Bu 4 qaydanın hamısı eyni zamanda ancaq o vaxt işləyir ki, classın obyekti yaradılmış olsun. Əgər `new` açar sözündən istifadə edərək classa müraciət edilmirsə, ancaq *1-ci* və *2-ci* bənd icra edilir. *3-cü* və *4-cü* bənd classın obyekti yaradılana qədər gözləməyə məcburdu, əks halda icra edilmir. Çünki classın obyekti yaradılanda konstruktor çağırılır və yalnız konstruktor çağırıldıqdan sonra instance dəyişən və bloklar initialize olunur (yaxud başladılır).

```
public class InitalizationOrder {
   private String name = "Orxan";
   { System.out.println(name); }
   private static int count = 2;
   static { System.out.println(count); }
   { count++; System.out.println(count); }
   public InitalizationOrder(){
      System.out.println("Constructor");
   }
   public static void main(String[] args) {
      System.out.println("Start main method");
      new InitalizationOrder();
```

```
    }
}
```

Output:

```
2
Start main method
Orxan
3
Constructor
```

Main metod `static` üzvlər (members) initialize olunduqdan sonra icra edilməyə başlayır. Aşağıdakı nümunə nisbətən daha qəlizdir:

```
public class YetMoreInitializationOrder {
    static { new YetMoreInitializationOrder(); }
    static { add(4); }
    static void add(int num){ System.out.print(num + " "); }
    YetMoreInitializationOrder(){ add(3); }
    static { add(5); }
    { add(1); }
    { add(2); }
    public static void main(String[] args) { }
}
```

Output:

```
 1 2 3 4 5
```

Bu mövzuda çaşdırıcı məqamlar çoxdur, ona görə də sizi çaşdıra biləcək bir neçə kod nümunəsinə baxacağıq. Aşağıdakı kod nümunəsi ilə başlayaq:

```
class B {
    static int i = 10;
    static { System.out.println("B Loaded"); }
}

public class A {
    static { System.out.println("A Loaded"); }
    public static void main(String[] args) {
        System.out.println("A should have been loaded");
        B b = null;  //line1
        System.out.println("B should not have been loaded");
        System.out.println(b.i);
    }
}
```

Output:

```
A Loaded
```

```
A should have been loaded
B should not have been loaded
B Loaded
10
```

Bu nümunədə bizim diqqətimizi çəkən line1 sətridir. Bu sətirdə `B` classı tipində bir dəyişən elan edilsə də `B` classı yüklənmir (not load). Çünki JVM görür ki, `B` classının hər hansı bir üzvünə müraciət yoxdur, ona görə də `B` classını hələ yükləməyə (load) ehtiyac olmadığını başa düşür. Amma sonuncu sətirdə `B` classına məxsus `i` dəyişəni çağırılır. Bu dəfə artıq `B` classı yüklənilir, çünki istifadə edilmədən öncə class mütləq yüklənilməlidir.

İndi qeyd edəcəyimiz nümunə bir növü yuxarıda qeyd etdiyimiz nümunənin məzmun olaraq davamıdır, yəni yuxarıdakı nümunəni əbəs yerə qeyd etməmişik. Amma bu nümunə daha çətindir, ilk baxışdan sadə görünməsinə baxmayaraq:

```
class Super { static String ID = "QBANK"; }

class Sub extends Super {
   static { System.out.print("In Sub"); }
}

class Test {
   public static void main(String[] args){
      System.out.println(Sub.ID);
   }
}
```

Nümunə Enthuware sual bankındandır və kodu icra etdikdə nə çap ediləcəyi soruşulur. Yəqin ki, çoxunuz fikirləşirsiniz ki, `"In Sub"` və `"QBANK"` hər ikisi çap ediləcək. Mən də belə fikirləşmişdim, amma bu düzgün cavab deyil. Kodu icra etdikdə sadəcə `"QBANK"` çap edilir. `"In Sub"` çap edilmir, səbəb isə budur: biz `Sub` classına məxsus hər hansı bir üzvə müraciət etməyənə qədər `Sub` classının static bloku icra edilmir. Bu nümunədə `ID` dəyişəni `Super` classına məxsusdur, onun `Sub` classı üzərindən çağırılmasına baxmayaraq `Sub` classının yüklənilməsinə ehtiyac yoxdur.

İndi isə Coderanch forumundan daha 2 sual nümunəsinə baxaq. Bu nümunələr də öz növbəsində həm çox maraqlıdır, həm də ki bir az qəribə. Birinci nümunəmizdən başlayaq:

```
class Egg {

   public static void main(String[] args) {
      Egg egg = new Egg();
      System.out.println("number: " + egg.number);
   }

   { number = 4; }          // line1
   private int number = 3;  // line2
}
```

Bu sualda da diqqətinizi çəkən yəqin line1 sətri olacaq. İlk baxışdan elə fikirləşirsən ki, bu sətir kompayl xətası verəcək, çünki `number` dəyişəninə elan olunduğu sətirdən bir öncəki sətirdə instance initializer blokda yeni dəyər mənimsədilib. Amma bu sətir xəta vermir, normal kompayl olunur. Əgər line2 sətirini kommentə atsaq, o zaman line1 xəta verəcək.

Bəs belə olan halda maraqlıdır kodu icra etdikdə `3` çap olunacaq yoxsa `4`? Cavab `3`-dür.

O zaman fikirləşə bilərik ki, yəqin line1 icra edilmir. Amma elə deyil, icra edilir. Çox qəribə olsa da `number` dəyişəninə əvvəlcə `4` dəyəri mənimsədilir, sonra isə `3` dəyəri ilə əvəz edilir (overridden with 3). Suala ətraflı aşağıdakı linkdən baxa bilərsiniz:

https://coderanch.com/t/658852/certification/Initialisation-order

İndi isə digər nümunəyə baxaq:

```
class A {
   private int iA = 1;
   public A() { print(); }
   public void print() { System.out.println("iA = " + iA); }
}
class B extends A {
   private int iB = 2;
   public B() {}
   public void print() { System.out.println("iB = " + iB); }
}
class TestAB {
   public static void main(String[] args) {
      new A();
      new B();
   }
}
```

Output:

```
iA = 1
iB = 0
```

Bu nümunədə də sizə qəribə gələn çox güman output`da `iB` dəyərinin `2` deyil `0` olmasıdır. `main` metodda `A` classının konstruktoru çağırıldıqda hər şey aydındır. `B` classının konstruktoru çağırıldıqda isə varislik özəlliyindən dolayı ilk öncə `A` classının konstruktoru və `print()` metodu çağırılır. `B` classı `print()` metodunu override etdiyindən `A` classının deyil, `B` classının `print()` metodu icra edilir. Bu zaman isə `iB` dəyişəninin dəyəri hələ `0` olur. Niyə? Çünki `B` classının instance dəyişənlərinə dəyərlər super classın konstruktoru icra edildikdən sonra mənimsədilir. Bu halda `A` classının konstruktorunun icrasi hələ bitmədiyindən `iB` dəyişəninə hələ dəyər mənimsədilməyib, ona görə də onun default dəyəri, yəni `0` çap edilir. Suala ətraflı aşağıdakı linkdən baxa bilərsiniz:

https://coderanch.com/t/653694/certification/instance-variable-creation-prior-object

Encapsulating Data

Enkapsulyasiya OOP-nin əsas prinsiplərindən biridir və classı arzuolunmaz davranışlardan qorumaq məqsədi ilə istifadə olunur. Classın dəyişənləri `private` olur və həmin dəyişənlərə ancaq classın daxilində yaradılmış public `getter/setter` metodları vasitəsilə müraciət etmək mümkün olur. Məqsəd həmin dəyişənlərə doğru olmayan dəyərlər mənimsədilməsinin qarşısını almaqdır.

```
public class Course {
   private int countStudents;
   public int getCountStudents() {    // also called accessor
      return countStudents;
   }
   public void setCountStudents(int countStudents) {  // also called mutator
      if(countStudents >= 0)    // mənfi say qəbul edilməsinin qarşısını almaq
         this.countStudents = countStudents;
   }
}
```

Enkapsulyasiya olunmuş classlar *JavaBeans* və həmin classın instance dəyişənləri isə *property* adlanır. İmtahanda JavaBeans ilə bağlı bilməli olduğumuz yeganə şey adlandırma qaydalarıdır (naming conventions). JavaBeans üçün "naming conventions" qaydaları:

Rule	Example
Properties are private.	`private int numEggs;`

Getter methods begin with "is" if the property is a boolean.	`public boolean isHappy() {`
	`  return happy;`
	`}`

Getter methods begin with "get" if the property is not a boolean.	`public int getNumEggs() {`
	`  return numEggs;`
	`}`

Setter methods begin with "set".	`public void setHappy(boolean happy) {`
	`  this.happy = happy;`
	`}`

The method name must have a prefix of set/get/is, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) {`
	`  numEggs = num;`
	`}`

Boolean dəyərlər ilə *getter* metodlarda prefix kimi adətən *"is"* istifadə olunur, amma *"get"* də istifadə edilə bilər, bununla bağlı məhdudiyyət yoxdur.

Nümunələrə baxaq:

```
3:  private boolean playing;
4:  private String name;
5:  public boolean isPlaying() { return playing; }
6:  public String name(){ return name; }
7:  public void updateName(String n) { name = n; }
8:  public void setname(String n) { name = n; }
```

Line 3, 4, 5 – adlandırma qaydalarına uyğundur (follow naming conventions);

Line 6, 7, 8 – adlandırma qaydalarına uyğun deyil (don't follow naming conventions).

Creating Immutable Classes

Classı kənar müdaxilələrdən qorumağın digər üsulu isə həmin classı immutable etməkdir. Classı immutable etmək üçün birinci addım *setter* metodları ləğv etməkdir. Bu zaman sual yarana bilər ki, bəs əgər classı çağıran adamın (caller) spesifik dəyər təyin etməyi hələ də bacarmasını istəsək nə etməliyik? Bu zaman parametrli konstruktorlar köməyimizə gəlir.

```
class ImmutableCourse {
   private int countStudents;
   public ImmutableCourse(int countStudents){
      this.countStudents = countStudents;
   }
   public int getCountStudents() {
      return countStudents;
   }
}
```

Burada setter metodun gördüyü işi konstruktor görür. Obyekt yaradıldıqdan sonra (after instantiation) isə dəyərlərin dəyişdirilməsi mümkün deyil. Amma mutable obyektlərlə diqqətli davranmaq lazımdı. Aşağıdakı nümunə ilə bunun səbəbini əsaslandırmağa çalışaq:

```
public class NotImmutable {
   private StringBuilder builder;
   public NotImmutable(StringBuilder b){
      builder = b;
   }
   public StringBuilder getBuilder(){
      return builder;
   }
}
```

```
class TestNotImmutable {
   public static void main(String[] args) {
      StringBuilder sb = new StringBuilder("kitab");
      NotImmutable n = new NotImmutable(sb);
      sb.append(" bilik");
      StringBuilder builder = n.getBuilder();
      builder.append(" mənbəyidir");
      System.out.println(n.getBuilder());    // kitab bilik mənbəyidir
   }
}
```

Gördüyünüz kimi biz ancaq "kitab" sözünü göndərdik, amma sonda nəticə dəyişdi və bu biz istəyən nəticə deyil! Problem ondadır ki, biz `builder` dəyişəninə yaratdığımız `StringBuilder` obyektinin referansını mənimsədirik və bununla da bu dəyişənlərin hər ikisi eyni `StringBuilder` obyektinə işarə edir, birini dəyişdikdə digəri də dəyişir. Yaxud `getBuilder()` metodundan `builder` referansının özünü geri qaytarırıq və bu referans vasitəsilə yenə də obyekti dəyişmək mümkündür. Həll yolu isə odur ki, biz bu obyektlərin kopyasını yaratmalıyıq:

```
class Immutable {
   private StringBuilder builder;
   public Immutable(StringBuilder b){
      builder = new StringBuilder(b);
   }
   public StringBuilder getBuilder(){
      return new StringBuilder(builder);
   }
}
```

Və yaxud da metodun geri dönüş tipini String olaraq dəyişmək olar.

```
public String getValue(){
   return builder.toString();
}
```

Writing Simple Lambdas

Bildiyimiz kimi java obyekt yönümlü dildir. Amma 8-ci versiyada gələn yeniliklərlə java funksional proqramlaşdırma imkanlarını da özünə əlavə edib. Bunun üçün lambda ifadələrindən (lambda expressions) istifadə olunur. Lambda ifadələrini biz anonim metod kimi də başa düşə bilərik. Yəni real metodlarda olduğu kimi lambda ifadələrinin də parametr listi və gövdəsi olur, yeganə fərqi odur ki, adı olmur. Biz lambda ifadəsini metoda parametr olaraq da göndərə bilərik. Lambda ifadələrinə qısaca *lambdas* deyilir.

OCA imtahanında ancaq sadə lambda ifadələri soruşulur, məqsəd lambda ilə bağlı əsas anlayışları və sintaksisi mənimsətməkdir. Əsas lambda sualları çox güman OCP imtahanında soruşulacaq.

Lambda Example

Lambda ifadələrinin nə qədər faydalı ola biləcəyini görmək üçün ənənəvi qayda ilə yazılmış bir nümunəyə baxaq və sonra həmin nümunədə müvafiq kod nümunələrini lambda ifadələri ilə əvəz edərək müqayisə aparaq. Görək lambda ifadələri nə dərəcədə faydalıdır.

```
1. public class Animal {
2.    private String name;
3.    private boolean canHop;
4.    private boolean canSwim;
5.
6.    public Animal(String speciesName, boolean hopper, boolean swimmer) {
7.       name = speciesName;
8.       canHop = hopper;
9.       canSwim = swimmer;
10.    }
11.
12.    public boolean canHop() { return canHop; }
13.    public boolean canSwim() { return canSwim; }
14.    public String toString() { return name; }    // P.S.
15. }
```

```
1.  public interface Checker {
2.     boolean test(Animal a);
3.  }
```

```
1.  public class CheckIfHopper implements Checker {
2.     @Override
3.     public boolean test(Animal a) {
4.        return a.canHop();
5.     }
6.  }
```

```
3.  public class TraditionalSearch {
4.
5.     public static void main(String[] args) {
6.        List<Animal> animals = new ArrayList<>();
7.        animals.add(new Animal("fish", false, true));
8.        animals.add(new Animal("kangaroo", true, false));
9.        animals.add(new Animal("rabbit", true, false));
```

```
10.       animals.add(new Animal("turtle", false, true));
11.       animals.add(new Animal("frog", true, true));
12.
13.       print(animals, new CheckIfHopper());
14.    }
15.
16.    private static void print(List<Animal> animals, Checker checker) {
17.       for (Animal animal: animals)
18.          if (checker.check(animal))
19.             System.out.println(animal);
20.    }
21. }
```

Output:

```
kangaroo
rabbit
frog
```

P.S. `toString()` metodu override olunduğuna görə biz `animal`ı print edəndə toString metodundan geri qaytardığımız dəyər print olunur.

Yuxarıdakı kodun gördüyü iş qısaca ondan ibarətdir ki, bizdə heyvanların listi var və biz bilmək istəyirik ki, bu heyvanlardan hansılar hoppanmağı bacarır. `print()` metodu bizim təyin etdiyimiz şərtə uyğun olaraq hoppanmağı bacaran heyvanların siyahısını bizə göstərir. Bəs biz əgər üzməyi bacaran heyvanların siyahısını görmək istəsək nə baş verəcək? Təəssüf ki, bunun üçün bizə başqa class lazım olacaq, `CheckIfSwims` classı yaratmalıyıq. Classı yaratdıqdan sonra isə nəticəni görmək üçün 13-cü sətirdən sonra yenidən `print()` metodunu çağırmalıyıq, bu dəfə isə parametr olaraq `new CheckIfSwims()` göndərməliyik, yəni yeni obyekt yaradılır. Başqa bir yoxlama üçün artıq əlavə olaraq 2 iş gördük. Bu əlavə işlərdən bizi lambdas xilas edir. Lambdas ilə yazdıqda `TraditionalSearch` classında sadəcə 13-cü sətir aşağıdakı sətirlə əvəz olunur:

```
13:  print(animals, a -> a.canHop());
```

və ya

```
13:  print(animals, (Animal a) -> { return a.canHop(); } );
```

Kodu lambdas ilə əvəz etdiyimizə görə artıq bizə `CheckIfHopper` classı lazım olmayacaq. Plyus əgər biz üzməyi bacaran heyvanları seçmək istəsək əlavə class yaratmağa ehtiyac qalmayacaq, bir sətirlə biz bunu edə bilərik:

```
print(animals, a -> a.canSwim());
```

və ya üzməyi bacarmayanlar:

```
print(animals, a -> ! a.canSwim());
```

Biz burada sadəcə 2 davranışı göstərdik, təsəvvür edin sizdən yalnız hoppanmaq və üzmək deyil, bir neçə xüsusiyyət üzrə filter etmək tələb olunur. Bu zaman lamdas-ın önəmi daha çox nəzərə çarpacaq.

Lambda Syntax

Yuxarıda yazdığımız lambdas nümunəyə baxaq:

```
a -> a.canHop()
```

Bu o deməkdir ki, bu anonim metod `Animal` class tipində parametr qəbul edir və geriyə `boolean` dəyər qaytarır (result of `a.canHop()`). Bütün bunları biz özümüz bilirik, çünki kodu biz yazmışıq, bəs java bunları necə bilir?

Lambda ifadəsi görəndə yada salınmalı olan ilk şey bir interfeysdir, çünki lambda ifadələri hər zaman bir interfeysə bağlı olaraq çalışır. Bunlar **functional interface** adlanır və bunların mütləq *bir abstrakt metodu* olmalıdır (interfeysdən öncə `@FunctionalInterface` annotasiyası yazılır, amma istəyə bağlıdır, yazılmaya da bilər). Abstrakt metodların sayı birdən çox olduqda kompayl xətası verir. Bəs lambda ifadəsi interfeys ilə necə əlaqələndirilir? İlk öncə bunu başa düşmək mənə də çətin idi, çünki praktikada mütəmadi istifadə etdiyimiz bir anlayış deyil, mahiyyətini dərindən dərk etmək zaman alır. Hər şeyi oxuyub başa düşürdüm, amma müəyyən bir müddət sonra yenidən bu mövzuya qayıtdıqda oxuduqlarımı yenidən unutduğumu görürdüm, səbəb - praktikada tətbiq etməmə. Bir az kodla pratik etdikdən sonra bunu bir üsulla yadda saxlamaq qərarına gəldim, ümid edirəm bu üsul sizin üçün də faydalı olar, baxaq.

Əvvəlki mövzuda da qeyd etdik ki, biz lambdas`a adsız metod kimi də baxa bilərik. Daha diqqətlə fikir versək görərik ki, bu əslində funksional interfeysdəki metodun override olunmuş formasıdır. Addım-addım gedək:

1) Qeyd etdik ki, lambda ifadəsi mütləq bir funksional interfeysə bağlı olur və biz parametr olaraq lambda ifadəsini hansı metoda göndəririksə, həmin metod da parametr olaraq funksional interfeysin referansını qəbul edir. Əvvəlki mövzudakı kod nümunəsinə fikir verək:

```
print(animals, a -> a.canHop());
```

Biz `print()` metoduna ikinci parametr olaraq lambda ifadəsi göndəririk. `print()` metodunun strukturuna baxsaq görərik ki, ikinci parametr olaraq `Checker` interfeysi tipində parametr qəbul edir. Burada `Checker` funksional interfeysdir.

```
private static void print(List<Animal> animals, Checker checker)
```

2) İndi isə `Checker` interfeysinin abstrakt metoduna fikir verməliyik. Qeyd etdik ki, funksional interfeys olduğu üçün ancaq və ancaq bir abstrakt metodu ola bilər. Aha, indi biz qeyd etdiyimiz məqama gəldik çıxdıq. Deməli bizim lambda ifadəsi ancaq bu abstrakt metoda

uyğun olmalıdır, başqa seçimimiz yoxdur. Başqa sözlə həmin abstrakt metodu parametr olaraq göndərilən lambda ifadəsinin içində override edir.

3) Bəs necə override edir? Gəlin abstrakt metodumuzu və lambda ifadəmizi alt-alta yazaq və müqayisə edək:

```
boolean test(Animal a);
(Animal a) -> { return a.canHop(); }
```

Abstrakt metodda hər şey aydındır, `Animal` classı tipində parametr qəbul edir və geriyə `boolean` dəyər qaytarır. Normal qaydada bu metodu aşağıdakı şəkildə override edə bilərik:

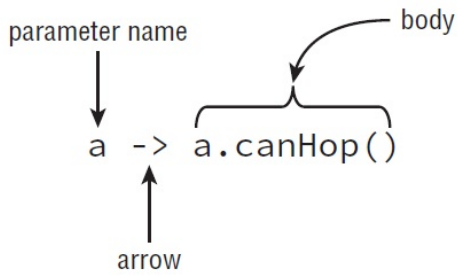
```
@Override
public boolean test(Animal a) {
   return a.canHop();
}
```

İndi keçək lambda ifadəmizə. Hər şey aydın olsun deyə əvvəlcə lambda ifadəsinin sintaksisinə baxaq. Bütün lambda ifadələri üç hissədən ibarət olur:

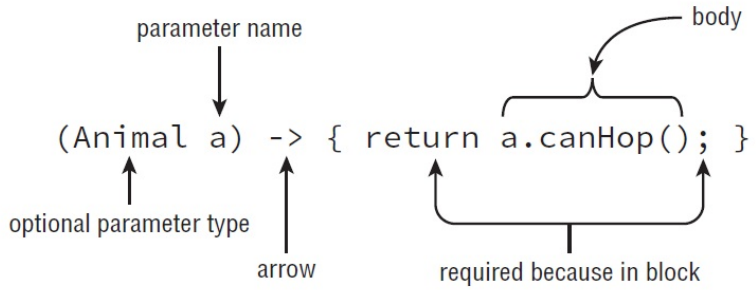
parametr listi, *ox (arrow)* və *gövdə (body)*.

Ox işarəsi parametr listi ilə gövdəni ayırır.

Şəkil 4.2 Lambda`nın sintaksisi (optional hissələr çıxarılmaqla)



Şəkil 4.3 Lambda`nın sintaksisi (optional hissələr daxil)



Bəzi hissələr optional`dı, tam formada da yazmaq olar, qısa formada da. İndi baxaq bizim nümunəni hansı formalarda yazmaq mümkündür:

```
a -> a.canHop()
(a) -> a.canHop()
(Animal a) -> a.canHop()
(Animal a) -> { return a.canHop(); }
```

Solda mötərizə ancaq o halda yazılmaya bilər ki, parametr sayı bir ədəd olsun və onun tipi aşkar şəkildə göstərilməsin. ***Sağda fiqurlu mötərizə ancaq o halda buraxıla bilər ki***, ifadələrin sayı ancaq bir ədəd olsun (only have a single statement). Əgər fiqurlu mötərizə yoxdursa, biz nə `return` ifadəsini, nə də nöqtəli-vergülü (;) yaza bilmərik. Və həmçinin nöqtəli-vergülü yazıb `return` ifadəsini yazmamaq da mümkün deyil, əgər bunlar yazılacaqsa ikisi də yazılmalıdır (burada bir **istisna** var, əgər funksional interfeysdəki metodun tipi `void` olarsa, o zaman nöqtəli-vergül yazılır, `return` yazılmır, `return` yazılarsa kompayl xətası verər). Qısaca əgər fiqurlu mötərizə varsa, bunlar da mütləq olmalıdır, fiqurlu mötərizə yoxdursa, bunlar da olmamalıdır. Əks halda kompayl xətası verir. Əgər ifadələrin sayı iki və daha çox olarsa fiqurlu mötərizə mütləq olmalıdır.

Valid lambdas:

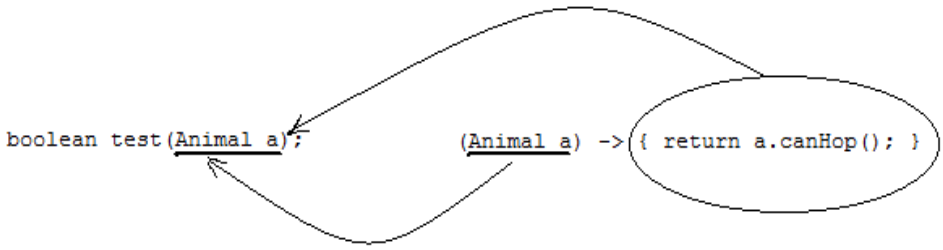
```
print( () -> true);                                    // 0 parameters
print( a -> a.startsWith("test"));                     // 1 parameter
print( (String a) -> a.startsWith("test"));            // 1 parameter
print( (a, b) -> a.startsWith("test"));                // 2 parameters
print( (String a, String b) -> a.startsWith("test"));  // 2 parameters
```

Invalid lambdas:

```
print( a, b -> a.startsWith("test"));            // needs parentheses
print( a -> { a.startsWith("test"); });          // missing return keyword
print( a -> { return a.startsWith("test") });    // missing semicolon
print( a -> { a.startsWith("test") });           // both missing
print( a -> return a.startsWith("test"); );      // there is no braces
```

Verilmiş nümunələrdə ancaq geriyə *boolean dəyər qaytaran lambdas nümunələri* göstərilib, çünki OCA imtahanı üçün bunu öyrənmək yetərlidir.

İndi qayıdaq abstrakt metod ilə lambda ifadəsinin əlaqəsinə.



Mövzunun əvvəlində axtardığımız "Bəs java bunları necə bilir?" sualının cavabı bu şəkildən aydın görünür. Deməli lambda ifadəsinin:

1) parametr listi abstrakt metodun parametr listi ilə eyni olmalıdır;
2) geri qaytardığı dəyər abstrakt metodun tipi ilə eyni olmalıdır.

Bu şəkili çəkib göstərməkdə məqsədim odur ki, lambda ifadəsinin gövdə hissəsini abstrakt metodun gövdə hissəsinə yerləşdirsək bir növü onu override etmiş olarıq. Ümid edirəm bu formada yanaşıldıqda daha rahat başa düşülər.

Aşağıdakı formada kod nümunələrinə imtahan suallarında tez-tez rast gələ bilərsiniz, bu zaman yadınıza salın ki, java eyni adlı lokal dəyişən elan etməyə icazə vermir:

```
(a, b) -> { int a = 0; return 5; }    // DOES NOT COMPILE
(a, b) -> { int c = 0; return 5; }
```

Bu nümunədə eyniadlı dəyişənlər eyni sətirdə olduğundan bunu tutmaq daha rahatdır, yəni diqqətimizi cəlb edir. Amma hərdən elə nümunələr ola bilər ki (Enthuware testlərində dəqiq biri var), dəyişən adı metod daxilində - ya parametr listində yaxud da daha öncəki sətirlərdə elan edilə bilər. **Və sonradan həmin dəyişən adı çaşdırmaq üçün lambda sintaksisində parametr listində yenidən istifadə edilə bilər.** Buna diqqət etmək lazımdır.

İmtahan mövzusuna daxil edilməsə də bilməkdə fayda var ki, lambda ifadələrində digər dəyişənlərdən də istifadə etməyə icazə verilir, lakin istisnalar var. *Static* və *instance* dəyişənləri ilə hər şey qaydasındadır. Amma metod parametrləri və lokal dəyişənləri ancaq o halda istifadə etmək mümkündür ki, onlara yeni dəyər mənimsədilməmiş olsun:

```
class Test {

   String s1 = "test";
   static String s2 = "test";

   public static void main(String[] args) {
      String s3, s4 = "test";
      s3 = "test";
      print(a -> a.startsWith(new Test().s1));
      print(a -> a.startsWith(s2));
      print(a -> a.startsWith(s3));
      s4 = "test";
```

```
        print(a -> a.startsWith(s4));     // DOES NOT COMPILE
    }

    private static void print(FunkInterface inf) {
        String s = "test";
        if (inf.test(s)) ;
    }

    interface FunkInterface {
        boolean test(String a);
    }
}
```

MyExamCloud test nümunələrindən birində maraqlı bir sualla rastlaşmışdım. Sualda soruşulurdu ki, aşağıdakı interfeyslərdən hansılar *functional interface* hesab edilə bilər:

```
1.   interface A<R> extends B {
2.      static void method(){
3.      }
4.   }
5.
6.   interface B<T> {
7.      public void print(T t);
8.
9.      static void print(){
10.     }
11.  }
12.
13.  interface C {
14.     void methodC(String s);
15.  }
16.
17.  interface D<T> extends A,B,C {
18.     default void printer(T t){
19.     }
20.  }
```

Mövzunun əvvəlində qeyd etmişdik ki, funksional interfeyslərin mütləq bir abstrakt metodu olmalıdır, birdən çox olduqda kompayl xətası verir. Amma bu qayda sırf abstrakt metodlara aiddir, interfeysin daxilində *default* və *static* metodlar ola bilər və bunlar interfeysin funksional olma özəlliyini pozmur.

Bu qaydalara istinadən birbaşa deyə bilərik ki, `B` və `C` interfeysləri funksional interfeyslərdir. `A` interfeysinin abstrakt metodu olmasa da göründüyü kimi `B` interfeysindən törəyir və `B` interfeysinin abstrakt `print(T t)` metodunu varis alır. Məhz bu səbəbdən `A` da funksional interfeys hesab edilir. `D` interfeysi sintaksis olaraq düzgün olsa da funksional interfeys hesab

edilə bilməz. Çünki həm `B`, həm də `C` interfeyslərinin abstrakt metodlarını varis alır və abstrakt metodlarının sayı birdən çox olduğu üçün funksional interfeys hesab edilmir.

Predicates

Yuxarıdakı nümunələrdə biz istədiyimiz şərtin doğruluğunu yoxlamaq üçün `Checker` funksional interfeysinin `test()` metodundan istifadə edirdik.

```
1:  public interface Checker {
2:     boolean test(Animal a);
3:  }
```

Burada biz parametr olaraq ancaq `Animal` tipində obyekt göndərə və şərtlərimizi ancaq `Animal` classı üçün yoxlaya bilirdik. Tutaq ki, indi bizim `Person` tipində classımız var və eyni şərtləri həmin class üçün də yoxlamalıyıq. Bunun üçün biz `Person` tipində obyekt qəbul edən yeni abstrakt metod yaratmalıyıq. Ancaq funksional interfeysdə birdən artıq abstrakt metod yaratmaq mümkün olmadığından məcburuq yeni funksional interfeys yaradaq. Bu problemi aradan qaldırmaq üçün java özü bizi hazır functional interfeyslər ilə təmin edir. Həmin funksional interfeyslər `java.util.function` paketində yerləşir və OCA imtahanı üçün bizə bunlardan yalnız `Predicate` interfeysini öyrənmək kifayət edir.

```
@FunctionalInterface                // optional
public interface Predicate<T> {
   boolean test(T t);
}
```

`Predicate` interfeysinin bizim `Checker` interfeysindən ancaq bir fərqi var, ***o da generics olmağıdır.*** İndi biz yeni metod yaratmadan `Predicate` interfeysinin `test()` metoduna `Animal`, `Person`, və yaxud hər hansı istənilən referans tip göndərə bilərik. Əgər `Predicate` interfeysinə hər hansı bir tip təyin edilməzsə, default olaraq `Object` qəbul edir.

İndi yuxarıda yazdığımız `TraditionalSearch` classını heç bir funksional interfeys yaratmadan `Predicate` interfeysi vasitəsilə yenidən yazaq:

```
import java.util.*;
import java.util.function.*;
public class PredicateSearch {
   public static void main(String[] args) {
      List<Animal> animals = new ArrayList<Animal>();
      animals.add(new Animal("fish", false, true));
      animals.add(new Animal("dog", true, true));

      print(animals, a -> a.canHop());
   }
   private static void print(List<Animal> animals, Predicate<Animal> checker){
```

```
        for (Animal animal: animals) {
            if (checker.test(animal))
                System.out.print(animal + " ");
        }
        System.out.println();
    }
}
```

Tam aydın olsun deyə daha bir nümunəyə baxaq. Bu nümunə https://dzone.com saytında rastıma çıxdı, lambdanın önəmini göstərmək baxımından daha faydalı gəldi mənə, ona görə qeyd etmək qərarına gəldim. Deməli sizin belə bir listiniz var:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
```

Və sizdən bu listin bütün elementlərinin cəmini verən metod yazmağınız tələb olunur. Aşağıdakı kimi bir metod yazırsınız:

```
public int sumAll(List<Integer> numbers) {
    int total = 0;
    for (int number: numbers) {
        total += number;
    }
    return total;
}
```

Səhəri gün listin cüt elementlərinin cəmi də tələb olunduğu deyilir. Və ağlınıza gələn ilk şey həmin metodu kopyalayıb, adını və içindəki şərti dəyişmək olur:

```
public int sumEven(List<Integer> numbers) {
    int total = 0;
    for (int number : numbers) {
        if (number % 2 == 0) {
            total += number;
        }
    }
    return total;
}
```

Növbəti gün isə listin 3-dən böyük olan elementlərinin cəmi lazımdır deyəndə artıq başa düşürsünüz ki, copy-paste üsulu bu işlər üçün uyğun deyil, sizə optimal bir yöntəm lazımdır. Bu zaman lambda bizim köməyimizə gəlir. Predicate interfeysindən istifadə edərək metodu bircə dəfə yazırıq:

```
public int sumAppropriateElements(List<Integer> numbers,Predicate<Integer> p){
    int total = 0;
    for (int number: numbers) {
        if (p.test(number)) {
            total += number;
```

```
        }
    }
    return total;
}
```

Və tələb olunan şərtə uyğun lambda ifadəsi göndərərək lazımi nəticəni alırıq:

```
sumAppropriateElements(numbers, n -> true);          // all
sumAppropriateElements(numbers, n -> n % 2 == 0);    // only even
sumAppropriateElements(numbers, n -> n > 3);         // greater than 3
```

İmtahanda `Predicate` ilə əlaqəli bilməli olduğumuz növbəti mövzu `removeIf()` metodudur. Bu metod `ArrayList` ilə istifadə edilir və parametr olaraq `Predicate` qəbul edir. Bu metodun üstünlüyü nədir baxaq. Tutaq ki, bizim adlar listimiz var və biz istəyirik ki, `C` hərfi ilə başlamayan adlar bu listdən silinsin. Ənənəvi qayda ilə yanaşsaq biz bu listi dövrə salıb hər elementi bir-bir şərtlə yoxlayıb silə bilərik. `removeIf()` metodunun köməkliyi ilə isə bunu bir sətirdə etmək mümkündür:

```
List<String> names = new ArrayList<>();
names.add("Elnur");
names.add("Ceyhun");
names.add("Cavid");
names.add("Adil");
names.add("Hasil");
names.add("Odər");
System.out.println(names);        // [Elnur, Ceyhun, Cavid, Adil, Hasil, Odər]
names.removeIf(s -> s.toLowerCase().charAt(0) != 'c');
System.out.println(names);        // [Ceyhun, Cavid]
```

`removeIf()` metodu ilə bağlı daha bir nümunəyə baxaq:

```
public class Whizlabs {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(21);
        list.add(13);
        list.add(30);
        list.add(11);
        list.add(2);
        // insert here
        System.out.println(list);
    }
}
```

Bu nümunə Whizlabs test bankında rastıma çıxmışdı və soruşulur ki, *"insert here"* yazısının əvəzinə aşağıdakı bəndlərdən hansı əlavə edilməlidir ki, ekrana `[21, 13, 11]` çap edilsin?

```
a. list.removeIf(e > e%2 != 0);
```

```
b. list.removeIf(e -> e%2 != 0);
c. list.removeIf(e -> e%2 == 0);
d. list.remove(e -> e%2 == 0);
e. Heç biri
```

Mövzunu yaxşı başa düşdüyünüzə əmin olmaq üçün cavabı özünüz tapmağa cəhd edin.

Enthuware-də rastlaşdığım suallardan biri də mənim üçün yenilik olmuşdu, kitabda həmin məsələyə rast gəlməmişdim. Kod nümunəsi və sual belə idi:

```
public class TestClass {
   public static boolean checkList(List list, Predicate<List> p) {
      return p.test(list);
   }
   public static void main(String[] args) {
      boolean b = //WRITE CODE HERE
      System.out.println(b);
   }
}
```

Qeyd olunan hissəyə aşağıdakı bəndlərdən hansılar əlavə edilərsə, ekrana `true` çap edilər?

```
a. checkList(new ArrayList(), al -> al.isEmpty());
b. checkList(new ArrayList(), ArrayList al -> al.isEmpty());
c. checkList(new ArrayList(), al -> return al.size() == 0);
d. checkList(new ArrayList(), al -> al.add("hello"));
e. checkList(new ArrayList(), (ArrayList al) -> al.isEmpty());
```

b və *c* bəndlərində aşkar sintaksis səhvi olduğundan düzgün cavablar deyil, qalır *a*, *d* və *e*. `add()` və `isEmpty()` metodları geri `true` qaytarır və sintaksis ilə hər şey qaydasında olduğuna görə *a* və *d* düzgün cavablardır. Geriyə bircə *e* variantı qalır və mənim də ən çox diqqətimi çəkən bənd bu olmuşdu. Bu bənddə də hər şey qaydasında görünürdü, fikirləşirdim yəqin bu da düzgün cavab ola bilər, amma elə deyil, *e* səhv cavabdır. Çünki `checkList` metodunda `Predicate`in tipi `List` göstərilib, *e* bəndində də `al` dəyişəninin tipi mütləq `List` olmalıdır, `ArrayList` ola bilməz. Buna diqqət etmək lazımdır.

`Predicate` interfeysi ilə birbaşa hazır filterlər də yaratmaq mümkündür. Məsələn, sizə belə bir filter lazımdı: sözün tərkibində kiçik *"a"* hərfi olsun, amma söz kiçik *"a"* ilə başlamasın. Bu filteri aşağıdakı şəkildə yaza bilərik:

```
Predicate<String> filter = s -> s.indexOf("a") > 0;
```

Sözün bu filterə uyğun gəlib-gəlmədiyini yoxlamaq üçün isə aşağıdakı kodu çağırmaq kifayətdir:

```
System.out.println(filter.test("java"));  // true
System.out.println(filter.test("asp"));   // false
```

Mövzunun sonunda isə öz imtahan təcrübəmdən bir məqamı vurğulamaq istəyirəm. Deməli, mən imtahan verən zaman 1Z0-808 (digər adı ilə OCA SE 8) imtahanında 77 sual olurdu. Və mənə cəmi-cümlətanı bircə lambda sualı düşmüşdü və həmin sualı da səhv cavablandırmışdım. İmtahandan çıxan kimi IDE-ni açıb birinci yoxladığım sual da elə bu sual olmuşdu və səhvin nədə olduğunu başa düşmüşdüm. Həqiqətən maraqlı sual idi, nə oxuduğum kitabda, nə də test banklarında bu tip sual ilə rastlaşmamışdım. Ola bilsin feedback`lər əsasında sonradan Enthuware test bankına əlavə etsinlər. Bu sual ilə bağlı ətraflı şəkildə imtahan təcrübəm bölməsində qeyd etmişəm.

Əlavələr

Nümunə 1:

Aşağıdakı kod nümunəsi icra edildikdə ekrana nə çap ediləcək?

```
class A {
   private int i = 6;
   private int j = i;
   public void A() {
      i = 5;
   }
   public static void main(String[] args) {
      A a = new A();
      System.out.println(a.i + a.j);
   }
}
```

Yəqin fikirləşirsiz ki, cavab `11`dir. Səbəb də odur ki, konstruktor çağırılarkən ilk öncə instance dəyişənlərə dəyərlər mənimsədilir, `i` və `j` hər ikisi `6` dəyərini alır. Daha sonra konstruktorun daxilində `i`-nin dəyəri dəyişdirilir və `5` olur. Və beləcə `(i+j)`-nin nəticəsi `11` olur.

Amma bunlar o zaman doğru olardı ki, `A()` metod yox konstruktor olsun. Koda diqqətlə baxsanız görərsiniz ki, `A()`-nın önündə `void` açar sözü var. Bildiyimiz kimi konstruktorlarda geri dönüş tipi olmamalıdır. Eyni zamanda Java class adı ilə eyni olan metod yaratmağa icazə verir. Bu səbəbdən kod nümunəsi heç bir xəta vermədən kompayl olunur və kodu icra etdikdə cavab `12` olur.

Nümunə 2:

```
class Egret {
   private String color;
   public void Egret() {
      Egret("white");
   }
   public void Egret(String color) {
      color = color;
   }
   public static void main(String[] args) {
      Egret e = new Egret();
      System.out.println("Color:" + e.color);
   }
}
```

Bu nümunə də öncəki nümunəyə bənzərdi, müəllif sualda metodu konstruktor kimi göstərib diqqəti yayındırmağa çalışıb.

Nümunə 3:

```
class Cardinal {
   static int number;
   Cardinal() {
      number++;
   }
   public static void main(String[] args) {
      Cardinal c1 = new Cardinal();      // line1
      if (c1 == null) {
         Cardinal c2 = new Cardinal();   // line2
      } else {
         Cardinal c2 = new Cardinal();   // line3
      }
      Cardinal c2 = new Cardinal();      // line4
      System.out.println(c1.number);
   }
}
```

Əgər `line4` sətrinin yerini dəyişib `line1` sətrindən sonra əlavə etsək, `line2` və `line3` sətirləri kompayl xətası verəcək. Hazırkı durumda isə kod heç bir xətasız kompayl edilir və icra etdikdə `line1`, `line3`, `line4` sətirlərində konstruktor çağırılır və output `3` olur. Maraqlı bir məqamı da qeyd edim; biz adətən `if` yaxud `else`-dən sonra fiqurlu mötərizəni yazmaya bilərik, əgər həmin blokun tərkibinə bir ifadə daxildirsə. Amma bu nümunədə fiqurlu mötərizələri yazmadıqda kompayl xətası alırıq.

Nümunə 4:

```
List numberList = Arrays.asList(5, 10, -5, -10);
Collections.sort(numberList);                          // [-10, -5, 5, 10]
int five = Collections.binarySearch(numberList, 5);  // 2
int four = Collections.binarySearch(numberList, 4);  // -3
System.out.println(five + four);   // -1
```

Nümunə 5:

Aşağıdakı kod nümunəsini icra etdikdə ekrana nə çap ediləcək?

```
class Panda {
   int age;
```

```
    public static void main(String[] args) {
       Panda p1 = new Panda();
       p1.age = 1;
       check(p1, p -> { p.age < 5 });
    }
    private static void check(Panda panda, Predicate<Panda> pred) {
       String result = pred.test(panda) ? "match" : "not match";
       System.out.print(result);
    }
}
```

A. `match`
B. `not match`
C. Compile error on line 8
D. Compile error on line 10
E. Compile error on line 11
F. A runtime exception is thrown

Nümunə 6:

Aşağıdakı kod nümunəsi icra edildikdə ekrana nə çap ediləcək?

```
public class PassByValueExample {

    public static void main(String[] args) {
        int x = 1, y = 2, z = 3;
        z = newValue(x, y, z);
        System.out.println(x + ":" + y + ":" + z);
        x = newValue(z, z, x);
        System.out.println(x + ":" + y + ":" + z);
        y = newValue(y, y, z);
        System.out.println(x + ":" + y + ":" + z);
    }

    public static int newValue(int z, int x, int y) {
        z--;
        x = 2 * y + z;
        y = x - 1;
        System.out.println(y + ":" + z);
        return x;
    }
}
```

“Pass by value” mövzusunu yaxşı mənimsəmək baxımından yuxarıdakı kod nümunəsi çox gözəl bir örnəkdir. Bu mövzu ilə yeni tanış olan bir çox proqramçı adətən çətinlik çəkir. Bu

sualın da qoyuluşu çaşdırmaq üçün qarışıq şəkildə verilib, maraqlı nüanslar var. Hissə-hissə araşdıraq və həll etməyə çalışaq.

Bildiyimiz kimi Java "pass-by-value" dildir, yəni bu kod nümunəsində metoda parametr kimi ötürülən primitiv dəyişənlərin özləri deyil, dəyərləri kopyalanıb göndərilir. Başqa sözlə desək, main metoddakı `x`, `y`, `z` dəyişənləri ilə `newValue()` metodundakı `x`, `y`, `z` dəyişənlərinin bir-biri ilə heç bir əlaqəsi yoxdur. Bir az daha açmış olsaq; tutaq ki, main metodda `z` dəyişəni `newValue()` metoduna 3-cü parametr kimi ötürülür. Amma `newValue()` metodunun parametr listində `z` dəyişəni birinci gəlir. Yəni, bu o demək deyil ki, main metoddakı `z` dəyişəninin dəyəri (3) `newValue()` metodundakı `z` dəyişəninə ötürüləcək. `newValue()` metodunun parametr listindəki dəyişənlərin adlandırılması şərtidir, bunu istədiyimiz kimi dəyişə bilərik (misal üçün `x`, `y`, `z` əvəzinə `a`, `b`, `c` də yaza bilərik, bu nəticəyə heç bir təsir etməyəcək). Ona görə də əsas fikir vermək lazımdır ki, main metodda `newValue()` metoduna dəyərlər hansı ardıcıllıqla ötürülürsə, `newValue()` metodunun parametr listində dayanan dəyişənlərə də həmin ardıcıllıqla mənimsədiləcək.

İkinci diqqət edəcəyimiz məsələ isə odur ki, `newValue()` metoduna `x`, `y`, `z` dəyişənlərinin kopyası göndərildiyinə görə `newValue()` metodunun gövdəsində baş verən istənilən hər hansı bir dəyişiklik main metoddakı `x`, `y`, `z` dəyişənlərinin dəyərinə heç bir təsir etməyəcək. Main metodda həmin dəyişənlərə ancaq yenidən dəyər mənimsətməklə dəyərlərini dəyişə bilərik. İndi isə addım-addım izahına baxaq:

Addım 1 → `z = newValue(1, 2, 3)`:

```
public static int newValue(int z, int x, int y) {   // [z=1, x=2, y=3]
    z--;            // 1 – növbəti sətirdə dəyəri azalacaq
    x = 2 * y + z;  // 2 * 3 + 0 = 6
    y = x - 1;      // 5
    System.out.println(y + ":" + z);  // 5:0
    return x;  // 6
}
```

`x` və `y` olduğu kimi qalır, `z` dəyişəninə isə `newValue(1, 2, 3)` metodundan qayıdan yeni dəyər mənimsədilir:

```
System.out.println(x + ":" + y + ":" + z);  // 1:2:6
```

Addım 2 → `x = newValue(6, 6, 1)`:

```
public static int newValue(int z, int x, int y) {   // [z=6, x=6, y=1]
    z--;            // 6 - növbəti sətirdə dəyəri azalacaq
    x = 2 * y + z;  // 2 * 1 + 5 = 7
    y = x - 1;      // 6
    System.out.println(y + ":" + z);   // 6:5
    return x;  // 7
}
```

y və z olduğu kimi qalır, x dəyişəninə isə `newValue(6, 6, 1)` metodundan qayıdan yeni dəyər mənimsədilir:

```
System.out.println(x + ":" + y + ":" + z);  // 7:2:6
```

Addım 3 → `y = newValue(2, 2, 6)`:

```
public static int newValue(int z, int x, int y) {   // [z=2, x=2, y=6]
    z--;            // 2 - növbəti sətirdə dəyəri azalacaq
    x = 2 * y + z;  // 2 * 6 + 1 = 13
    y = x - 1;      // 12
    System.out.println(y + ":" + z);   // 12:1
    return x;  // 13
}
```

x və z olduğu kimi qalır, y dəyişəninə isə `newValue(2, 2, 6)` metodundan qayıdan yeni dəyər mənimsədilir:

```
System.out.println(x + ":" + y + ":" + z);  // 7:13:6
```

Aldığımız bütün nəticələri print ifadələrinə uyğun olaraq ardıcıllıqla yazsaq nəticə belə olacaq:

```
5:0
1:2:6
6:5
7:2:6
12:1
7:13:6
```

Chapter 5. Class Design

Introducing Class Inheritance

Əgər bir class digər bir classdan törədilibsə, törədilən class "*child class*" və ya "*descendant*", törəməsi alınan class isə *"parent class"* və ya *"ancestor"* class adlanır. Əgər `X` classı `Y` classından törənibsə, `Y` də öz növbəsində `Z` classından törənibsə, `X` classı dolayı yolla `Z` classının da child classı hesab olunur.

Java təkvarisliliyi (single inheritance) dəstəkləyir, yəni bir class birbaşa olaraq ancaq bir parent classdan törənə bilər. Çoxvarisliliyi (multiple inheritance) dəstəkləmir, ancaq zəncirvari şəkildə bir neçə törəmə etmək mümkündür.

```
class Z { }
class Y extends Z { }
class X extends Y { }
```

Təkvarislilikdə qeyd etdik ki, bir classın birbaşa olaraq ancaq bir parent classı ola bilər. Lakin parent classın bir neçə child classı ola bilər.

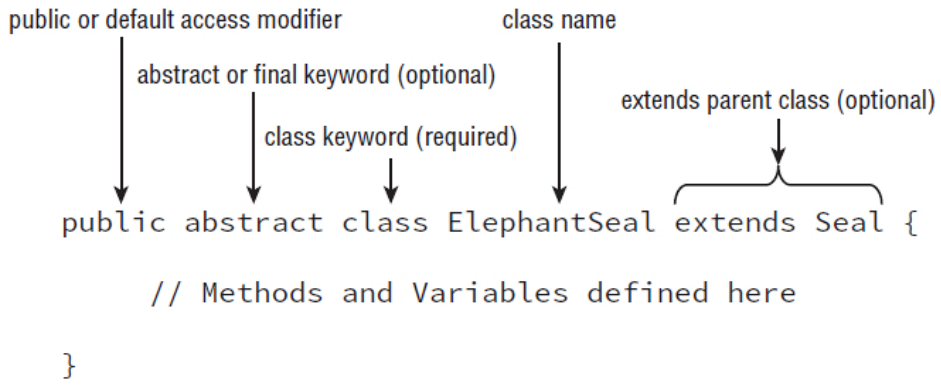
Əgər hər hansı bir classdan törəmə alınmasını istəmiriksə, o zaman həmin classı `final` edirik. Əgər final classdan törəmə almaq istəsək kompilyator xəta verəcək.

```
final class Z { }
class Y extends Z { }     // cannot inherit from final Z
```

Extending a Class

Classın törəməsini almaq üçün `extends` açar sözündən istifadə edilir.

Şəkil 5.1 Defining and extending a class



Animal.java:

```
class Animal {
   private int age;
   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }
}
```

Lion.java:

```
class Lion extends Animal {
   private void roar() {
      System.out.println("The " + getAge() + " year old lion says: Roar!");
      System.out.println("Lion is " + age + " year old");  // DOES NOT COMPILE
   }
}
```

Applying Class Access Modifiers

Top-level classlar ya `public` ola bilər ya da ki `default`, `protected` və `private` olmasına icazə verilmir. *Inner* classlar `protected` və `private` ola bilər, amma *inner* classlar OCA imtahanı mövzularına daxil deyil.

```
public  class Test1 { }
class Test2 { }
protected  class Test3 { }     // DOES NOT COMPILE
private  class Test4 { }       // DOES NOT COMPILE
```

Public classlar istənilən class tərəfindən istifadə oluna bilər. Lakin default classlara **ancaq eyni paketdə olan** subclasslar və ya classlar müraciət edə bilər.

Bir .java faylda maksimum bir `public` class mövcud ola bilər, amma istənilən sayda `public` olmayan class yaratmaq mümkündür. Ümumiyyətlə, *.java* faylda `public` class olmaya da bilər.

Yuxarıda qeyd olunan access modifiers ilə əlaqəli qaydalar interfeyslər üçün də keçərlidir. Top-level interfeyslər `public` və ya default ola bilər. Həmçinin java faylda maksimum bir `public` interfeys ola bilər.

Creating Java Objects

`java.lang.Object` yeganə classdır ki, heç bir parent classı yoxdur və javada bütün classlar bu classdan törəyib. Object classını xüsusi olaraq extends etməyə ehtiyac yoxdur, kompilyator avtomatik olaraq onu əlavə edir. Aşağıdakı classlar ekvivalentdir:

```
public class Book { }
public class Book extends java.lang.Object { }
```

Əgər class başqa bir classdan törəmirsə, dərhal həmin classın adının sonuna `extends java.lang.Object` ifadəsi əlavə edilir. Yox əgər başqa bir classdan törəyirsə, o zaman bu sintaksis əlavə edilmir, amma dolayı yolla həmin classın child`ı hesab olunur. Bir sözlə `java.lang.Object` hər zaman zəncirin ən yuxarısında dayanır (top of the tree).

Defining Constructors

Java`da hər konstruktorda birinci ifadə (first statement) class daxilində olan başqa bir konstruktoru çağırır (using `this()` ) və yaxud da birbaşa parent classın konstruktorunu çağırır (using `super()` ). Əgər parent konstruktor arqument qəbul edirsə, `super()` konstruktor da müvafiq olaraq arqument qəbul edəcək.

```
public class City {
   private int population;
   public City(int population) {
      super();    // line 1
      this.population = population;
   }
}
class Baku extends City {
   public Baku(int population) {
      super(population);   // line 2
   }
   public Baku() {
```

```
        this(2_181_800);  // line 3
    }
}
```

`line 1` - `java.lang.Object` classında təyin edilmiş parent konstruktoru çağırır.
`line 2` - `City` classının parametr qəbul edən konstruktorunu çağırır.
`line 3` - `super()` çağırılmır, əvəzində həmin classın digər konstruktoru çağırılır.

*`line 3`ə əsasən bu qənaətə gəlmək olar ki, əgər `this();` varsa çox güman `super()` ifadəsindən imtina edilir (ignore), çünki default olaraq tək `super()` çağırılsa bu kompayl olunmamalıdır, çünki parent classda default konstruktor buraxılır (omit).

`super()` də `this()` kimi konstruktorun ancaq birinci ifadəsi (first statement) olaraq çağırıla bilər, əks halda kompayl xətası verir.

Əgər parent classda birdən artıq konstruktor mövcuddursa, child class bu konstruktorlardan istənilən hər hansı birini istifadə edə bilər.

```
class Animal2 {
    private int age;
    private String name;
    public Animal2(int age, String name) {
        super();
        this.age = age;
        this.name = name;
    }
    public Animal2(int age) {
        super();
        this.age = age;
        this.name = null;
    }
}

class Gorilla extends Animal2 {
    public Gorilla(int age) {
        super(age, "Gorilla");
    }
    public Gorilla() {
        super(4);
    }
}
```

Child classın tək parametrli konstruktoru parent classın iki parametrli konstruktorunu, parametrsiz konstruktoru isə parent classın bir parametrli konstruktorunu çağırır. Yəni child konstruktor uyğun parent konstruktoru çağırmağı tələb etmir.

Understanding Compiler Enhancements

İndiyə kimi baxdığımız classların çoxunda parent konstruktor `super()` açar sözü ilə aşkar şəkildə çağırılmırdı, sual oluna bilər ki, belə olan halda bəs kod necə kompayl olunur? Çünki, əgər parent konstruktor aşkar şəkildə çağırılmayıbsa, java kompilyator avtomatik olaraq `super()` ifadəsini əlavə edir. Aşağıdakı classların üçü də bir-biri ilə ekvivalentdir:

```
public class Room {}

public class Room {
   public Room() {}
}

public class Room {
   public Room() {
      super();
   }
}
```

Bəs parent classın parametrsiz konstruktoru olmadıqda nə baş verəcək? Bu zaman java kompilyator bizə kömək etməyəcək, bu halda biz mütləq child classda ən azı bir konstruktor yaratmalıyıq və həmin konstruktor parent konstruktoru **`super()` ilə aşkar şəkildə çağırmalıdır.** Əks halda kompayl xətası verəcək:

```
class Mammal {
   public Mammal(int age) {
   }
}

class Elephant extends Mammal {   // DOES NOT  COMPILE
}
```

Yuxarıda qeyd etdiyimiz kimi child classda ən azı bir konstruktor yaratmalıyıq:

```
class Elephant extends Mammal {
   public Elephant() {   // DOES NOT COMPILE
   }
}
```

Kompilyator yaratdığımız bu child konstruktorun ilk sətrinə `super()` əlavə etməyə cəhd edir, amma görür ki, parent classda uyğun konstruktor yoxdur. Ona görə də biz ancaq parametrli `super()` ifadəsini aşkar şəkildə əlavə etməklə problemi aradan qaldıra bilərik:

```
class Elephant extends Mammal {
   public Elephant() {
      super(7);
   }
}
```

Enthuware sualları içərisində bu mövzu ilə bağlı maraqlı sual nümunələri var, onlardan birinə baxaq:

```
class A {
   int i;
   public A(int x) {
      this.i = x;
   }
}

class B extends A {
   int j;
   public B(int x, int y) {
      super(x);
      this.j = y;
   }
}
```

Sualda soruşulur ki, aşağıdakı konstruktorlardan hansını `B` classına əlavə etsək, kod xətasız kompayl olunar?

```
a) B() { }
b) B(int y) { j = y; }
c) B(int y) { super(y*2); j = y; }
d) B(int y) { j = y; j = y*2; }
e) B(int z) { this(z, z); }
```

Sualda iki düzgün cavab bəndi var. Az öncə yuxarda qeyd etdiyimiz qaydaları bu suala tətbiq etməklə düzgün cavabları rahatlıqla tapmaq mümkündür. Çalışın özünüz tapmağa cəhd edəsiniz.

Öz real imtahan təcrübəmdən deyə bilərəm ki, ən çox soruşulan mövzulardan biri bu mövzudur, ona görə də bu mövzuya xüsusi diqqət ayırmaq lazımdır.

Reviewing Constructor Rules

Yuxarıda dediyimiz qaydaları aşağıdakı şəkildə ümumiləşdirə bilərik:

1. Hər konstruktorun ilk ifadəsi ya `this()` açar sözündən istifadə edərək class daxilindəki digər konstruktoru çağırır, ya da `super()` açar sözündən istifadə edərək birbaşa parent classın konstruktorunu çağırır.

2. `super()` və `this()` ancaq konstruktorun birinci ifadəsi (first statement) olmalıdır, əks halda xəta verir.

3. Əgər `super()` konstruktorda aşkar şəkildə əlavə edilməzsə, Java konstruktorun ilk sətrinə arqumentsiz `super()` ifadəsini əlavə edəcək.

4. Əgər parent classın parametrsiz konstruktoru yoxdursa və child classda hər hansı bir konstruktor yaradılmayıbsa, kompilyator xəta verəcək və arqumentsiz default konstruktoru (default no-argument constructor) child classa əlavə etməyə cəhd edəcək.

5. Əgər parent classın parametrsiz konstruktoru yoxdursa, kompilyator hər child konstruktorda parent konstruktoru aşkar şəkildə çağırmağı tələb edir.

Calling Constructors

Java`da parent konstruktor həmişə child konstruktordan əvvəl icra olunur.

```
class Continent {
   public Continent() {
      System.out.println("Continent");
   }
}

class Country extends Continent {
   public Country() {
      System.out.println("Country");
   }
}

class City extends Country {
   public static void main(String[] args) {
      new City();
   }
}
```

Kompilyator əvvəlcə `Continent` və `Country` konstruktorlarının birinci sətrinə `super()` əlavə edir. Sonra `City` classına default no-argument konstruktor əlavə edir və birinci sətrinə də `super()`. Bu səbəbdən də parent konstruktorlar ilk öncə icra olunur. Output:

```
Continent
Country
```

Calling Inherited Class Members

Child classlar parent classın istənilən `public` və `protected` üzvlərini (members) istifadə edə bilər. Əgər parent və child class eyni paketdə olarsa, o zaman parent classın default üzvlərini də istifadə edə bilər. Amma parent classın `private` üzvlərinə heç vaxt birbaşa müraciət edə bilməz. `private` üzvlərə `public` və ya `protected` metodlar vasitəsilə dollayı yolla müraciət etmək olar.

```
class Fish {
   protected int size;
   private int age;

   public Fish(int age) {
      this.age = age;
   }

   public int getAge() {
      return age;
   }
}

class Shark extends Fish {
   private int number = 8;

   public Shark(int age) {
      super(age);
      this.size = 7;
   }

   public void displaySharkDetails() {
      System.out.println("Shark age: "    + getAge());   // 1
      System.out.println("Shark size: "   + size);       // 7
      System.out.println("Shark number: " + number);     // 8
   }

   public static void main(String[] args) {
      new Shark(1).displaySharkDetails();
   }
}
```

Child classda biz parent classın `public` metodunu (`getAge()`) `və protected` dəyişənini (`size`) birbaşa istifadə edə bilərik. Ümumiyyətlə, varisliyə görə child class parent classın bütün üzvlərinə sahiblənir, yəni miras alır (access modifier`in icazə verdiyi üzvlər). Buna görə də biz child classda `this` açar sözündən istifadə edərək həm child classın, həm də parent classın üzvlərini çağıra bilərik. Yuxarıdakı metod ilə bu metod mahiyyət baxımından eynidir:

```
public void displaySharkDetails() {
   System.out.println("Shark age: "    + this.getAge());   // 1
```

```
      System.out.println("Shark size: "   + this.size);       // 7
      System.out.println("Shark number: " + this.number);     // 8
   }
```

Biz parent classın üzvlərini `super` açar sözünü istifadə edərək də çağıra bilərik, amma `super` vasitəsilə cari classın üzvlərini çağıra bilmərik. Başqa sözlə `this` və `super` ilə həm parent class, həm də child classın üzvlərini çağıra bilərik, amma cari classın üzvlərini ancaq `this` ilə çağırmaq mümkündür, əks halda kompayl xətası verəcək:

```
   public void displaySharkDetails() {
      System.out.println("Shark age: "    + super.getAge());  // 1
      System.out.println("Shark size: "   + super.size);      // 7
      System.out.println("Shark number: " + this.number);     // 8
      System.out.println("Shark number: " + super.number);    // DOES NOT COMPILE
   }
```

MyExamCloud test suallarının içərisində təxminən belə bir nümunə ilə rastlaşmışdım, output`un nə olacağı soruşulurdu:

```
1.  package packageA;
2.
3.  public class A {
4.     protected int y = 10;
5.  }
```

```
1.  package packageB;
2.
3.  import packageA.A;
4.
5.  public class B extends A {
6.     int y = 5;
7.     public void print() {
8.        A a = new A();
9.        System.out.println(a.y + y);
10.    }
11. }
```

İlk baxışdan sadə görünür. Yuxarıda da qeyd etdik ki, child class parent classın `protected` üzvlərini istifadə edə bilər. Bu qaydaya istinad edərək suala yanaşdıqda ağlımıza gələn ilk cavab `15`-dir. Amma sualda çox incə bir nüans var, bəlkə də bu sertifikat sualları içərisində ən çaşdırıcı məqamlardan biridir. Əslində kod nümunəsi kompayl olunmur və səbəb `protected` açar sözünün istifadəsi ilə bağlıdır. *Parent classdakı protected üzvlərə child classda ancaq child classın referansı vasitəsilə müraciət oluna bilər, parent classın öz referansı vasitəsilə müraciət edilə bilməz.* Yuxarıdakı kod nümunəsində gördüyünüz kimi `B` classında `A` classının `y` dəyişəninə `A` classının öz referansı vasitəsilə müraciət edilir (`a.y`), bu səbəbdən də bu sətir kompayl xətası verir. Əgər `B` classının referansı vasitəsilə müraciət edilsə, xəta verməyəcək.

Amma bu nümunədə həm `A`, həm də `B` classında istifadə edilən dəyişən adı eyni olduğundan burada `y` dəyişənini `B` classının referansı vasitəsilə çağırsaq, `B` classına məxsus `y` dəyişəninə müraciət ediləcək.

Yuxarıdakı kod nümunəsi üzərində biraz dəyişiklik edib, `protected` üzvlərə necə müraciət edə bilərik baxaq:

```
1.  package packageA;
2.
3.  public class A {
4.     protected int k = 10;
5.     protected int n = 15;
6.  }
```

```
1.  package packageB;
2.
3.  import packageA.A;
4.
5.  public class B extends A {
6.     int k = 5;
7.
8.     public void print() {
9.        A a = new A();
10.       B b = new B();
11.       System.out.println(k + b.k);      // 10
12.       System.out.println(k + super.k);  // 15
13.       System.out.println(k + b.n);      // 20
14.       System.out.println(k + a.n);      // DOES NOT COMPILE
15.    }
16.
17.    public static void main(String[] args) {
18.       B b = new B();
19.       b.print();
20.    }
21. }
```

super() vs super

`this` və `this()` ifadələrinin bir-biri ilə əlaqəsi olmadığı kimi, `super` və `super()` ifadələri də bir-birindən tamamilə fərqlənir.

Düzgün istifadə qaydası:

```
public Book(int age) {   // constructor
   super();
```

```
    super.setAge (10);
}
```

Səhv istifadə qaydası:

```
public Book(int age) {
    super;                  // DOES NOT COMPILE
    super().setAge (10);    // DOES NOT COMPILE
}
```

Aşağıdakı kod nümunəsini Enthuware testlərindən götürmüşəm, amma kodun daxilini biraz dəyişmişəm, daha maraqlı ola biləcək bir fakt əlavə etmişəm:

```
class Doll {
    String name;
    Doll(String nm) {
        this.name = nm;
    }
}

class Barbie extends Doll {
    // String name;    // line1

    Barbie(){
        this(name);     // DOES NOT COMPILE
    }
    Barbie(String nm){
        super(name);    // DOES NOT COMPILE
    }
}

public class Test {
    public static void main(String[] args) {
        Barbie b = new Barbie("mydoll");
    }
}
```

Bu sual həm də "Order of Initialization" mövzusu ilə əlaqəlidir, kodu kompayl etməyə cəhd etsək görəcəyik ki, kompilyator bizə *"cannot reference name before supertype constructor has been called"* xətasını göstərəcək. Əgər line1-i kommentdən çıxarsaq kod yenə kompayl olunmayacaq, çünki super konstruktor çağırılmamış `name` dəyişəninə dəyər mənimsədilmir.

Overriding a Method

Metodu override etmək – child classda parent classdakı metodla eyni signature (ad və parametr siyahısı) və geri dönüş tipində olan yeni metod yaratmaqdır. Metodu override

etdikdən sonra artıq `super` və `this` açar sözlərindən istifadə edərək metodun cari yaxud parent versiyasından istifadə etmək istədiyimizi təyin edə bilərik:

```
class Dog {
   public int getAge() {
      return 8;
   }
}

class Husky extends Dog {
   public int getAge() {
      return super.getAge() + 3;  // əgər super`i silsək, sonsuz dövrə düşəcək
   }
   public static void main(String[] args) {
      System.out.println(new Dog().getAge());
      System.out.println(new Husky().getAge());
   }
}
```

Output:

```
8
11
```

Düzgün override olunmuş metod aşağıdakı qaydalara riayət etməlidir:

1. Child classdakı metod parent classdakı metod ilə eyni quruluşa (the same signature) malik olmalıdır. Yəni, metod adları və parametr siyahısı mütləq eyni olmalıdır.

2. Child classdakı metodun access modifier`i parent classdakı metodla ən azı eyni olmalıdır, ya da ki, daha yüksək.

3. Parent classdakı metodun fırlatdığı (throws) exceptionun tipi həmişə child classdakı metoddan daha geniş olmalıdır. Child classdakı metod maksimum halda eyni exception tipini fırlada, yaxud da kiçik tip, və yaxud da heç exception fırlatmaya bilər. Amma burada söhbət checked exceptionlardan gedir, runtime exceptionlara bu məhdudiyyət tətbiq olunmur.

4. Hər iki metodun geriyə qaytardığı dəyər eyni tipdə olmalıdır və ya override edilən/parent classdakı metodun subclassı (*covariant return types*).

** private metodlar override olunmur.*
*** Bu qaydalara nümunələr üzərində baxdıqda daha aydın olacaq.*

Bu 4 qayda ilə bağlı nümunələrə keçməzdən öncə `private` metodların override edilməməsi məsələsinə bir az aydınlıq gətirək. Bəli, parent classdakı `private` metodlar override olunmur, ***hiding*** olur. Yəni əgər həmin metod parent classda çağırılırsa, parent classdakı metod işləyir, child classda çağırılırsa, child classdakı metod. Nümunə üzərindən baxaq:

```
public abstract class Bird {
   private void fly() { System.out.println("Bird is flying"); }
   public static void main(String[] args) {
      Bird bird = new Pelican();
      bird.fly();
   }
}

class Pelican extends Bird {
   protected void fly() { System.out.println("Pelican is flying"); }
}
```

Nümunənin override ilə bağlı izahına keçməzdən əvvəl kodla bağlı bir maraqlı məqamı qeyd edim ki, əgər biz `Pelican` classını icra etsək həmin classın daxilində `main` metod olmamasına baxmayaraq class icra ediləcək. Səbəb isə varisliklə bağlıdır. Əgər `Pelican`ın superclassının daxilində `main` metod varsa, biz `Pelican` classını icra etdikdə o birbaşa gedib `Bird` classının daxilindəki `main` metodu çağırır.

İndi qayıdaq nümunəmizin override ilə bağlı hissəsinə. Hər iki classda `fly()` metodu var və override olunmanın demək olar ki, bütün qaydalarına riayət edilir. Amma burada az öncə qeyd etdiyimiz istisna var, parent classdakı metod `private`dır, ona görə də bu override hesab olunmur, hiding hesab olunur. Kodu icra etsək nəticə belə olacaqdır:

```
Bird is flying
```

Çünki `fly()` metodu parent classda çağırılır. Biz əgər `Pelican` classındakı `fly()` metodunu kommentə salsaq kod yenə kompayl olunacaq və icra etdikdə eyni nəticəni alacağıq. Amma `Bird` classındakı `fly()` metodunu kommentə salsaq kod kompayl olunmayacaqdır. Çünki `bird` referansı `Bird` classına aiddir və o kompayl vaxtı `Bird` classında belə bir metodun olmadığını aşkarlayır və xəta verir. Əgər `Bird bird = new Pelican();` əvəzinə `Pelican bird = new Pelican();` yazsaq kod kompayl olunacaq və nəticə `Pelican is flying` olacaqdır.

Əgər biz `main` metodunu `Bird` classından çıxarıb `Pelican` classında yazsaq, kod yenə kompayl olunmayacaqdır. Çünki `Bird` classındakı `fly()` metodu `private`dır və subclass onu görə bilmir:

```
public abstract class Bird {
   private void fly() { System.out.println("Bird is flying"); }
}

class Pelican extends Bird {
   protected void fly() { System.out.println("Pelican is flying"); }
```

```
   public static void main(String[] args) {
      Bird bird = new Pelican();
      bird.fly();    // DOES NOT COMPILE
   }
}
```

4-cü bənddə ***"covariant return types"*** ifadəsi işlətdik, bunu biraz daha geniş izah edək. Deməli, bu xüsusiyyət Java`ya 1.5-ci versiyadan sonra gəlib. Anlamı isə odur ki, override edən (*overriding*) metodun return tipi override edilən (*overridden*) metodun return tipinin subclassı ola bilər. Yəni, əgər super classdakı metodun return tipi `Number` olarsa, subclassdakı metodun return tipi `Integer` (yaxud `Number` classından törənmiş hər hansı bir başqa class) ola bilər.

Amma bu qayda primitiv tiplər üçün keçərli deyil. Misal üçün, əgər override edən (overriding) metodun return tipi `int` olarsa, override edilən (overridden) metodun return tipi də mütləq `int` olmalıdır. `short`, `long` və yaxud `Integer` ola bilməz:

```
class Super {
   public Number getNumber() {
      return 5;
   }
   public int getInt() {
      return 6;
   }
}

class CovariantReturnTypes extends Super {
   public Integer getNumber(){  // OK
      return 7;
   }
   public short getInt(){       // DOES NOT COMPILE
      return 8;
   }
}
```

Overriding ilə overloading bir-birinə çox bənzəyir, çünki hər ikisində də metod adları eynidir. Amma overriding`dən fərqli olaraq overloading həmişə fərqli metod signature istifadə edir.

```
class Bird {
   public void fly() {
      System.out.println("Bird is flying");
   }
   public int eat(int food) {
      System.out.println("Bird is eating " + food + " units of food");
      return food;
   }
}
```

```
public class Eagle extends Bird {
   public int fly(int height) {  // Everything is ok, because of overloading
      System.out.println("Bird is flying at " + height + " meters");
      return height;
   }
   public void eat(int food) {   // DOES NOT COMPILE, because of overriding,
                                 // return type is different
      System.out.println("Bird is eating " + food + " units of food");
   }
}
```

Əgər imtahanda child və parent classda eyni adlı metod görsəniz, ilk növbədə onun **override** yaxud **overload** metod olduğunu təyin etmək, sualı tez tapmaq baxımından daha faydalı olacaqdır.

İndi isə qeyd etdiyimiz 4 qayda ilə bağlı nümunələrə baxaq:

Nümunə 1:

```
public class Camel {
   protected String getNumberOfHumps() {
      return "Undefined";
   }
}

class BactrianCamel extends Camel {
   private int getNumberOfHumps() {   // DOES NOT COMPILE
      return 2;
   }
}
```

2-ci və **4-cü** qaydalar pozulur burada. Child classdakı metodun access modifier`i ancaq `protected` və `public` ola bilər. Geri dönüş tipi isə birində `String`dir, digərində `int`.

Nümunə 2:

```
class MyException extends Exception {}

class Reptile {
   protected boolean hasLegs() throws MyException {
      throw new MyException();
   }
   protected double getWeight() throws Exception {
      return 2;
   }
}
```

```
class Snake extends Reptile {
   protected boolean hasLegs() {
      return false;
   }
   protected double getWeight() throws MyException {
      return 2;
   }
}
```

`Reptile` classındakı hər iki metod `Snake` classında override edilib və bütün qaydalara riayət edilib, xüsusilə də 3-cü qaydanın tələbləri pozulmayıb. Əgər nümunə aşağıdakı kimi olsa idi, o zaman qaydalar pozulmuş olardı:

```
class MyException extends Exception {}

class Reptile {
   protected double getHeight() throws MyException {
      return 2;
   }
   protected int getLength() {
      return 5;
   }
}

class Snake extends Reptile {
   protected double getHeight() throws Exception {  // DOES NOT COMPILE
      return 2;
   }
   protected int getLength() throws MyException {   // DOES NOT COMPILE
      return 5;
   }
}
```

İndi isə daha qarışıq sual nümunələrinə baxaq. Aşağıda qeyd edəcəyim nümunə Enthuware testləri içərisində ən maraqlı və ən yaxşı suallardan biridir. Buna bənzər bir nümunəni "Order of Initialization" mövzusunda qeyd etmişəm, bu suala isə bu məqalədə ayrıca baxaq:

```
class A {
   A() { print(); }
   void print() { System.out.println("A"); }
}
class B extends A {
   int i = 4;
   public static void main(String[] args){
      A a = new B();
      a.print();
   }
```

```
    void print() { System.out.println(i); }
}
```

`B` classı icra edildikdə ekrana nə çap ediləcəyi soruşulur və cavab variantları belədir:

```
a) It will print A, 4
b) It will print A, A
c) It will print 0, 4
d) It will print 4, 4
e) None of the above.
```

Cavab variantları da bir-birinə çox yaxındır, ona görə də sual maksimum həssaslıq və diqqət istəyir. O vaxtkı qeydlərimə baxıram, Standart Test`ləri edərkən özüm də bu sualı səhv cavablandırmışam. Mənim seçdiyim cavab *a* variantı idi, mən hesab edirdim ki, `A` classının konstruktoru çağırılarkən icra edilən `print()` metodu `A` classına məxsus `print()` metodu olacaq, amma override özəlliyindən dolayı `B` classının `print()` metodu icra edilir, buna diqqət etmək lazımdır.

Artıq bildik ki, `A` classının `print()` metodu çağırılmır, bu səbəbdən də *a* və *b* variantlarını çıxara bilərik, geriyə qalır 3 variant. Kömək üçün qeyd edim ki, *e* variantını da çıxara bilərik, cavablar arasında düzgün variant var. Geriyə qalır *c* və *d* variantları. Amma sualın ən maraqlı hissəsi hələ qabaqdadır, işi bitmiş hesab etmək olmaz.

Yəqin ki, çoxumuz *c* və *d* variantları arasında *d*-ni seçərdik. Amma *d* düzgün cavab deyil, düzgün cavab variantı *c*-dir. Maraqlı bir sual ortaya çıxır. Niyə?

Sualın bu hissəsi *"order of initialization"* ilə bağlıdır. Deməli sualda bütün hadisələr aşağıdakı ardıcıllıqla baş verir:

1. Main metodda ilk sətirdə `A` classının referansı yaradılır və bu zaman `B` classının konstruktoru çağırılır (bu cür yazılış polimorfizm özəlliyindən dolayı düzgündür, növbəti məqalələrdə bu mövzuya toxunacağıq);
2. `B` classının super classı varsa, ilk növbədə super classın konstruktoru çağırılmalıdır və bu nümunədə `A` classının konstruktoru çağırılır;
3. `A` classının konstruktorunda `print()` metodu çağırılır. `print()` metodu `B` classında override edildiyindən, `A` classındakı `print()` metodu deyil, `B` classındakı `print()` metodu icra edilir;
4. Bu addımda `i` dəyişəni çap edilir və işin ən maraqlı hissələrindən biri baş verir. `i` dəyişəni `B` classının instance dəyişənidir və instance dəyişənlərə dəyərlər super konstruktor icra edildikdən sonra mənimsədilir. Bu mərhələdə isə `print()` metodu super konstruktorun gövdəsində çağırılıb və bu səbəbdən super konstruktorun icrası hələ bitməyib. Məhz buna görə `i` dəyişəninə hələ dəyər mənimsədilməyib, ona görə də `i` dəyişəninin default dəyəri (*sıfır*) çap edilir;

5. Super konstruktorun icrası bitir və `i` dəyişəninə `4` dəyəri mənimsədilir;
6. Main metodda `a` referansı üzərindən `print()` metodu çağırılır. `a` referansı `A` classına aid olsa da "actual object" `B` classına məxsusdur. Ona görə də icra vaxtı `B` classında `print()` metodunun override edilib edilməməsi yoxlanılır. `B` classında `print()` metodu override edilib, bu səbəbdən də həmin metod çağırılır və `i`nin dəyəri (`4`) çap edilir.

Başqa bir nümunəyə baxaq. Tutaq ki, bizim 3 classımız var (`A`, `B`, `C`) və bu classlar zəncirvari şəkildə bir-birilərindən törəyiblər; `C` classı `B` classından, `B` classı da öz növbəsində `A` classından:

```
class A {
   public void print() {
      System.out.println("A");
   }
}

class B extends A {
   public void print() {
      System.out.println("B");
   }
}

class C extends B {
   public void print() {
      System.out.println("C");
   }
}
```

Bu kod nümunəsində siz `C` classının daxilində `C` classının instansını istifadə etməklə `A` classının `print()` metoduna *heç cürə* müraciət edə bilmirsiniz:

```
class C extends B {
   public void print() {
      System.out.println("C");
   }

   public void test(){
      A a = new C();
      a.print();            // C
      ((A)this).print();    // C

      this.print();         // C
      super.print();        // B
      super.super.print();  // DOES NOT COMPILE
   }
}
```

Redeclaring private Methods

Java`da parent classda olan `private` metodları child classda override etmək mümkün deyil, çünki access modifier`ə görə `private` metodlar child classda görsənmir. Amma bu o demək deyil ki, biz child classda parent classdakı `private` metodla eyni quruluşa malik olan metod yarada bilmərik. Bu mümkündür, amma child classda yaradılmış bu metod tamamilə yeni, ayrı və müstəqil metoddur, parent classdakı metodun override olunmuş versiyası deyil və bir-biri ilə heç bir əlaqələri yoxdur.

```
public class City {
   private String getNumberOfPeople() {
      return "Undefined";
   }
}

class Baku extends City {
   private long getNumberOfPeople() {
      return 2_225_800L;
   }
}
```

Yuxarıdakı kod normal kompayl olunur. Çünki bu metodlar bir-birindən ayrı, müstəqil metodlardır və child classdakı metod override qaydalarına əməl etmək məcburiyyətində deyil.

Hiding Static Methods

Gizli metodlar (hidden/static method) override olunmuş metodlara çox oxşayır, amma fərqli cəhətləri də var. Gizli metodlar da override metodlar üçün keçərli olan 4 qaydaya mütləq riayət etməli və əlavə olaraq yeni 5-ci qaydanın tələblərinə uyğun olmalıdır:

5. Əgər parent classdakı metod static`dirsə, o zaman child classdakı metod da mütləq static olmalıdır (*method hiding*). Yox əgər parent classdakı metod static deyilsə, o zaman child classdakı metod da static olmamalıdır (*method overriding*).

```
class Bear {
   public static void eat() {
      System.out.println("Bear is eating");
   }
   public static void sleep() {
      System.out.println("Bear is slepping");
   }
   protected void wakeUp() {
      System.out.println("Bear wake up");
```

```
    }
}

public class Panda extends Bear {
    public static void eat() {
        System.out.println("Panda is eating");
    }
    public void sleep() {    // DOES NOT COMPILE
        System.out.println("Panda is slepping");
    }
    protected static void wakeUp() {    // DOES NOT COMPILE
        System.out.println("Panda wake up");
    }
    public static void main(String[] args) {
        Bear b = new Panda();
        b.eat();    // Bear is eating
    }
}
```

`eat()` metodu gizləmə (hiding) qaydalarına riayət edir, həm parent, həm də child metod `static`dir. Ona görə də `main()` metodda `eat()` metodunu çağıranda child classdakı metodu override etmir.

`sleep()` metodu parent classda `static`dir, amma child classda yox. Buna görə də kompilyator şübhələnir ki, siz əslində gizli olmalı olan metodu override etməyə çalışırsız və buna görə də kompayl xətası verir.

`wakeUp()` metodu parent classda instance metoddur, amma child classda `static`. Bu səbəbdən kompilyator elə fikirləşir ki, siz əslində override etməli olduğunuz metodu gizlətməyə çalışırsız və xəta verir.

**Praktikada gizli metodlardan (hiding static methods) istifadə etmək məsləhət görülmür.*

Overriding vs. Hiding Methods

İcra vaxtı metodun child yaxud parent classda çağırılmasından asılı olmayaraq həmişə child classda override olunmuş versiya icra olunur. Bu qaydaya görə parent metod heç vaxt istifadə edilmir, əgər aşkar olaraq həmin metod çağırılmayıbsa – `new ParentClassName().method()`.

Amma `static` metodlarda vəziyyət fərqlidir. Əgər gizli (static) metod parent classda çağırılıbsa, o zaman həmin metodun həmişə parent versiyası icra ediləcək.

```
public class Vehicle {
    public static int getNumberOfWheels() {
        return 4;
```

```
    }
    public void vehicleDescription() {
       System.out.println("The most of vehicles have "
                    + getNumberOfWheels() + " wheels.");
    }
}

class Bicycle extends Vehicle {
    public static int getNumberOfWheels() {
       return 2;
    }
    public void bicycleDescription() {
       System.out.println("Bicycle has " + getNumberOfWheels() + " wheels.");
    }

    public static void main(String[] args) {
       Bicycle b = new Bicycle();
       b.vehicleDescription();
       b.bicycleDescription();
    }
}
```

Output:

```
The most of vehicles have 4 wheels.
Bicycle has 2 wheels.
```

Əgər hər iki classdakı `getNumberOfWheels()` metodunundan `static` sözünü silsək o zaman nəticə dəyişəcək:

```
The most of vehicles have 2 wheels.
Bicycle has 2 wheels.
```

Creating final methods

Əgər biz parent classdakı hər hansı bir metodun child classda override olunmasını istəmiriksə, həmin metodu `final` edirik. Static metodlar üçün də bu qayda keçərlidir, `final` olan `static` metodu parent classda gizlədə bilmərik:

```
class Bear {
    protected final void wakeUp() {
       System.out.println("Bear wake up");
    }
    public static final void eat() {
       System.out.println("Bear is eating");
    }
    public void sleep() {
```

```
        System.out.println("Bear is slepping");
    }
}

public class Panda extends Bear {
    public void wakeUp() {                      // DOES NOT COMPILE
        System.out.println("Panda wake up");
    }
    public static final void eat() {            // DOES NOT COMPILE
        System.out.println("Panda is eating");
    }
    public final void sleep() {
        System.out.println("Panda is slepping");
    }
}
```

Inheriting Variables

Dəyişənlər üçün qaydalar sadədir. Java dəyişənləri override etməyə icazə vermir. Əgər parent və child classda eyni adlar dəyişənlər varsa, *java override əvəzinə gizlədir*.

Hiding Variables

Static metodu gizlətmək üçün istifadə olunan qaydalar dəyişənlər üçün də eynidir. Dəyişənlər override olunmur, ona görə də əgər biz eyniadlı dəyişənə parent classda müraciət ediriksə, parent classdakı dəyişən istifadə edilir. Yox əgər child classın daxilində müraciət ediriksə, o zaman child classdakı dəyişən istifadə edilir. Amma child classın daxilində `super` açar sözü ilə birbaşa parent dəyişəni də istifadə edə bilərik:

```
class Car {
    protected String color = "black";
    static double speed = 200.;
    public void getCarDetails() {
        System.out.println("color of car: " + color);
        System.out.println("speed of car: " + speed);
    }
}

public class BMW extends Car {
    protected String color = "white";
    double speed = 320.;
    public void getBMWDetails() {
        System.out.println("color of BMW: " + color);
```

```
        System.out.println("speed of BMW: " + speed);
        System.out.println("speed difference: " + (speed - super.speed));
    }
    public static void main(String[] args) {
        BMW obj = new BMW();
        obj.getCarDetails();
        System.out.println();
        obj.getBMWDetails();
    }
}
```

Output:

```
color of car: black
speed of car: 200.0
color of BMW: white
speed of BMW: 320.0
speed difference: 120.0
```

Göründüyü kimi dəyişənlər override edilmir, referans hansı classa aiddirsə, həmin classa məxsus olan dəyişən də çağırılır:

```
Car car = new BMW();
BMW bmw = new BMW();
System.out.println(car.color);  // black
System.out.println(bmw.color);  // white
```

Creating Abstract Classes

Abstrakt classlar bizə nə üçün lazım ola bilər?

Tutaq ki, bizim `Animal` classımız var və onun daxilində bir neçə metod mövcuddur. Biz istəyirik ki, `Animal` classını varis alan (`extends` edən) bütün classlar `getName()` metodunu mütləq *implement/override* etsinlər. Bu zaman bizim köməyimizə abstrakt class və abstrakt metodlar gəlir. *Abstrakt class* - `abstract` açar sözü ilə yaradılan classlardır və **onların birbaşa obyektini yaratmaq mümkün deyil**. *Abstrakt metod* - `abstract` açar sözü ilə yaradılan metodlardır və **bu metodların gövdəsi olmur**.

```
abstract class Animal {
    protected int age;
    public void eat() {
        System.out.println("Animal is eating");
    }
    public abstract String getName();
}
```

```
public class Dog extends Animal {
   public String getName() {
      return "dog";
   }
}
```

Defining an Abstract Class

Abstrakt classın daxilində abstrakt olmayan (non-abstract) metod və dəyişənlər də təyin etmək mümkündür. Maraqlı səslənsə də bu doğrudur ki, abstrakt classın daxilində ümumiyyətlə abstrakt metod olmaya da bilər, yəni elə bir məcburiyyət yoxdur ki, abstrakt classın daxilində mütləq hər hansı bir abstrakt metod olmalıdır. Amma abstrakt metod ancaq abstrakt classda yaradıla (define) bilər, adi classda icazə verilmir:

```
public abstract class Rabbit {}       // everything is ok

public class Puppy {
   public abstract void bark();       // DOES NOT COMPILE
}
```

Əvvəlki mövzuda qeyd etdik ki, abstrakt metodlar gövdəsiz olurlar, əks halda kompayl xətası verir:

```
public abstract class Turtle {
   public abstract void swim() {}     // DOES NOT COMPILE
   public abstract int getAge() {     // DOES NOT COMPILE
      return 15;
   }
}
```

Abstrakt class `final` ola bilməz! Çünki abstrakt classı birbaşa obyektini yaratmaqla istifadə etmək mümkün deyil, ancaq başqa class tərəfindən `extends` edilməklə istifadə edilə bilər. Onu `final` təyin etməklə isə artıq bildiririk ki, bu class varis alına, yəni `extends` edilə bilməz. Ona görə də kompilyator classın eyni zamanda həm `abstract`, həm də `final` olduğunu gördükdə xəta verir:

```
public final abstract class Tortoise {}   // DOES NOT COMPILE
```

Eləcə də, **abstrakt metodlar `final` ola bilməz**. Çünki abstrakt metodlar mütləq override olunmalıdır, `final` olduqda isə override edilməyə icazə vermir və bu səbəbdən konflikt yaranır. Həmçinin bənzər səbəblərə görə metod eyni zamanda

- həm `abstract`, həm `private`
- həm `abstract`, həm `static`

ola bilməz:

```
public abstract class Goat {
```

```
   public abstract final void chew();  // DOES NOT COMPILE
   private abstract void eat();        // DOES NOT COMPILE
   static abstract void walk();        // DOES NOT COMPILE
}
```

Gündəlik praktikada aşağıdakı formada kod nümunəsinə rast gəlməyəcəksiniz, amma sadəcə bilmək yaxşı olardı ki, bu formada yazılış kompayl xətası vermir:

```
public abstract class Calculator {
   abstract void calculate();
   public static void main(String[] args){
      System.out.println("calculating");
      Calculator x = null;
      x.calculate();        // does compile, but throws NullPointerException
   }
}
```

Creating a Concrete Class

Qeyd etdik ki, abstrakt classların birbaşa obyektini yaratmaq mümkün deyil, kompayl xətası verir:

```
public abstract class Cat {
   public static void main(String[] args) {
      final Cat cat = new Cat();          // DOES NOT COMPILE
   }
}
```

Abstrakt classların istifadəsi o zaman faydalı olur ki, onlar konkret subclasslar tərəfindən varis alınsın (extends edilsin). *Konkret class (concrete class)* - abstrakt classı `extends` edən birinci abstrakt olmayan subclassdır (first nonabstract subclass) və bütün abstrakt metodları *override* etməyi tələb edir:

```
public abstract class Animal {
   public abstract String getName();
}

class Tiger extends Animal {}  // DOES NOT COMPILE
```

`Tiger` birinci konkret subclassdır və ona görə də `getName()` metodunu override etməlidir.

Elə hal ola bilər ki, birinci konkret class yox (bu nümunədə `Tiger`), bu konkret classın özünü `extends` edən digər bir konkret class abstrakt metodu (bu nümunədə `getName()`) override etsin. Amma bu halda da kompayl xətası verəcək, çünki istənilən halda *ancaq birinci konkret subclass abstrakt metodları override etməlidir*:

```
public abstract class Animal {
```

```
   public abstract String getName();
}

class Tiger extends Animal {}  // DOES NOT COMPILE

class TigerCub extends Tiger {
   public String getName() {
      return "tiger-cub";
   }
}
```

Extending an Abstract Class

Əvvəlki mövzuda yazdığımız kodda kiçik bir dəyişiklik etsək, fərqli nəticə ala bilərik:

```
public abstract class Animal {
   public abstract String getName();
}

class Tiger extends Animal {}          // DOES NOT COMPILE
abstract class Lion extends Animal {}
```

Nümunədən də göründüyü kimi əgər abstrakt class digər abstrakt classı extends edirsə, onun abstrakt metodlarını override etmək məcburiyyətində deyil, artıq istəyə bağlıdır. Amma override etməsə belə, istənilən halda həmin abstrakt metodları varis alır və bu abstrakt metodun özünü extends edən ilk konkret subclass artıq hər iki abstrakt classın abstrakt metodlarını override etməyə məcburdur:

```
abstract class Animal {
   public abstract String getName();
}

abstract class BigCat extends Animal {
   public abstract void roar();
}

public class Lion extends BigCat {
   public String getName() {
      return "Lion";
   }
   public void roar() {
      System.out.println("roar");
   }
}
```

Amma bu qaydanın özündə də bir istisna var. Əgər ortadakı abstrakt class (bizim nümunədə `BigCat`) birinci abstrakt classdakı abstrakt metodu (bizim nümunədə `getName()`) override

edərsə, o zaman artıq birinci konkret subclassdan həmin metodu override etmək tələb olunmayacaq. Çünki subclass artıq həmin metodu abstrakt metod kimi yox, konkret metod kimi qəbul edəcək.

```
abstract class Animal {
   public abstract String getName();
}

abstract class BigCat extends Animal {
   public String getName() {
      return "BigCat";
   }
   public abstract void roar();
}

public class Lion extends BigCat {
   public void roar() {
      System.out.println("roar");
   }
}
```

Bütün bu qeyd etdiklərimizi yekunlaşdıraraq aşağıdakı nəticəyə gələ bilərik;

Abstrakt class ilə bağlı vacib qaydalar:

1. Abstrakt classın `new` ilə birbaşa obyekti yaradıla bilməz;
2. Abstrakt classda istənilən sayda (və ya heç bir) abstrakt və yaxud abstrakt olmayan metod təyin edilə bilər;
3. Abstrakt classlar `private` və ya `final` ola bilməz (burada söhbət top-level classlardan gedir);
4. Abstrakt class başqa bir abstrakt classı extends edirsə, onun bütün abstrakt metodlarını öz şəxsi abstrakt metodları kimi varis alır;
5. Abstrakt classı extends edən birinci konkret subclass həmin abstrakt classın bütün abstrakt metodlarını override etməlidir.

Abstrakt metod ilə bağlı vacib qaydalar:

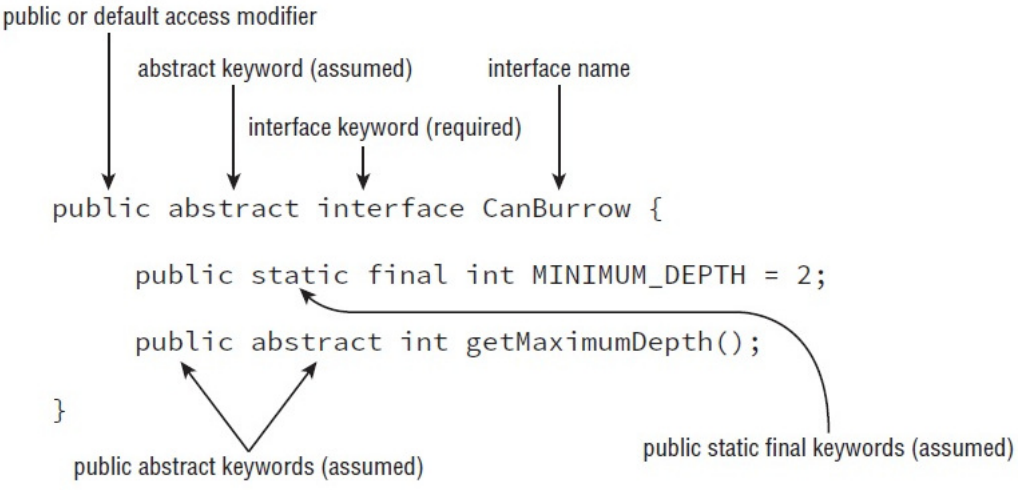
1. Abstrakt metodlar ancaq abstrakt classda təyin edilə bilər;
2. Abstrakt metodlar `private, final` və yaxud `static` elan edilə bilməz;
3. Abstrakt classda elan edilmiş abstrakt metodların gövdəsi olmamalıdır;

4. Abstrakt metodların subclassda override edilməsi qaydaları adi metodların override edilməsi qaydaları ilə eynidir.

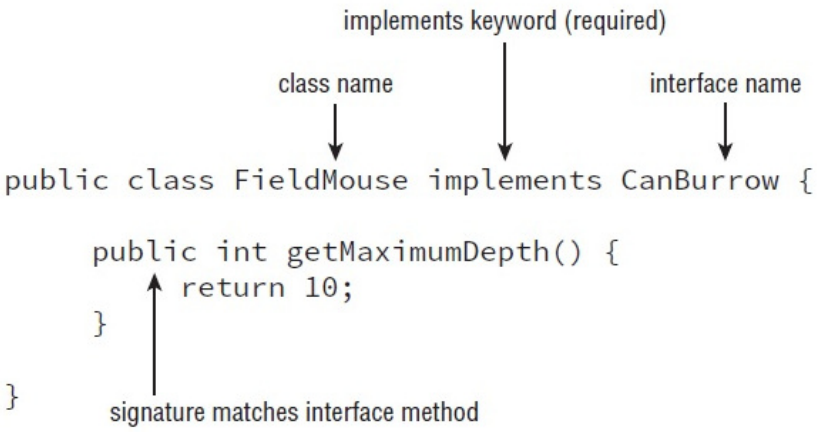
Implementing Interfaces

Java çoxvarisliliyə icazə verməsə də, bir classın istənilən sayda interfeysi implements etməsinə icazə verir və həmin interfeyslər bir-birindən vergüllə ayrılır.

Şəkil 5.2 İnterfeysin strukturu



Şəkil 5.3 İnterfeysin varis alınması



Defining an Interface

İnterfeys ilə bağlı bilməli olduğumuz vacib qaydalar aşağıdakılardır:

1. İnterfeysin `new` ilə birbaşa obyekti yaradıla bilməz;
2. İnterfeysin daxilində heç bir metod olmaya da bilər, yəni hər hansı sayda metodun olmasını tələb etmir;
3. İnterfeys `final` elan edilə bilməz;
4. Bütün top-level interfeyslər access level olaraq ancaq `public` və default ola bilərlər. Və həmçinin aşkar qeyd olunmamış olsa belə, bütün interfeyslərin `abstract` olduğu güman/qəbul edilməlidir. Buna görə də interfeysi `private`, `protected` və `final` etsək, kompayl xətası alarıq;
5. İnterfeys daxilindəki bütün metodlar (`default` və `static` metodlar istisna olmaqla) aşkar qeyd olunmamış olsa belə `abstract` və `public` metodlardır. Ona görə də interfeys metodlarını `private`, `protected` və `final` etsək, qeyd etdiyimiz qayda ilə ziddiyyət təşkil etdiyinə görə kompayl xətası alarıq (Java 9-cu versiyadan etibarən artıq interfeysdə `private` metodlar da elan etmək mümkündür, amma bu OCA SE 8 imtahanının mövzusuna daxil deyil).

Birinci 3 qayda abstrakt class üçün qeyd etdiyimiz qaydalarla eynidir. 4-cü qayda daxili (inner) interfeyslərə aid deyil (daxili interfeyslər OCA imtahanı mövzusuna daxil deyil).

Qaydalarla bağlı nümunələrə baxaq:

```
/* example for second rule */
public interface TestInterface { }

/* example for first rule */
public class TestClass {
   public static void main(String[] args) {
      TestInterface example = new TestInterface ();   // DOES NOT COMPILE
   }
}

/* example for third rule */
public final interface TestInterface { }    // DOES NOT COMPILE
```

4-cü və 5-ci qaydalar isə *"assumed keywords"* ilə əlaqəlidir. Həmin qaydalarda qeyd etdiyimiz açar sözləri əgər biz özümüz qeyd etməsək, kompilyator onları bizim əvəzimizə avtomatik olaraq əlavə edəcəkdir. Yəni aşağıda qeyd etdiyimiz nümunələr bir-biri ilə ekvivalentdir. 1-ci nümunəki kod kompilyator tərəfindən 2-ci nümunədəki koda çevrilir.

```
interface Activity {            // example-1
   void run(int speed);
   abstract void stop();
```

```
   public abstract double jump();
}

abstract interface Activity {   // example-2
   public abstract void run(int speed);
   public abstract void stop();
   public abstract double jump();
}
```

Aşağıdakı kodlar isə kompayl olunmur:

```
private final interface CanCrawl {      // DOES NOT COMPILE
   private void dig(int depth);         // DOES NOT COMPILE
   protected abstract double depth();   // DOES NOT COMPILE
   public final void surface();         // DOES NOT COMPILE
}
```

5-ci qayda ilə bağlı bir haşiyəyə çıxacam, orada qeyd etdik ki, interfeys metodları aşkar qeyd olunmasa da `public` metodlardır. İmtahanda sizi ən çox çaşdıran məqamlardan biri bu metodların override edilməsi zamanı access modifier'ə fikir verməmək ola bilər. Biz override qaydalarında qeyd etmişdik ki, child classdakı metodun access modifier`i parent classdakı metodla ən azı eyni olmalıdır, ya da ki, daha yüksək. Siz interfeysdə metodların qarşısında `public` açar sözünü görmədikdə, onu override edən zaman subclassda metodun qarşısında diqqətinizdən yayınıb `public` açar sözünü unuda bilərsiniz. Artıq bu vaxt siz kompayl xətası alacaqsınız, bu məqama diqqət etmək lazımdır:

```
interface MyInterface {
   void myMethod();
}

class MyClass implements MyInterface {
   void myMethod() {  // DOES NOT COMPILE
      System.out.println("This is my method");
   }
}
```

Aşağıda qeyd olunan kod nümunəsi sizin üçün maraqlı ola bilər. Gördüyünüz kimi interfeysin daxilində `main(String[] )` metod, *Instance Initializer Block*, *Static Initializer Block* və *Constructor* yazmışıq. Print ifadələri özü izah edir, amma təkrar olaraq qeyd edim ki, Instance Initializer Block, Static Initializer Block və Constructor interfeysin daxilində kompayl olunmur, xəta verir. Amma maraqlıdır ki, `main(String[] )` metod kompayl xətası vermir, hətta siz kodu icra etdikdə `main` metod daxilindəki *print* ifadə çap edilir. Səbəb isə odur ki, Java 8-ci versiyadan etibarən interfeysdə `static` metoda icazə verir (bu haqda daha ətraflı ayrıca bir mövzuda baxacağıq).

```
interface ITest {
```

```
    public static void main(String[] args) {
       System.out.println("This method compile successfully and print this");
    }

    { System.out.println("Instance block doesn't compile in Interface"); }
    static { System.out.println("Static block doesn't compile in Interface"); }

    public ITest() {
       System.out.println("Constructor does not compile in Interface");
    }
}
```

Amma abstrakt classların daxilində instance Initializer və static bloklar, eləcə də konstruktor yaratmaq mümkündür.

Inheriting an Interface

İnterfeys ilə bağlı bu iki qaydanı yadda saxlamaq lazımdır:

1. Əgər interfeys başqa bir interfeysi `extends` edirsə, və yaxud abstrakt class interfeysi `implements` edirsə, həmin interfeysin bütün abstrakt metodlarını öz abstrakt metodu kimi miras alır.
2. İnterfeysi `implements` edən və ya interfeysi `implements` etmiş abstrakt classı `extends` edən birinci konkret class *(first concrete class)* bütün abstrakt metodları *override* etməlidir.

1-ci bənddəki ilk cümlə sizi çaşdırmasın, əgər interfeys başqa bir interfeysi miras olaraq almaq istəyirsə, bu zaman `implements` deyil, `extends` açar sözü istifadə edilməlidir. Abstrakt classın əksinə olaraq interfeys eyni zamanda bir neçə interfeysi `extends` edə bilər:

```
public interface HasTail {
   public int getTailLength();
}

public interface HasWhiskers {
   public int getNumberOfWhiskers();
}

public interface Seal extends HasTail, HasWhiskers { }
```

`Seal` interfeysini `implements` edən istənilən konkret class `HasTail` və `HasWhiskers` interfeyslərinin də abstrakt metodlarını (`getTailLength()` və `getNumberOfWhiskers()`) override etməlidir:

```
public interface HasTail {
   public int getTailLength();
```

```
}

public interface HasWhiskers {
   public int getNumberOfWhiskers();
}

public abstract class HarborSeal implements HasTail, HasWhiskers {}
public class LeopardSeal implements HasTail, HasWhiskers {}  // DOES NOT COMPILE
```

Classes, Interfaces, and Keywords

Class interfeysi ancaq `implements` edə bilər, `extends` edə bilməz. Eləcə də interfeys başqa bir interfeysi ancaq `extends` edə bilər, `implements` edə bilməz.

```
public interface CanRun {}

class Cheetah extends CanRun {}     // DOES NOT COMPILE

class Hyena {}

interface HasFur extends Hyena {}   // DOES NOT COMPILE
```

İnterfeys ilə class arasında ancaq bir əlaqə var və o da bu sintaksis ilə mümkündür:

class implements interface.

Abstract Methods and Multiple Inheritance

Bildiyimiz kimi java eyni anda bir neçə interfeysi `implements` etməyə icazə verir. O zaman sual yarana bilər ki, bəs fərqli interfeyslərdə eyni abstrakt metod olarsa və onlar hər hansı bir class tərəfindən eyni anda `implements` edilsə nə baş verəcək?

```
interface Herbivore {
   public void eatPlants();
}

interface Omnivore {
   public void eatPlants();
   public void eatMeat();
}

public class Bear implements Herbivore, Omnivore {
   public void eatMeat() {
      System.out.println("Eating meat");
   }
   public void eatPlants() {
      System.out.println("Eating plants");
```

```
    }
}
```

Yuxarıdakı kod heç bir kompayl xətası olmadan çalışacaq. Çünki hər iki metod abstraktdır və eyni özəlliyə/davranışa malikdir. Bu halda bu metodlardan birini override etdikdə, avtomatik olaraq ikinci metod da override olunmuş hesab edilir. Başqa sözlə, əgər interfeys metodları eyni struktura (the same signature) malik olarsa, o zaman onlar **dublikat** (duplicate) metodlar hesab olunur.

Əgər metodlar fərqli struktura (metod adı eyni, parametr listi fərqli) malik olsalar, o zaman heç bir konflikt olmayacaq, çünki bu artıq metod *overloading*`dir.

Yox əgər metod adı və parametr listi eyni, amma geri dönüş tipi (return types) fərqli olsa, bu zaman həmin interfeysləri eyni anda `implements` etdikdə kompayl xətası baş verəcək. Çünki bir classda eyni ada və parametr siyahısına, lakin fərqli dönüş tipinə malik iki metod elan etmək mümkün deyil:

```
interface Herbivore {
   public int eatPlants();
}

interface Omnivore {
   public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {  // DOES NOT COMPILE
   public int eatPlants() {      // DOES NOT COMPILE
      System.out.println("Eating plants: 15");
      return 15;
   }
   public void eatPlants() {     // DOES NOT COMPILE
      System.out.println("Eating plants");
   }
}
```

Əgər biz `Bear` classında override etdiyimiz metodlardan birini silsək, kompayl xətası yenə də qalacaq. Çünki bu zaman kompilyator bizə xəbərdarlıq edəcək ki, override olunması tələb olunan metodlardan biri buraxılıb.

Əgər yuxarıdakı nümunələrdə qeyd etdiyimiz interfeyslər digər interfeys və ya abstrakt class tərəfindən varis alınsa belə yenə kompayl xətası çıxacaq, çünki kompilyator override etmədən belə bu problemi aşkarlayır:

```
interface Herbivore {
   public int eatPlants();
}

interface Omnivore {
```

```
    public void eatPlants();
}

interface Supervore extends Herbivore, Omnivore {}            //doesn't compile

abstract class AbstractBear implements Herbivore, Omnivore {}  //doesn't compile
```

Qeydlərimin içərisində bu mövzu ilə əlaqəli maraqlı bir suala rast gəldim. Deməli, bu sual 2015-ci ildə Facebookda "Java sertifikat sualları" qrupumuzda paylaşılıb, sual maraqlı olduğu üçün mən də onu "screenshot" edib atmışam qeydlərə. Sual belədir:

Consider these two interfaces:

```
interface I1 { void m1() throws java.io.IOException; }
interface I2 { void m1() throws java.sql.SQLException; }
```

What methods have to be implemented by a class that says it implements `I1` and `I2` ?

A) Both, `public void m1() throws SQLException;` and `public void m1() throws IOException;`
B) `public void m1() throws Exception`
C) The class cannot implement both the interfaces simultaneously as they have conflicting methods.
D) `public void m1() throws SQLException, IOException;`
E) None of the above.

Sual çətin sualdır, təkrar baxdıqda yenə hiss etdim bunu. Amma izahını elə o vaxt qrupda yazmışam (https://www.facebook.com/groups/javacertification/permalink/918028724912933/), imtahan ərəfəsində daha hazırlıqlı olursan, amma zamanla bəzi mövzular unudulur. İzaha baxmazdan öncə özünüz tapmağa cəhd edin.

Interface Variables

İnterfeys dəyişənləri üçün aşağıdakı 2 qayda mövcuddur:

1. İnterfeys dəyişənlərinin `public`, `static` və `final` olduğu fərz edilməlidir, aşkar qeyd olunmasa belə. Ona görə də bu dəyişənləri `private`, `protected` və ya `abstract` elan etmək istəsək kompayl xətası alarıq.
2. İnterfeys dəyişənləri `final` olduğuna görə onların dəyəri elan edilən zaman mənimsədilməlidir.

Bu qaydalara əsasən interfeys dəyişənlərinə ***sabit (constant)*** dəyişənlər kimi də baxmaq olar.

Aşağıdakı kod nümunələri bir-birilə ilə ekvivalentdir. 1-ci nümunəki kod kompilyator tərəfindən 2-ci nümunədəki koda çevrilir:

```
public interface CanSwim {    // example-1
   int MAXIMUM_DEPTH = 100;
   final static boolean UNDERWATER = true;
   public static final String TYPE = "Submersible";
}

public interface CanSwim {    // example-2
   public static final int MAXIMUM_DEPTH = 100;
   public static final boolean UNDERWATER = true;
   public static final String TYPE = "Submersible";
}
```

Aşağıdakı kod nümunələri isə kompayl olunmur:

```
public interface CanDig {
   private int MAXIMUM_DEPTH = 100;                 // DOES NOT COMPILE
   protected abstract boolean UNDERWATER = false;   // DOES NOT COMPILE
   public static String TYPE;                       // DOES NOT COMPILE
}
```

Öncəki mövzulardan birində qeyd etmişdik ki, fərqli interfeyslərdə eyni abstrakt metod olarsa və onlar hər hansı bir class tərəfindən eyni anda `implements` edilərsə, class bu metodlardan birini override etdikdə, avtomatik olaraq ikinci metod da override olunmuş hesab edilir. Bəs fərqli interfeyslərdə eyni adlı dəyişənlər olarsa və onlar hər hansı bir class tərəfindən eyni anda `implements` edilərsə, onda nə baş verəcək? Axı dəyişənlər override edilmir?!

Deməli, kompilyator implements zamanı fərqli interfeyslərdə eyni adlı dəyişənlər aşkarladıqda heç bir problem olmadan kodu kompayl edir, amma həmin dəyişəni çağırarkən onun qarşısında interfeysin adı aşkar şəkildə qeyd edilməlidir, əks halda kompilyator dəyişənin hansı interfeysə aid olduğunu təyin edə bilmir və xəta verir:

```
interface First {
   String s = "first";
   void test();
}

interface Second {
   String s = "second";
   void test();
}

class Row implements First, Second {

   public void test() {
      System.out.println("test");
   }
```

```
    public static void main(String[] args) {
        Row r = new Row();
        r.test();
        System.out.println(r.s);  // DOES NOT COMPILE, reference to s is ambiguous
        System.out.println(((First)r).s);   // first
        System.out.println(((Second)r).s);  // second
    }
}
```

Default Interface Methods

Default metod java 8 ilə yeni gələn özəlliklərdən biridir. Default metod interfeysin daxilində `default` açar sözü ilə təyin edilən və gövdəsi olan metoddur. İnterfeysi `implements` edən classların default metodu override etmək kimi bir məcburiyyəti yoxdur. Classların sərbəst seçimi var, istəsə override edə bilər, istəməsə yox. Əgər override etməsə, o zaman interfeysdə elan edilən default metod istifadə ediləcəkdir. Sintaksisi aşağıdakı kimidir:

```
public interface DefaultMethod {
    public int getCount();

    public default int getNumber(){
        return 7;
    }

    public default void showNumber();  // DOES NOT COMPILE

    public String getName(){           // DOES NOT COMPILE
        return "Ülvi";
    }
}
```

Default metodlar üçün aşağıdakı qaydalar mövcuddur:

1. Default metod ancaq interfeys daxilində elan edilə bilər, class və abstrakt class daxilində elan edilə **bilməz**;
2. Default metodlar `default` açar sözü ilə işarələnməlidir və əgər default açar sözü ilə işarələnibsə, mütləq gövdəsi **olmalıdır**;
3. Default metodların `static`, `final` və ya `abstract` olduğu fərz **edilməməlidir** *(daha dəqiq desək default metodlarla bu açar sözlərin birgə istifadəsinə icazə verilmir),* onlar interfeysi `implements` edən class tərəfindən istifadə edilə və ya override oluna bilər;
4. Bütün interfeys metodları kimi default metodların da `public` olduğu fərz **edilməlidir**. Və bu səbəbdən `private` və ya `protected` kimi işarələnən default metodlar kompayl olunmur.

Əgər hər hansı bir interfeys, daxilində default metod olan başqa bir interfeysi `extends` edərsə, həmin default metod ilə bir neçə cür davrana bilər:

- ✓ default metodla bağlı hər hansı bir davranışdan imtina edə bilər, bu zaman original default metod üçün təyin edilmiş gövdə istifadə olunur;
- ✓ default metodu yenidən override edə bilər;
- ✓ default metodu yenidən `abstract` metod kimi elan edə bilər. Belə olduqda isə artıq ikinci interfeysi `implements` edən class həmin metodu override etmək məcburiyyətindədir.

Nümunə:

```
public interface HasFins {
   default int getInt() {
      return 1;
   }
   public default double getDouble() {
      return 2;
   }
   default public boolean getBoolean() {
      return true;
   }
}

interface SharkFamily extends HasFins {
   int getInt();
   default double getDouble() {
      return 5;
   }
}

class TestHasFins implements HasFins { }

class TestSharkFamily implements SharkFamily {
   TestSharkFamily() {
      System.out.println(getDouble());   // 5.0
   }
   public int getInt() {
      return 7;
   }
   public static void main(String args[]) {
      new TestSharkFamily();
   }
}
```

İnterfeysin `default` metodlarını həmin interfeysi `implements` edən konkret class daxilində `super` açar sözü ilə çağırmaq mümkün deyil. `super` açar sözü ancaq classlar üçün istifadə edildikdə düzgün işləyir. Əgər `default` metodlar üçün istifadə etmək istəyiriksə, mütləq `super` açar sözünün qarşısında həmin interfeysin adını qeyd etmək lazımdır:

```
interface Walk {
   public default int getSpeed() {
      return 10;
   }
}

abstract class Animal {
   public String getColor(){
      return "black";
   }
}

class Cat extends Animal implements Walk {

   public Cat() {
      System.out.println(this.getColor());
      System.out.println(super.getColor());

      System.out.println(this.getSpeed());
      System.out.println(super.getSpeed());       // DOES NOT COMPILE
       System.out.println(Walk.super.getSpeed());
   }
}
```

MyExamCloud'da suallardan birində maraqlı bir faktla qarşılaşmışdım. Deməli, faktda qeyd olunurdu ki, siz interfeysdə default metodları `java.lang.Object` classına aid (`final` olmayan) hər hansı bir metodu override etmək üçün istifadə edə bilməzsiniz, əks halda kompayl xətası alacaqsınız:

```
interface A {
   int groupID = 10;
   default boolean equals(Object obj) {    // DOES NOT COMPILE
      return this.groupID == ((A) obj).groupID;
   }
   static void print() {
      System.out.println("A");
   }
}
```

Default Methods and Multiple Inheritance

Əvvəlki mövzulardan birində qeyd etdik ki, hər hansı bir class tərəfindən `implements` olunan interfeyslərdə eyni abstrakt metodlar olarsa, problemsiz kompayl olunur, çünki dublikat metodlar sayılır. Amma default metodlarla vəziyyət nisbətən fərqlidir. Kompilyator eyni default metodlar aşkarladıqda xəta verir:

```
interface Walk {
   public default int getSpeed() { return 10; }
}

interface Run {
   public default int getSpeed() { return 20; }
}

public class Cat implements Walk, Run {     // DOES NOT COMPILE
   public static void main(String[] args){
      System.out.println(new Cat().getSpeed());
   }
}
```

Bu nümunədə `Cat` classı iki eyni default metodu (`getSpeed()`) irs alır, amma təyin edə bilmir ki, hansı metodu istifadə etməlidir, `10` çap olunmalıdı ya `20` ?! Ona görə də kompilyator belə konfliktlə rastlaşdıqda xəta verir. Amma burada bir istisna var: əgər subclass konflikt yaradan default metodu (`getSpeed()`) override etsə, o zaman bu kod heç bir problem olmadan kompayl olunacaqdır:

```
public class Cat implements Walk, Run {
   public int getSpeed(){ return 30; }
   public static void main(String[] args){
      System.out.println(new Cat().getSpeed());   // 30
   }
}
```

Static Interface Methods

Java 8 ilə gələn yeniliklərdən biri də odur ki, artıq interfeys daxilində `static` metodlar elan etmək mümkündür. Bu metodlar aşkar şəkildə (explicitly) `static` açar sözü istifadə edərək elan edilir və class daxilində istifadə etdiyimiz adi `static` metodlarla demək olar ki, eynidir, ***aralarında ancaq bir fərq var*. İnterfeys daxilində elan edilmiş static metodlar həmin interfeysi implements edən classlar tərəfindən miras alınmır (is not inherited).** Əgər həmin metodu istifadə etmək istəyiriksə, o zaman mütləq metodun əvvəlində interfeysin adı qeyd olunmalıdır.

Static metodlar üçün aşağıdakı qaydalar mövcuddur:

1. Bütün interfeys metodları kimi `static` metodların da `public` olduğu fərz **edilməlidir**. Və bu səbəbdən `private` və ya `protected` kimi işarələnən `static` metodlar kompayl olunmur;
2. Bu `static` metodlara müraciət etmək üçün **mütləq** metodun əvvəlində interfeysin adı istifadə edilməlidir.

Nümunə:

```
interface Hop {
   static int getJumpHight() {
      return 7;
   }
}

abstract class Animal {
   static int getSpeed(){
      return 11;
   }
}

public class Dog extends Animal implements Hop {
   public static void main(String[] args) {
      Hop h = new Dog();
      System.out.println("Dog speed: " + getSpeed());
      System.out.println("Dog jump hight: "+getJumpHight());   // DOESN'T COMPILE
      System.out.println("Dog jump hight: "+h.getJumpHight()); // DOESN'T COMPILE
   }
}
```

Nümunədən də gördüyümüz kimi `Dog` classı abstrakt `Animal` classının `static` metodunu miras alır, amma `Hop` interfeysinin `static` metodunu miras almır, ona görə də `getJumpHight()` metodu kompayl olunmur. Kompayl olunması üçün bu metodun adının əvvəlində `Hop` interfeysi aşkar şəkildə qeyd edilməlidir:

```
public class Dog extends Animal implements Hop {
   public static void main(String[] args) {
      System.out.println("Dog jump hight is " + Hop.getJumpHight());
   }
}
```

Default metodlar mövzusunda qeyd etdik ki, kompilyator eyni `default` metodlar aşkarladıqda xəta verir. Amma eyni `static` metodlar üçün bu qayda keçərli deyil. Belə ki, kompilyator `implements` zamanı fərqli interfeyslərdə eyni `static` metodlar aşkarladıqda heç bir problem olmadan kodu kompayl edir, çünki `static` metodlar subclasslar tərəfindən miras alınmır və hər `static` metoda onun məxsus olduğu interfeys adı ilə müraciət olunmalıdır:

```
public interface Interface1 {
```

```
    static void staticMethod() {}
}

public interface Interface2 {
    static void staticMethod() {}
}

public interface Interface3 extends Interface1, Interface2 {}

public class AnyClass implements Interface1, Interface2 {}
```

Understanding Polymorphism

Polimorfizm Java'da OOP-nin əsas prinsiplərindən biri hesab edilir. Polimorfizm nədir deyə sual versək, ona şəkildəyişdirmə, çoxşəkillilik kimi tərifl ər verə bilərik. Fərqli mənbələrdə polimorfizmə fərqli yanaşmalar, fərqli tərifl ər verilir. Onları ümumiləşdirsək deyə bilərik ki, polimorfizm eyni bir hərəkətin bir neçə fərqli şəkildə icra edilməsidir.

Öz yazdığım real bir kod parçasından bir nümunə verməyə çalışacam. Deməli bizim `Vacancy` adlı bir interfeysimiz var və onun `getVacancies()` adlı bir metodu var. Və bu interfeysi implements edən 3 classımız var. Daha yaxşı olar ki, əvvəlcə kod nümunəsini tam şəkildə yazaq və sonra izaha həmin kod nümunəsi üzərindən davam edək:

```
interface Vacancy {
    public void getVacancies();
}

class JobSearchAz implements Vacancy {
    public void getVacancies() {
        System.out.printf("%-23s %s%n", "jobsearch.az selector:",
                                        "table.hotvac tr td.hotv_text");
    }
}

class BossAz implements Vacancy {
    public void getVacancies() {
        System.out.printf("%-23s %s%n", "boss.az selector:",
                                        ".results .results-i");
    }
}

class RabotaAz implements Vacancy {
    public void getVacancies() {
        System.out.printf("%-23s %s%n", "rabota.az selector:",
                              "div#vacancy-list ul.visitor-vacancy-list li");
    }
```

```
}

public class VacancyFinder {

   public static void main(String[] args) {

      /* 1st - long way
      List<Vacancy> listWebSite = new ArrayList<>();
      listWebSite.add(new JobSearchAz());
      listWebSite.add(new BossAz());
      listWebSite.add(new RabotaAz());
      for (Vacancy v : listWebSite)
         displayVacancySelector(v);
      */

      // 2nd - short way
      List<Vacancy> listWebSite = Arrays.asList(new JobSearchAz(),
                                                new BossAz(),
                                                new RabotaAz());
      listWebSite.forEach(v -> displayVacancySelector(v));
   }

   private static void displayVacancySelector(Vacancy v){
      v.getVacancies();
   }
}
```

Output:

```
jobsearch.az selector:  table.hotvac tr td.hotv_text
boss.az selector:       .results .results-i
rabota.az selector:     div#vacancy-list ul.visitor-vacancy-list li
```

Koddan da göründüyü kimi bizim 3 vakansiya saytımız (classımız) var:

```
JobSearchAz, BossAz, RabotaAz
```

Bizdə icra edilən ancaq bir hərəkət var - vakansiyanın tapılması. Amma bütün saytlar üçün vakansiyanın tapılması şəkli eyni deyildir. Hər saytın öz fərqli strukturu var və biz hər sayt üçün özünəməxsus selector'lar istifadə etməliyik. Başqa sözlə biz eyni bir davranışı (vakansiyanın tapılması) bir neçə fərqli şəkildə icra edəcəyik (saytların sayına uyğun). Bu məqsədlə də biz hər üç vakansiya classımız üçün `Vacancy` interfeysini `implements` edirik və onun `getVacancies()` metodunu override edirik. Kod nümunəsində diqqət yetirdinizsə `displayVacancySelector()` metodu `Vacancy` interfeysi tipində parametr qəbul edir, amma biz həmin metoda həmin interfeysi implements edən classların instanslarını göndəririk. Bu polimorfizm özəlliyinin sayəsində mümkündür. Buna *"polymorphic parameters"* də deyilir, daha aşağıda ayrıca baxacağıq.

Ümumiyyətlə, java obyektinə bir neçə cür müraciət edilə bilər:

- obyektin öz tipində olan referans ilə;
- obyektin superclassının referansı ilə;
- obyektin `implements` etdiyi interfeys tipində olan referans ilə və s.

Əgər obyekt implements/extends etdiyi interface/superclassa mənimsədilibsə, cast etmək tələb olunmur:

```
interface MyInterface {
   public boolean interfaceMethod();
}

class Super {
   public String superMethod() {
      return "super";
   }
}

public class Sub extends Super implements MyInterface {
   public int var = 10;

   public boolean interfaceMethod() {
      return false;
   }

   public static void main(String args[]) {
      Sub sub = new Sub();
      System.out.println(sub.var);
      MyInterface inf = sub;
      System.out.println(inf.interfaceMethod());
      Super sup = sub;
      System.out.println(sup.superMethod());
   }
}
```

Output:

```
10
false
super
```

Bu nümunədə ancaq bir obyekt yaradılır (`Sub`) və həmin obyekt öz superclassı - `Super` və implements etdiyi interfeys - `MyInterface`in referanslarına heç bir problem olmadan mənimsədilir. Bu polimorfizmin qeyd etdiyimiz özəllikləri sayəsində mümkündür.

Artıq həmin superclass və interfeysin referansları vasitəsilə bəzi metod və dəyişənləri çağırmaq mümkündür. Bəs hansı metod və dəyişənləri? **Ancaq və ancaq həmin referansın aid olduğu obyektin daxilində mövcud olan metod və dəyişənlər çağırıla bilər.** Əks halda kompayl xətası baş verir:

```
MyInterface inf = sub;
System.out.println(inf.var);                // DOES NOT COMPILE

Super sup = sub;
System.out.println(sup.interfaceMethod());  // DOES NOT COMPILE
```

Biz bu nümunədə `inf` referansı vasitəsilə birbaşa olaraq ancaq `MyInterface` interfeysinin daxilində mövcud olan metodları (həmçinin dəyişənləri, amma interfeys dəyişənləri `static` olduğundan onları birbaşa interfeys adı ilə də çağırmaq mümkündür, referansa ehtiyac yoxdur) çağıra bilərik. Ona görə də o `var` dəyişənini tanımır. Eləcə də `sup` referansının ancaq `Super` classı daxilində təyin edilmiş metod və dəyişənlərə birbaşa icazəsi var, `interfaceMethod()` metoduna birbaşa müraciət edə bilməz.

Polimorfizmin avantajları:

- təkrar istifadə edilə bilən (reusable) kod yaratmağa imkan verir;
- kodu daha dinamik şəklə gətirməyə kömək edir.

Casting Objects

Əvvəlki mövzuda biz təkcə `Sub` classının instansını yaratdıq və həmin obyektə superclass və interfeys referansı ilə müraciət etdik. İndi biz həmin referans tiplərini yenidən subclassın referansına mənimsədə bilərik. Birinci dəfə biz belə desək daha kiçik tipi daha böyük tipə mənimsədirdik. Bu *upcasting* adlanır. Upcasting`də aşkar şəkildə cast edilmək tələb olunmur və daha etibarlıdır. Yox əgər biz tərsinə, yəni böyük tipi kiçik tipə mənimsətmək istəyiriksə, bu artıq *downcasting* adlanır və aşkar şəkildə cast edilməyi tələb edir, əks halda kompayl xətası verir:

```
Super sup = sub;
MyInterface inf = sub;
Sub sub2 = sup;    // DOES NOT COMPILE
Sub sub3 = inf;    // DOES NOT COMPILE
```

Cast etdikdən sonra isə kompayl olunur:

```
Sub sub4 = (Sub) sup;
// Sub sub4_ = (Sub)new Super();  // does compile but throws exception
System.out.println(sub4.var);                // 10
System.out.println(sub4.interfaceMethod());  // false
System.out.println(sub4.superMethod());      // super

Sub sub5 = (Sub) inf;
System.out.println(sub5.var);                // 10
System.out.println(sub5.interfaceMethod());  // false
System.out.println(sub5.superMethod());      // super
```

Casting ilə bağlı yadda saxlamalı bəzi əsas qaydalar:

1. Əgər obyekt subclassdan superclassa cast olunursa, onda aşkar cast tələb **olunmur**;
2. Əgər obyekt superclassdan subclassa cast olunursa, onda aşkar cast tələb **olunur**;
3. Kompilyator bir-biri ilə əlaqəsi olmayan tipləri (unrelated types) cast etməyə icazə vermir;
4. Hətta kod problemsiz kompayl olunsa belə, icra vaxtı exception verə bilər (buna nümunə ilə baxdıqda daha aydın olacaq).

1-ci və 2-ci qayda ilə bağlı nümunəyə baxdıq, indi **3-cü** qayda ilə bağlı nümunəyə baxaq:

```
public interface Fly {}

public class Bird {}

public class Fish {
   public static void main(String[] args) {
      Fish fish = new Fish();
      Bird bird = (Fish)fish;   // line1, DOES NOT COMPILE
      Fly fly = (Fly)fish;      // line2, does compile but throws exception
   }
}
```

Burada `Bird` və `Fish` classları arasında heç bir əlaqə yoxdur, ona görə də kompayl xətası verir. Əgər `Fish` classı `Bird` classını `extends` etsə idi, o zaman *line1* kompayl olunardı.

İnterfeyslər ilə isə vəziyyət nisbətən fərqlidir, `Fly` interfeysi ilə `Fish` classı arasında heç bir bağlılıq olmasa da cast etdikdə *line2* kompayl xətası baş vermədən dərlənir, amma icra vaxtı exception baş verir. Əgər `Fish` classı `Fly` interfeysni implements etsə, o zaman exception baş verməyəcək və normal icra ediləcək (bunu 4-cü qaydaya da aid etmək olar, bu qaydaya bir az sonra aydınlıq gətirəcəyik).

İndi isə 4-cü qayda ilə bağlı nümunəyə baxaq:

```
public class Mercedes {}
public class ML350 extends Mercedes {
   public static void main(String[] args) {
      Mercedes mercedes = new Mercedes();
      // mercedes = new ML350();  // exception will not throw if we uncomment this line
      ML350 ml = (ML350)mercedes; // does compile but throw ClassCastException
   }
}
```

İcra vaxtı müəyyən edilir ki, `ml` referansının işarə etdiyi obyekt `ML350` classının instansı deyil, ona görə də `ClassCastException` fırladılır. Bu nümunədə əsas onu yadda saxlamaq lazımdır ki, yaradılan obyektin `ML350` classı ilə istənilən halda heç bir əlaqəsi yoxdur.

`ClassCastException`ın qarşısını almaq üçün adətən `instanceof` operatorundan istifadə edilir:

```
if(mercedes instanceof ML350){
   ML350 ml = (ML350)mercedes;
}
```

`instanceof` operatorundan sağ tərəfdə class adı gəlməlidir, sol tərəfində isə yoxlanılacaq referans qeyd edilməlidir.

Coderanch forumunda bu mövzu ilə bağlı bir suala rast gəlmişdim və bu sualın sayəsində sertifikat imtahanına hazırlıq ərəfəsində ən maraqlı faktlardan birini öyrənmiş olmuşdum. Sual belə idi (http://www.coderanch.com/t/656149/ocajp/certification/ClassCastException-compiler-error-incompactible-types):

```
1.  class Ink{}
2.  interface Printable {}
3.  class ColorInk extends Ink implements Printable {}
4.  class BlackInk extends Ink{}
5.  class TwistInTaleCasting {
6.    public static void main(String args[]) {
7.      Printable printable = null;
8.      BlackInk blackInk = new BlackInk();
9.      printable = (Printable)blackInk;  //does compile, but throws ClassCastException
10.   }
11. }
```

Deməli, istifadəçi soruşur ki, `BlackInk` classı ilə `Printable` interfeysi arasında heç bir iyerarxik əlaqə yoxdur, bəs niyə 9-cu sətir kompayl xətası vermir?

Casting ilə bağlı qaydalarda biz də qeyd etdik ki, kompilyator bir-biri ilə əlaqəsi olmayan tipləri (unrelated types) cast etməyə icazə vermir. Bəs bu nümunədə niyə buna icazə verir?

İlk öncə qeyd edək ki, `Printable` sırf interfeys olduğu üçün bu baş verir. Əgər biz `Printable` interfeysini dəyişib class etsək, kompilyator artıq xəta verəcək. İndi isə keçək izahına.

9-cu sətrin kompayl xətası verməməsinin yeganə səbəbi odur ki, `BlackInk` classı `final` deyil. Əgər `BlackInk` classını `final` etsək kompayl xətası alacağıq. Bunun mətləbə nə dəxli var deyirsinizsə, onda ardına baxaq.

Kompilyator fikirləşir ki, kimsə `BlackInk` classından törəmiş hər hansı bir subclass yarada bilər və bu subclass da öz növbəsində `Printable` interfeysini implements edə bilər:

```
class DarkBlackInk extends BlackInk implements Printable {}
```

Və daha sonra biz 8-ci sətri aşağıdakı formada dəyişə bilərik:

```
BlackInk blackInk = new DarkBlackInk();
```

Artıq bu dəyişikliklərdən sonra `TwistInTaleCasting` classı heç bir kompayl xətası olmadan dərlənəcək və icra vaxtı heç bir exception baş verməyəcək. Sırf bu ehtimalın mümkün ola biləcəyindən dolayı kompilyator `BlackInk` classı ilə `Printable` interfeysi arasında birbaşa

iyerarxik əlaqə olmasa da `BlackInk` classının instansını `Printable` interfeysinin referansına mənimsətdikdə cast ilə bağlı kompayl zamanı xəta mesajı vermir.

Amma biz `BlackInk` classını `final` etməklə deyirik ki, bu classdan varis alına bilməz, başqa sözlə subclassı ola bilməz. Artıq bu zaman kompilyator bizə bildirir ki, *"incompatible types: BlackInk cannot be converted to Printable"*.

Enthuware test bankından başqa bir maraqlı sual nümunəsinə baxaq:

```
interface I {}
class A implements I {}
class B extends A {}
class C extends B {}
class D {
   public static void main(String[] args) {
      A a = new A();
      B b = new B();
   }
}
```

Sualda soruşulur ki, verilmiş kod nümunəsinə əsasən aşağıdakı bəndlərdən hansı kompayl və icra vaxtı xəta vermədən çalışacaq?

A) `a = (B)(I) b;`
B) `b = (B)(I) a;`
C) `a = (I) b;`
D) `I i = (C) a;`

Əvvəlcə yanlış cavablardan başlayaq.

C - bəndi kompayl xətası verəcək, çünki *"A is-a I"* əlaqəsi doğru olsa da, *"I is-a A"* əlaqəsi doğru deyildir;

B - bəndi kompayl xətası vermir, çünki sonda biz `a` referansını `B` classına cast edirik. Amma icra vaxtı `ClassCastException` verəcək, çünki `a` referansı `B` classına aid obyektə işarə etmir. Əgər `A a = new B();` olarsa, bu bənd doğru olar;

D - bəndi kompayl xətası vermir, çünki `C` classı `B` classından törəyib, `B` classı da öz növbəsində `A` classından. Bu səbəbdən də `C` classı eyni zamanda `A` classının da subclassı hesab edilir. `A` classı da `I` interfeysini implements etdiyindən *"C is-a I"* əlaqəsi doğrudur. Amma icra vaxtı bu da `ClassCastException` verəcək, çünki `a` referansı `C` classına aid obyektə işarə etmir. Əgər `A a = new C();` olarsa, bu bənd doğru olar;

A - bəndi doğru cavabdır, çünki `b` referansı `I` interfeysinə və sonra yenidən `B` classına cast edilə bilər. `b` referansı `B` classına işarə etdiyindən və həmçinin *"B is-a A"* əlaqəsi doğru olduğundan kod nümunəsi normal kompayl olunur və heç bir xəta baş vermədən icra edilir.

"IS-A" əlaqəsi ilə bağlı ətraflı məlumat üçün aşağıdakı yazını oxuya bilərsiniz:

https://www.w3resource.com/java-tutorial/inheritance-composition-relationship.php

Virtual Methods

Virtual metodlar o metodlar hesab edilir ki, icra edilən vaxta qədər onun dəqiq implementasiyasını təyin etmək mümkün deyil (*A virtual method is a method in which the specific implementation is not determined until runtime*).

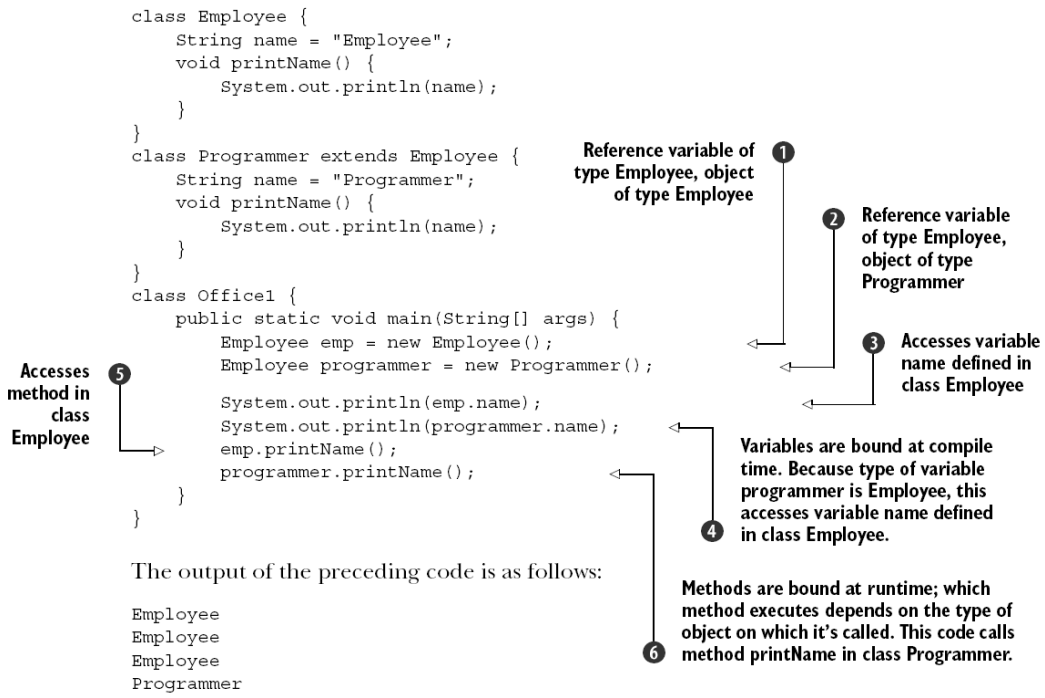
Faktiki olaraq `final`, `static` və `private` olmayan bütün java metodları virtual metodlar hesab olunur, çünki onlardan istəniləni icra vaxtı override oluna bilər.

```
class Bird {
   String getName() {
      return "Unknown";
   }
   void displayInformation() {
      System.out.println("The bird name is: " + getName());
   }
}

public class Pigeon extends Bird {
   String getName() {
      return "Pigeon";
   }
   public static void main(String[] args) {
      Bird bird1 = new Bird();
      bird1.displayInformation();     // The bird name is: Unknown

      Bird bird2 = new Pigeon();
      bird2.displayInformation();     // The bird name is: Pigeon

      Pigeon pigeon1 = (Pigeon)bird2; // bird2`ni bird1 ilə əvəz etdikdə exception verir
      pigeon1.displayInformation();   // The bird name is: Pigeon

      Pigeon pigeon2 = (Pigeon) new Bird();   // throws ClassCastException
      pigeon2.displayInformation();
   }
}
```

Burada əsas diqqət ediləsi məqam odur ki, `bird2.displayInformation()` metodunu çağırdıqda, bu metodun daxilində çağırılan `getName()` metodu `Pigeon` classında olan eyniadlı metodla yerini dəyişir. Başqa sözlə desək, parent class olan `Bird` classının da `getName()` metodu var və kompayl vaxtı `Bird` classı `Pigeon` classındakı `getName()` metodu ilə bağlı

məlumatlı olmur. Ancaq icra vaxtı artıq həmin metodun override olunmuş versiyası çağırılır. Və bu da polimorfizmin təbiəti ilə əlaqəli bir nüansdır.

Aşağıdakı nümunədə bu qeyd etdiklərimiz əyani olaraq bir daha göstərilir:



Polymorphic Parameters

Polimorfizmin ən faydalı xüsusiyyətlərindən biri subclassın və ya interfeysin instansının metoda parametr olaraq göndərilə bilinməsidir. Tutaq ki, bir metodumuz var və parametr olaraq interfeys instansı qəbul edir. Bu qaydaya görə həmin interfeysi implements edən istənilən classın obyektini bu metoda parametr olaraq göndərə bilərik. Çünki subtipdən supertipə cast edəndə aşkar cast tələb olunmur. Bu özəlliyi metodun *polimorfik parametrləri* kimi də adlandırmaq olar.

```
interface ProcessingService {
   public void service();
}

class AzeriCard implements ProcessingService {
   public void service() {
      System.out.println("AzeriCard is servicing..");
   }
}
```

```
class MilliKart implements ProcessingService {
   public void service() {
      System.out.println("MilliKart is servicing..");
   }
}

class KapitalBank implements ProcessingService {
   public void service() {
      System.out.println("KapitalBank is servicing..");
   }
}

public class CardProcessingCentre {
   public static void service(ProcessingService processing) {
      processing.service();
   }
   public static void main(String[] args) {
      service(new AzeriCard());
      service(new MilliKart());
      service(new KapitalBank());
   }
}
```

Output:

```
AzeriCard is servicing..
MilliKart is servicing..
KapitalBank is servicing..
```

Əlavələr

Nümunə 1:

```
class X {
   public String print() { return "X"; }
}
class Y extends X {
   // public String print() { return "Y"; }
}
class Z extends Y {
   // public String print() { return "Z"; }
}
class A extends Z {}

class Main {
   public static void main(String[] args) {
      System.out.println(new A().print());   // prints X
   }
}
```

Əgər `Y` classındakı `print()` metodunu kommentdən çıxarsaq `Y` çap ediləcək, `Z` classındakı `print()` metodunu kommentdən çıxarsaq isə `Z` çap ediləcək.

Nümunə 2:

Aşağıdakı kod icra edildikdə ekrana nə çap ediləcək?

```
3: interface Inf1 { int method1(); }
4:
5: interface Inf2 { boolean method2(); }
6:
7: class Test implements Inf1, Inf2 {
8:    public int method1() { return 7; }
9:    public boolean method2() { return true; }
10:   public static void main(String[] args) {
11:      Object obj = new Test();
12:      Inf1 i1 = (Inf1) obj;
13:      Inf2 i2 = (Inf2) obj;
14:      System.out.print(i1.method1());
15:      System.out.print(i2.method2());
16:   }
17: }
```

Bu cür sual nümunələri mənə görə "Chapter 5" üzrə ən çətin sual nümunələri hesab edilə bilər. Əgər işin içinə "downcasting" daxil oldusa, o zaman diqqətli olmaq lazımdır. Bu nümunədə ilk öncə kompayl xətasının olub-olmaması yoxlanılmalıdır. Əgər görsək ki, hər şey qaydasındadır, kod kompayl olunur, o zaman icra vaxtı exception`un baş verib verməyəcəyi araşdırılmalıdır. Əgər bu dəfə də hər şey qaydasında olarsa, o zaman print ifadədə nə çap ediləcək ona baxılmalıdır.

Main metoda keçməzdən öncə baxmalıyıq ki, `Test` classı `Inf1` və `Inf2` interfeyslərinin abstrakt metodlarını override edibmi?

Abstrakt metodlar override edilib. Artıq main metoda keçə bilərik. *11-ci* sətirdə problem yoxdur, çünki bütün classlar `Object` classından törəmədir. Burada "upcasting" baş verir və `Test` classının instansı `obj` referansına mənimsədilir. *12* və *13-cü* sətirlərdə diqqətli olmaq lazımdır, çünki burada downcasting baş verir (əslində buna tam olaraq downcasting də demək olmur, çünki interfeyslər `Object` classından törəmirlər). Downcasting zamanı həm kodun kompayl olunması, həm də icra vaxtı `ClassCastException`un baş verə bilmə ehtimalı - hər ikisi yoxlanılmalıdır.

`Object` classını interfeysə cast etmək mümkündür, buna görə də *12* və *13-cü* sətirlər kompayl xətası vermir. `obj` referansı özündə `Test` classının instansını saxladığından və `Test` classı da `Inf1` və `Inf2` interfeyslərini implements etdiyindən bu sətirlər icra vaxtı `ClassCastException` verməyəcək. Deməli, kod həm normal kompayl olunur, həm də icra vaxtı exception vermir. Və kodu icra etdikdə `7true` ifadəsi çap edilir.

Nümunə 3:

```
interface Behaviour {}

abstract class Parent {
   public int age;
}

class Child extends Parent implements Behaviour {}

class Main {
   public void test(Parent parent) {
      System.out.println(parent.age);
   }
   public static void main(String[] args) {
      new Main().test(_____);
   }
}
```

Boş (blank) hissəyə aşağıdakı bəndlərdən hansılar əlavə edildikdə kod kompayl olunacaqdır?

A. `new Parent()`
B. `new Child()`
C. `new Behaviour()`
D. `(Parent)new Object()`
E. `(Parent)new String()`
F. `null`

Bu da maraqlı kod nümunəsidir, sualla bağlı bilməli olduğunuz bütün nüanslar Chapter 5`in daxilində verilib. Əgər çətinlik çəksəniz, yaxşı olar ki, mövzunu yenidən oxuyasınız.

Nümunə 4:

```
1:  package objects;
2:  public class Car {
3:     protected void start(){}
4:     void stop(){}
5:  }

1:  package car;
2:  import objects.*;
3:  public class Kia extends Car {
4:     public static void main(String[] args) {
5:        Kia k = new Kia();
6:        k.start();
7:        k.stop();
8:        Car c = new Kia();
9:        c.start();
10:       c.stop();
11:    }
12: }
```

`Kia` classında hansı sətirlər kompayl xətası verir?

A. `5`
B. `6`
C. `7`
D. `8`
E. `9`
F. `10`

Sualın cavabı C, E və F bəndləridir. Siz isə səbəbini tapmağa çalışın. Bu sual ilə bağlı bilməli olduğunuz ən vacib fakt mövzunun daxilində izah edilib. Əgər həmin faktı xatırlamasanız, mövzuya təkrar bir də nəzər salın.

Nümunə 5:

```
1:   abstract class Grandma {
2:      public abstract int getAge();
3:   }
4:
5:   abstract class Grandpa {
6:      public int getAge();
7:   }
8:
9:   abstract interface Walk {}
10:
11:  class Grandchild extends Grandma, Grandpa implements Walk {
12:     int getAge() { return 3; }
13:  }
```

Yuxarıdakı kod nümunəsi ilə bağlı aşağıdakı fikirlərdən hansılar doğrudur?

A. Kod heç bir xəta baş vermədən kompayl olunur.
B. 2-ci sətirdə kompayl xətası baş verir.
C. 6-cı sətirdə kompayl xətası baş verir.
D. 11-ci sətirdə kompayl xətası baş verir.
E. 12-ci sətirdə kompayl xətası baş verir.
F. Kod kompayl olunur amma icra vaxtı exception baş verir.

Kod nümunəsində OOP-nin ən vacib prinsiplərindən biri pozulub, ona görə də kod kompayl olunmur, bu səbəbdən **A** və **F** doğru fikirlər deyil. Bildiyimiz kimi Java çoxvarisliliyi dəstəkləmir, amma *11-ci* sətirdə `Grandchild` classı `Grandma` və `Grandpa` classlarının ikisini də eyni anda varis almaq istəyir. Bu səbəbdən kompilyator xəta verir.

6-cı sətirdə də xəta baş verir, çünki `getAge()` metodunun gövdəsi yoxdur. Abstrakt class daxilində gövdəsiz metod elan edilərkən mütləq metodun adının əvvəlinə `abstract` açar sözü əlavə edilməlidir. `Grandpa` abstrakt class deyil, interfeys olsa idi, o zaman bu metod kompayl olunardı, çünki interfeysdə `abstract` açar sözünün yazılması məcburi deyil.

Kodda daha bir kompayl xətası var. Maraqlıdır ki, *12-ci* sətir də kompayl xətası var. Əslində kompayl xətası verməsi normaldır, metodların override edilməsi qaydalarında da qeyd etmişdik ki, child classdakı metodun access modifier`i parent classdakı metodla ən azı eyni olmalıdır, ya da ki, daha yüksək. Ona görə də *12-ci* sətirdə `getAge()` metodunun access modifier`i mütləq `public` olmalıdır. Sadəcə *11-ci* sətirdə çoxvarisliliklikdən dolayı xəta baş verdiyi üçün fikirləşmək olardı ki, `Grandchild` `Grandma` və `Grandpa` classlarını düzgün varis almadığı üçün ola bilsin `getAge()` metodunu override kimi görməsin. Amma sual qeyd edilən mənbədə (Sybex, Practise exam) də vurğulanıb ki, java kompilyator belə fikirləşmir, bunu da xəta kimi görür. Düzgün cavab – **C**, **D**, **E**

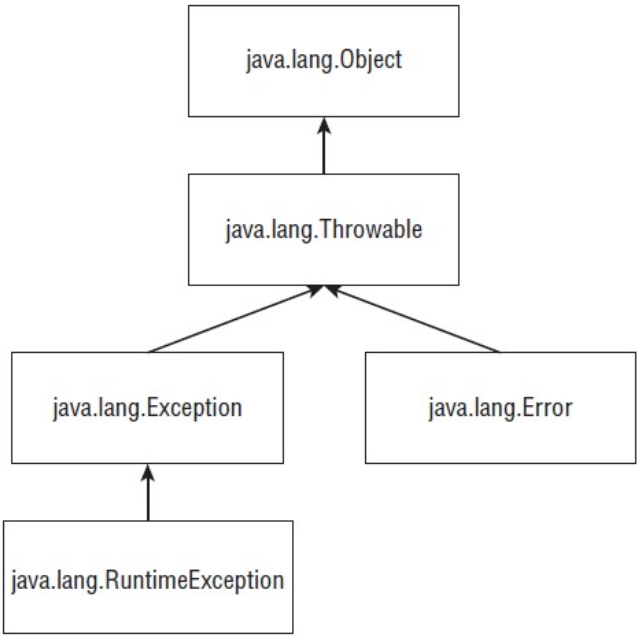
Chapter 6. Exceptions

Understanding Exception Types

Exception - proqramın axışını dəyişən hadisədir (event). Exception bir növü bir mexanizmdir, hansı ki onun köməkliyi ilə siz proqramda hər hansı bir gözlənilməz hal, xəta yarananda, baş vermiş xətanın məzmununa uyğun proqramın istiqamətinə istədiyiniz şəkildə yön verə bilərsiniz.

Bütün exceptionlar `Throwable` classından törəyir.

Şəkil 6.1 Exception`un iyerarxiyası



error - proqramda nələrinsə ciddi dərəcədə səhv getdiyini və proqramın onu bərpa etmək gücündə olmadığı anlamına gəlir. Misal üçün disk sürücüsünün "itməsini" göstərmək olar.

runtime exception - `RuntimeException` və ondan törəyən subclasslar vasitəsilə yaradılan exceptionlar hesab edilirlər. Runtime exceptionlar da gözlənilməyən istisnalar hesab edilir, amma errorlar kimi qorxulu deyil. Bu exceptionlar həm də *unchecked exceptions* adlanır.

checked exception - `Exception` və ondan törəyən bütün subclasslar (`RuntimeException` classı istisna olmaqla) bu exceptiona daxildir. Checked exceptionlar əvvəlcədən görülə bilən exceptionlar hesab edilir. Məsələn, olmayan bir faylı oxumağa cəhd etmək.

Bəs nə üçün bu exceptionlar *"checked"* adlanır? Çünki java bu exceptionların "handle" və ya "declare" olunmasını tələb edir. "handle" və "declare" ifadələrini biz tez-tez istifadə edəcəyik, ona görə də qısaca izah verək ki, "handle" – try-catch mexanizmi vasitəsilə xətanın yaxalanması, tutulması anlamına gəlir. "Declare" isə exceptionun metodda yenidən elan edilməsidir, yəni bu o anlama gəlir ki, bu metod elan etdiyi exceptionu öz üzərindən atır. Artıq bu metodu istifadə edən digər metodlar həmin exceptionu handle etməlidir, və yaxud onlar da yenidən həmin exceptionu declare edə bilər.

```
void test1(){
   throw new Exception();    // DOES NOT COMPILE
}
void test2() throws Exception{
   throw new Exception();
}
void test3(){
   throw new RuntimeException();
}
```

`throw` açar sözünü biz yeni exception fırlatmaq üçün istifadə edirik. `throws` açar sözü ilə isə elan edirik ki, bu metod exceptionu öz üzərindən onu çağıran metodun üzərinə atır. Yuxarıdakı nümunə ilə bağlı qısa olaraq qeyd etmək olar ki, checked exception`lar əvvəlcədən göründüyünə görə Java proqramçını məcbur edir ki, onunla bağlı bir tədbir görsün, ya handle etsin, ya da declare. Amma unchecked exception`lar belə desək qeyri-müəyyəndir, əvvəlcədən təyin etmək çətindir. Məsələn, `NullPointerException` `null` referans üzərindən hər hansı bir üzvü (member) çağırmağa cəhd edən zaman yaranır və bu da istənilən metodda baş verə bilər. Biz əgər buna görə hər yerdə bu exceptionu declare etməli olsaq, kodda qarmaqarışıqlıq yaranacaq. Bu səbəbdən Java unchecked (runtime) exception`ları handle və ya declare etməyi tələb etmir.

Throwing an Exception

Biz yeni exception fırladarkən mesaj olaraq `String` parametr də göndərə bilərik və yaxud da default olaraq parametrsiz:

```
throw new Exception();
```

```
throw new Exception("Hi! I am Exception");
throw new RuntimeException();
throw new RuntimeException("Hi! I am RuntimeException");
```

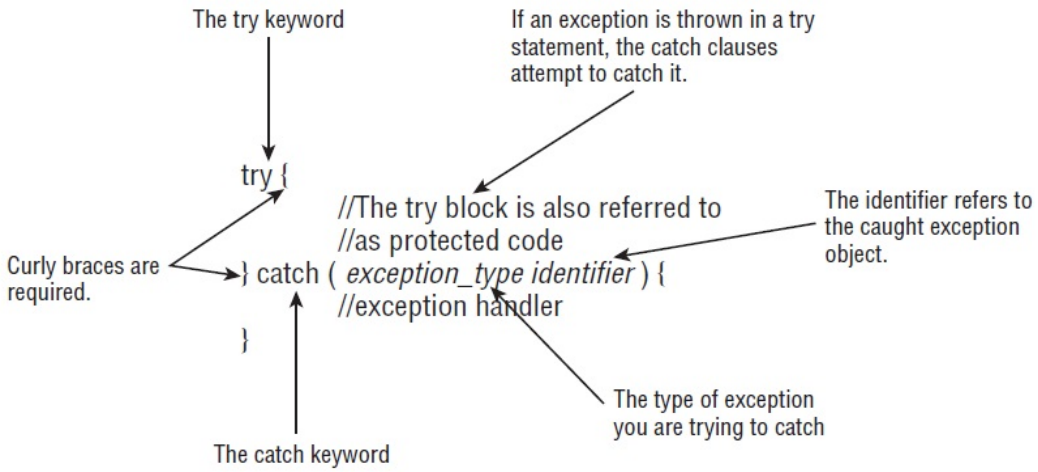
Aşağıdakı cədvəldə Exception`un növləri göstərilmişdir və burada qeyd olunan qaydaları bilmək vacibdir:

Type	How to recognize	Okay for program to catch	Is program required to handle or declare?
`Runtime exception`	Subclass of RuntimeException	Yes	No
`Checked exception`	Subclass of Exception but not subclass of RuntimeException	Yes	Yes
`Error`	Subclass of Error	No	No

Using a try Statement

Exceptionu handle (idarə) etmək üçün `try` ifadəsindən istifadə edilir.

Şəkil 6.2 try ifadəsinin sintaksisi



Try blokunda kodlar normal icra olunur. Əgər exception baş verərsə, bu zaman try blokunun icrası dayanır və icra catch blokuna keçir. Əgər try blokunda heç bir exception baş verməsə, o zaman catch bloku ("*block*" və ya "*clause*", hər ikisi işlədilə bilər) icra olunmur.

`try` ifadəsi də metodlar kimidir, `try` və `catch` bloklarının tərkibinə bir ifadə də daxil olsa belə yenə də mötərizə tələb edir:

```
try                    // DOES NOT COMPILE
   test3();
catch(Exception e)     // DOES NOT COMPILE
   System.out.println("I am catch block");
```

Kompayl olunması üçün mötərizələr əlavə olunmalıdır:

```
try {
   test3();
} catch(Exception e){
   System.out.println("I am catch block");
}
```

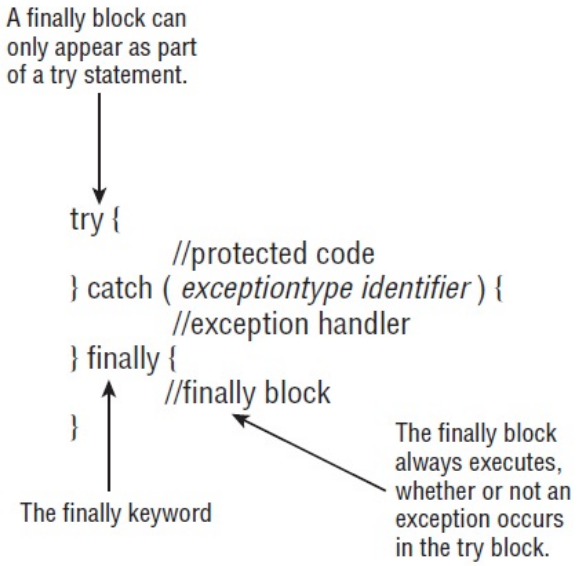
`try` bloku təklikdə işlədilə bilməz, ondan sonra ya `catch`, ya da `finally` bloku gəlməlidir:

```
try {          // DOES NOT COMPILE
   test3();
}
```

Adding a finally Block

`finally` bloku `try` ifadəsinin sonunda gəlir və sintaksisi belədir:

Şəkil 6.3 try ifadəsinin finally ilə birgə istifadəsi



Əgər exception baş verərsə, finally bloku catch blokundan sonra, yox əgər baş verməsə try blokundan sonra icra edilir. Amma catch blokundan fərqli olaraq finally bloku exception baş verib verməməsindən asılı olmayaraq həmişə icra edilir (bir istisnadan başqa).

Qeyd etdik ki, `try` bloku təklikdə işlədilə bilməz, ancaq *try-with-resources* `catch` və ya `finally` bloku olmadan təklikdə işlənə bilər. Lakin try-with-resources OCP imtahanının mövzusudur, OCA imtahanındakı nümunələrdə `try` bloku mütləq `catch` və ya `finally` bloku ilə birlikdə işlədilməlidir və catch bloku finally blokundan sonra gələ bilməz:

```
11:  try {
12:     test3();
13:  } finally {
14:     System.out.println("I am finally block");
15:  } catch(Exception e) {    // DOES NOT COMPILE
16:     System.out.println("I am catch block");
17:  }
18:
19:  try {   // DOES NOT COMPILE
20:     test3();
21:  }
22:
23:  try {
24:     test3();
25:  } finally {
26:     System.out.println("I am finally block");
27:  }
```

`finally` blok varsa, `catch` bloku tələb olunmur.

Biz qeyd etmişdik ki, finally blok həmişə icra olunur, amma bir istisna var. Əgər try və ya catch blokunda `System.exit(0);` çağırılarsa, proqram sonlanır və `finally` blok icra olunmur. Amma `System.exit(0);` -dən sonra yazılan kodlar *"unreachable code"* kimi kompayl xətası vermir, baxmayaraq ki, `System.exit(0);` -dən sonra yazılan heç bir kod icra olunmur.

Catching Various Types of Exceptions

İndiyə kimi yazdığımız nümunələrdə try ifadəsində ancaq bir catch blokundan istifadə etdik, amma try ifadəsində catch bloklarının sayına məhdudiyyət qoyulmur, istənilən sayda catch bloku istifadə edilə bilər, amma müvafiq qaydalara riayət etməklə. Ümumiyyətlə, OCA imtahanında təməl (basic) exceptionlarla bağlı suallar düşür, öz yaratdığınız exceptionlarla bağlı və digər suallar əsasən OCP imtahanında soruşulur. OCA imtahanı üçün exceptionlarla bağlı əsas 2 şeyi bacarmalıyıq:

1. Exception`un *checked* yaxud *unchecked* olduğunu təyin etməyi;

2. Hansı exceptionun hansından törədiyini, başqa sözlə hansının super, hansının sub class olduğunu ayırd etməyi.

```
class AnimalsOutForAWalk extends RuntimeException {}
class ExhibitClosed extends RuntimeException {}
class ExhibitClosedForLunch extends ExhibitClosed {}

class CatchException {
   public void visitPorcupine(){
      try {
         seeAnimals();
      } catch (AnimalsOutForAWalk e) {
         System.out.println("catch block-1");
      } catch (ExhibitClosed e) {
         System.out.println("catch block-2");
      }
   }
}
```

Yuxarıdakı nümunə üçün 3 mümkün hal var:

- ✓ exception baş verməsə heç bir catch bloku icra olunmur;
- ✓ `AnimalsOutForAWalk` exceptionu baş versə, ancaq 1-ci catch bloku icra olunacaq;
- ✓ `ExhibitClosed` exceptionu baş versə, ancaq 2-ci catch bloku icra olunacaq.

Catch bloklarının sıralaması ilə bağlı bilməli olduğumuz çox vacib bir qayda var: **catch blokunda exceptionlar sub classdan super classa doğru sıralanmalıdır.** Çünki catch bloklarının sayından asılı olmayaraq, istənilən halda ancaq bir catch bloku icra olunur və bu baş verən exceptiona uyğun gələn *birinci* catch bloku olur. Ona görə də əgər super class sub classdan əvvəl gələrsə, subclassa uyğun gələn exceptionu artıq super class yaxalayacaq və sub classın super classdan sonra gəlməsinin heç bir əhəmiyyəti qalmayacaq. Bu səbəbdən super classdan sonra gələn sub class "*unreachable statement*" hesab edilir və kompayl olunmur.

Yuxarıdakı nümunədə `AnimalsOutForAWalk` və `ExhibitClosed` exceptionlarının sırasını dəyişdirsək, heç bir xəta baş vermədən kod yenə kompayl olunacaqdır. Çünki bu exceptionlar bir-birindən törəməyib, hər ikisi `RuntimeException` classından törədiyinə görə paralel hüquqlu sayılır, ona görə də yerlərini dəyişdirmək mümkündür.

Aşağıdakı nümunədə isə subclass-superclass ardıcıllığına riayət olunur:

```
public void visitMonkeys() {
   try {
      seeAnimals();
   } catch (ExhibitClosedForLunch e) {  // subclass exception
      System.out.println("subclass");
   } catch (ExhibitClosed e) {          // superclass exception
      System.out.println("superclass");
```

```
    }
}
```

Yuxarıdakı nümunədə əgər subclassla superclass exceptionun yerini dəyişsək kompayl xətası alacağıq.

Sıralamanın pozulması ilə bağlı başqa bir nümunəyə baxaq:

```
public void visitSnakes() {
    try {
        seeAnimals();
    } catch (RuntimeException e) {
        System.out.println("runtime exception");
    } catch (ExhibitClosed e) {    // DOES NOT COMPILE
        System.out.println("exhibit closed exception");
    } catch (Exception e) {
        System.out.println("exception");
    } catch (Exception e) {        // DOES NOT COMPILE, because of duplicate
        System.out.println("exception");
    }
}
```

Throwing a Second Exception

Biz indiyə kimi yazdığımız nümunələrdə ancaq bir `try` ifadəsi işlətdik. Amma `try` blokunun içində, eləcə də `catch` və `finally` bloklarının içində də digər bir `try` ifadəsi işlədə bilərik:

```
import java.io.*;
public class FileReaderExc {
    public static void main(String[] args) {
        FileReader reader = null;
        try {
            reader = read();
        } catch (IOException e) {
            try {
                if (reader != null) reader.close();
            } catch (IOException inner) {
            }
        }
    }

    private static FileReader read() throws IOException {
        // Code goes here #line1
    }
}
```

`reader` obyekti ilə işimiz bitdikdən sonra onu bağlamalıyıq, bunun üçün `close()` metodunu çağırırıq. `close()` metodunun özü də `IOException` fırladır və `IOException` checked exception olduğundan onu da try-catch blokuna salıb handle edirik.

Əgər catch və finally bloklarının hər ikisində də exception fırladılarsa, fırladılan son exception olaraq **finally blokundakı exception götürülür, catch blokundakı exception gizlədilir:**

```
20:  public static void test(){
21:     try {
22:        throw new RuntimeException();
23:     } catch (Exception e) {
24:        throw new Exception();
25:     } finally {
26:        throw new RuntimeException();   // the last exception
27:     }
28:  }
```

Bu metodu çağırsaq `RuntimeException` baş verəcək. Əgər finally blok olmasa, o zaman kompilyator fırladılan son exception olaraq sətir 24-ü qəbul edəcək və xəta verəcək. Çünki Exception ckecked exception qrupuna aiddir, ona görə də ya handle olunmalıdır, ya da declare. Amma sətir 26-ya görə artıq sətir 24 unudulur və son exception olaraq `RuntimeException` qəbul edilir.

Sonda qeyd etdiklərimizi yekunlaşdıraraq daha bir nümunəyə baxaq və metodun geriyə qaytardığı dəyərə diqqət edək:

```
30:  public String exceptions() {
31:     String result = "";
32:     String v = null;
33:     try {
34:        try {
35:           result += "before ";
36:           v.length();
37:           result += "after ";
38:        } catch (NullPointerException e) {
39:           result += "catch ";
40:           throw new RuntimeException();
41:        } finally {
42:           result += "finally ";
43:           throw new Exception();
44:        }
45:     } catch (Exception e) {
46:        result += "done ";
47:     }
48:     return result;
49:  }
```

Metodun geri qaytardığı dəyər:

```
before catch finally done
```

Daxildəki try ifadəsinin fırlatdığı son exception (`Exception`) checked exceptiondur və həmin try ifadəsi digər bir try ifadəsinin içində yazıldığından baş verən exception çöldəki (outer) try ifadəsinin catch blokunda tutulur, ona görə də kod kompayl olunur. Əgər sətir 45-də `Exception`u `RuntimeException` etsək, sətir 43 kompayl olunmayacaq.

Bu kodla bağlı bəzi dəyişikliklər edib, qayıdan dəyərin necə dəyişəcəyinə diqqət yetirək:

1. Əgər sətir 40 və 43-ü kommentə salsaq, o zaman nəticə belə olacaqdır: `before catch finally`
2. Əgər sətir 40-ın əvəzinə `return result;` yazsaq və sətir 43-ü kommentə salsaq, o zaman nəticə belə olacaqdır: `before catch`
   Sırf bu nüansla bağlı bir nümunəni Exception mövzusunun əlavələrində qeyd etmişəm. Orada qeyd olunan linkdən bunun niyə belə baş verməsi ilə bağlı daha ətraflı məlumat ala bilərsiniz.
3. Əgər yalnız sətir 40-ın əvəzinə `return result;` yazsaq, o zaman nəticə belə olacaqdır: `before catch finally done`
   Belə görünür ki, `catch` blokunda `return` ifadəsi işlədilsə də `finally` blokunda yeni exception fırladılırsa, kodun axışı dəyişir, `return` ifadəsi yazılmasına baxmayaraq nəzarət artıq daxildəki `catch` blokundan çıxır.

Runtime Exceptions

Runtime exception`lar `RuntimeException` classından törənmiş exception`lardır. Bu exception`lar handle və declare olunmağı tələb etmir. Həm proqramçı, həm də JVM tərəfindən fırlada bilər. Ən çox istifadə edilən runtime exception`lar bunlardır:

ArithmeticException

Sıfıra bölmə əməliyyatına cəhd zamanı baş verir və JVM tərəfindən fırladılır.

```
int result = 7 / 0;
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

ArrayIndexOutOfBoundsException

Massivin doğru indeksinə müraciət etmədikdə baş verir və JVM tərəfindən fırladılır.

```
int arr[] = new int[3];
for(int i=0; i<=arr.length; i++) {
```

```
        System.out.print(arr[i] + " ");
    }
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

ClassCastException

Uyğun olmayan tipləri bir-birinə cast edən zaman yaranır və JVM tərəfindən fırladılır. Bəzən kompilyator qeyri-mümkün castların qarşısını alır:

```
String type = "number";
Integer number = (Integer)type;  // DOES NOT COMPILE
```

`Integer` `String`in subclassı deyil, ona görə də kompayl xətası baş verir. Əgər biz `String` referansını əvvəlcə `Object` tipinə, sonra `Integer` tipinə cast etsək kompayl xətası baş verməyəcək, amma `ClassCastException` baş verəcək:

```
String type = "number";
Object obj = type;
Integer number = (Integer)obj;
```

Output:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
be cast to java.lang.Integer
```

IllegalArgumentException

Çox vaxt metoda düzgün parametr göndərməyəndə baş verir və **proqramçı** tərəfindən fırladılır. Tutaq ki, bir setter metodumuz var və mənfi parametr qəbul edə bilməz:

```
5:  public void setCountStudents(int countStudents){
6:     if(countStudents >= 0 )
7:        this.countStudents = countStudents;
8:  }
```

Bu kod normal işləyir, lakin mənfi parametr göndərdikdə requestdən imtina (ignore) edilir və bu metodu çağıranın (caller) bundan xəbəri olmur. Amma biz istəyirik ki, baş verən problem barədə proqramçını xəbərdar edək və o, bu problemi həll etsin. Bu zaman `IllegalArgumentException` istifadə etmək tam məqsədəuyğun olur:

```
public void setCountStudents(int countStudents){
   if(countStudents < 0 )
      throw new IllegalArgumentException("# count must not be negative");
   this.countStudents = countStudents;
}
```

Əgər metoda mənfi parametr göndərilsə, o zaman aşağıdakı kimi xəbərdarlıq çıxacaq:

```
Exception in thread "main" java.lang.IllegalArgumentException: # count must not
be negative
```

NullPointerException

Instance dəyişən və metodlar `null` referans üzərindən çağırılan zaman baş verir və JVM tərəfindən fırladılır.

```
String name;
public void printLength() throws NullPointerException {
   System.out.println(name.length());
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException
```

Başqa bir nümunəyə baxaq:

```
class TestClass {
   public static void main(String args[]) throws Exception {
      Exception e = null;
      throw e;  // throws NullPointerException
   }
}
```

NumberFormatException

Uyğun formatda olmayan String`i numeric tipə çevirən zaman baş verir və **proqramçı** tərəfindən fırladılır. `NumberFormatException` classı `IllegalArgumentException`un subclassıdır.

```
Integer.parseInt("num");
```

Output:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "num"
```

Checked Exceptions

`Exception` və ondan törəyən bütün subclasslar (`RuntimeException` classı istisna olmaqla) bu exceptiona daxildir. Mütləq handle və ya declare olunmalıdırlar. Proqramçı və ya JVM tərəfindən fırladıla bilər. Ən çox istifadə edilən checked exceptionlar bunlardır:

IOException

Faylın oxunması və ya yazılması zamanı problem baş verdikdə yaranır (proqramçı tərəfindən fırladılır).

FileNotFoundException

Mövcud olmayan fayla müraciət etməyə çalışdıqda baş verir. `IOException`un subclassıdır (proqramçı tərəfindən fırladılır).

OCA imtahanı üçün bu məlumatları bilmək yetərlidir. Bu iki exceptionu detallı şəkildə OCP imtahanında görəcəksiniz.

Errors

Error`lar `Error` classından törəyir və JVM tərəfindən fırladılır. Declare və ya handle olunmağı məsləhət görülmür. Errorlar nadir hallarda baş verir, OCA imtahanında əsas bu errorları görə bilərsiz:

ExceptionInInitializerError

Static initializer blokda exception baş verdikdə və handle olunmadıqda yaranır və JVM tərəfindən fırladılır. `ExceptionInInitializerError` ona görə error hesab olunur ki, Java bütünlüklə classın yüklənməsini dayandırır. Çünki classa müraciət olunan zaman ilk olaraq static initializer blok işə düşür və əgər həmin blokda exception baş verirsə, Java classdan istifadəyə başlaya bilmir.

```
public class ExceptionInIntializerError {
   static {
      int[] numbers = new int[3];
      int num = numbers[-1];
   }
   public static void main(String[] args) {}
}
```

Output:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
```

Exception static blokda baş verdiyindən `ExceptionInInitializerError` alırıq. Amma bu errorla bağlı məlumat problemi həll etmək üçün yetərli olmadığından static blokda baş vermiş exceptionla bağlı məlumat da əlavə olaraq göstərilir.

StackOverflowError

Əsasən metod özü özünü çağıraraq sonsuz dövrə düşdüyündə (*infinite recursion*) baş verir və JVM tərəfindən fırladılır:

```
public static void recursiveMethod(int i){
```

```
    recursiveMethod(7);
}
```

Output:

```
Exception in thread "main" java.lang.StackOverflowError
```

Metod təkrar-təkrar özünü çağırır və sonlanmır. Nəhayət, stack ehtiyatlar tükənir və error baş verir. Bu "infinite recursion" adlanır. Infinite recursion infinite loop`a nisbətən daha "yaxşı" hesab olunur, çünki Java sonunda onu yaxalayır və error verir. Amma infinite loop onu "kill" etməyənə qədər bütün CPU resurslarını istifadə edir.

Başqa bir maraqlı nümunəyə baxaq:

```
class Recursive {
    // Recursive obj = new Recursive();      // throws StackOverflowError
    static Recursive obj = new Recursive();  // no exceptions
    int[] ia = new int[1000];
    public static void main(String[] args) {
        Recursive ro = new Recursive();
    }
}
```

NoClassDefFoundError

Kompayl vaxtı mövcud olan classın icra vaxtı (runtime) tapılmaması zamanı baş verir və JVM tərəfindən fırladılır. Bu error imtahanda kod daxilində istifadə olunmur, sadəcə bilməliyik ki, bu errordur.

Bu errorlarla bağlı daha ətraflı məlumat üçün aşağıdakı linkə baxa bilərsiniz:

http://www.coderanch.com/t/655737/ocajp/certification/ExceptionInInitializerError-NoClassDefFoundError

Calling Methods That Throw Exceptions

Əgər metod exception fırladırsa, həmin metodu çağıran zaman bəzi qaydalara riayət edilməlidir.

```
class NoCarrotException extends Exception {}

public class Bunny {
    public static void main(String[] args) {
        eatCarrot();  // DOES NOT COMPILE
    }
    private static void eatCarrot() throws NoCarrotException {}
}
```

Bu kod ona görə kompayl xətası verir ki, `NoCarrotException` checked exceptiondur. Checked exceptionlar mütləq handle və ya declare olunmalıdır. Əgər main metodu aşağıdakı formalarda dəyişsək, kod kompayl olunacaqdır:

```
public static void main(String[] args) throws NoCarrotException { // declare
   eatCarrot();
}
```

yaxud

```
public static void main(String[] args) {
   try {
      eatCarrot();
   } catch (NoCarrotException e) { // handle
      System.err.println(e);
   }
}
```

Fikir versək görərik ki, əslində `eatCarrot()` metodunda exception baş vermir, sadəcə `NoCarrotException` declare olunub. Elə bu kifayət edir ki, bu metodu hər hansı bir metodda çağırdıqda kompilyator handle və ya declare tələb etsin.

Metodun `throws` bölməsində ancaq `java.lang.Throwable` classından törəmiş classlar yazıla bilər.

Diqqət etməli olduğumuz daha bir vacib qayda var. Əgər hər hansı bir metodda checked exception fırladılmır və ya declare olunmursa, həmin metodu digər metod daxilində çağırdıqda həmin checked exception'u handle etmək olmaz, çünki *"try blokunda belə bir exception baş vermir"* xətası verəcək və kompayl olunmayacaq. Amma declare etmək mümkündür. Nümunə üzərindən baxaq:

```
public void bad() {
   try {
      eatCarrot();
   } catch (NoCarrotException ex) {   // DOES NOT COMPILE
      System.err.println(ex);
   }
}
public void good() throws NoCarrotException {
   eatCarrot();
}
private static void eatCarrot() {}
```

`eatCarrot()` metodunda checked exception baş vermədiyindən kompilyator başa düşür ki, `bad()` metodunda heç bir halda catch blokuna çatmaq (reach) mümkün deyil və ona görə də xəta verir. Amma `good()` metodu istənilən exceptionu declare etməkdə sərbəstdir.

`bad()` metodunun catch blokunda `NoCarrotException`u `Exception` və ya `RuntimeException` ilə əvəz etsək kod kompayl olunacaqdır.

Subclasses

Metod overridinglə bağlı 3-cü qayda checked exceptionlar ilə əlaqəli idi. Həmin qaydalara yenidən baxaq və görək *checked exceptionlar* üçün hansı hallar mövcuddur:

1) Əgər superclassdakı metod heç bir exception fırlatmırsa, subclassdakı metod yeni checked exception fırlada bilməz:

```
class NewException extends Exception {}

class Hopper {
   public void hop() {}
}

class Bunny extends Hopper {
   public void hop() throws NewException {}   // DOES NOT COMPILE
}
```

2) Yuxarıda qeyd etdiyimiz 1-ci halın tərsi mümkündür. Subclassdakı metod exception fırlatmasa belə, superclassdakı metod exception fırlada bilər:

```
class NewException extends Exception {}

class Hopper {
   public void hop() throws NewException {}
}

class Bunny extends Hopper {
   public void hop() {}
}
```

3) Subclassdakı metodun fırlatdığı exception superclassdakı metodun fırlatdığı exceptionun subclassı olmalıdır; daha kiçik və ya eyni tipə icazə verilir:

```
class NewException extends Exception {}

class Hopper {
   public void hop() throws Exception {}
}

class Bunny extends Hopper {
   public void hop() throws NewException {}
}
```

Təkrar olaraq vurğulayaq ki, bu qaydalar ancaq checked exceptionlara aiddir. Runtime exceptionlarla bağlı heç bir məhdudiyyət yoxdur:

```
class NewException extends Exception {}

class Hopper {
   public void hop() {}
}

class Bunny extends Hopper {
   public void hop()throws IllegalStateException {}
}
```

Printing an Exception

Exceptionu print etməyin əsas 3 yolu var:

```
7:   public static void main(String[] args) {
8:      try {
9:         hop();
10:     } catch(Exception e){
11:        System.out.println(e);                  // first way
12:        System.out.println(e.getMessage());     // second way
13:        e.printStackTrace();                    // third way
14:     }
15:  }
16:  private static void hop(){
17:     throw new RuntimeException("can not hop");
18:  }
```

Output:

```
java.lang.RuntimeException: can not hop                    // first way
can not hop                                                // second way
java.lang.RuntimeException: can not hop                    // third way
    at chapter6.PrintingException.hop(Handling.java:17)
    at chapter6.PrintingException.main(Handling.java:9)
```

Praktikada həmişə catch blokunda exceptionla bağlı bildirişin çap edilməsi məsləhət görülür. Nümunə üçün aşağıdakı koda baxaq:

```
import java.io.IOException;
class ExceptionMessage {
   public static void main(String[] args) {
      String textInFile = null;
      try {
         readInFile();
```

```
        } catch (IOException e) {
            // ignore exception message
        }
        // many lines of code
        System.out.println(textInFile.replace(" ", ""));
    }
    private static void readInFile() throws IOException {
        throw new IOException();
    }
}
```

Bu kodu çalışdırsaq `NullPointerException` verəcək, amma `IOException`un baş verməsi ilə bağlı heç bir xəbərimiz olmayacaq.

Əlavələr

Coderanch forumundan götürülmüş maraqlı bir nümunəyə baxaq və özümüzü yoxlayaq. Aşağıdakı kod nümunəsi verilmişdir:

```
class FoodException extends Exception {}
class VeggieException extends FoodException {}
class Animal {
   public void eat() throws FoodException { System.out.println("animal"); }
}
class Elephant extends Animal {
   public void eat() { System.out.println("elephant"); }
}
class Tiger extends Animal {
   public void eat() throws VeggieException { System.out.println("tiger"); }
}

class Test {

   public void method1() {
      Animal a = new Animal();
      try {
         a.eat();
      } catch (FoodException e) {}
   }

   public void method2() {
      Animal a = new Elephant();
      a.eat();
   }

   public void method3() {
      Tiger t = new Tiger();
      try {
         t.eat();
      } catch (VeggieException e) {}
   }

   public void method4() {
      Elephant e = new Elephant();
      ((Animal)e).eat();
   }

   public void method5() throws Exception {
      Animal a = new Animal();
      ((Tiger) a).eat();
   }
```

```
    public void method6() throws FoodException {
       Animal a = new Elephant();
       ((Elephant)a).eat();
    }

    public void method7() {
       Animal a = new Elephant();
       try {
          ((Elephant)a).eat();
       } catch (FoodException e) {}
    }
}
```

Deməli `Test` classında 7 metod verilib. Bizdən nəyi tapmaq tələb olunur? `Test` classında verilmiş hansı metodlar:

1) kompayl olunmur?
2) kompayl olunur, amma icra vaxtı exception verir?
3) kompayl olunur və icra vaxtı heç bir exception baş vermir?

Cavab:

1) `method2(), method4(), method7()`
2) `method5()`
3) `method1(), method3(), method6()`

`method4()` -də verilmiş cast ifadəsi:

```
((Animal) e).eat();
```

aşağıdakı kod ilə ekvivalentdir:

```
Animal a = (Animal) e;
a.eat();
```

`method5()` və `method6()` -dakı cast ifadələri də yuxarıdakı məntiqə əsasən çalışır.

Sualla bağlı ətraflı izah aşağıdakı linkdə verilmişdir:

http://www.coderanch.com/t/651798/ocajp/certification/Mock-exam-wrong-answer-Java

Başqa bir nümunəyə baxaq:

```
import java.io.*;
class MyException extends FileNotFoundException {}
class TestExc {
  public static void main(String[] args) throws FileNotFoundException { // line1
     try {
        if(true)                      // line2
           throw new MyException();   // line3
```

```
            System.out.println("try ");   // line4
        } catch (RuntimeException e){
            System.out.println("catch ");
        } finally {
            System.out.println("finally ");
        }
    }
}
```

line3-də `MyException` əvəzinə `IOException` yazsaq, kod kompayl olunmayacaq. Çünki `catch` bloku `IOException`u tuta bilmir. Həm də *line1*-də declare olunan exception `IOException`'dan daha kiçik tip olduğuna görə onu öz üzərindən fırlada bilməyəcək. *line2*-ni kommentə atsaq kod yenə kompayl olunmayacaq, çünki *line4* "unreachable" kod hesab olunacaq.

Bir birindən törəmə classlar catch blokunda aşağıdakı şəkildə birgə işlədilə bilməz, ancaq paralel olanlara icazə verilir:

```
try{
}catch(ArrayIndexOutOfBoundsException | NullPointerException | RuntimeException e)
{ }  // DOES NOT COMPILE

try{
}catch(ArrayIndexOutOfBoundsException|NullPointerException|ClassCastException e){}
```

Biz qeyd etdik ki, static blokda exception baş verərsə, bu zaman JVM tərəfindən `ExceptionInInitializerError` fırladılır. Amma static və instance initializer bloklarda birbaşa olaraq throw ilə exception fırlatdıqda kompilyator bunu qəbul etmir və xəta verir:

```
public class Test {
    static {   // DOES NOT COMPILE, initializer must be able to complete normally
        throw new NullPointerException();
    }

    {  // DOES NOT COMPILE, initializer must be able to complete normally..
       // həm də unreachable statement xətası verir. Əgər static blokla bu blokun
       // yerini dəyişsək yenə də unreachable statement xətası bu blokda baş verəcək,
       // çünki kompilyator bilir ki, 1-ci static blok icra edilir..
        throw new ClassCastException();
    }
}
```

Əgər `if(true)` şərtini əlavə etsək, o zaman xəta aradan qalxacaq, çünki kompilyator `if` blokunun icra olunub olunmayacağını ancaq icra vaxtı müəyyən edir:

```
static {
    if(true)
        throw new NullPointerException();
}
```

Amma `if` ilə yazdıqda belə ancaq `RuntimeException` fırlatmağa icazə verir, checked exception fırlatsaq kompayl xətası alacağıq:

```
static {
   if (true)
      throw new IOException();  // DOES NOT COMPILE
}
```

List/massivlərlə əlaqəli baş verəcək xətalarla bağlı bir nümunəyə baxaq:

```
public static void main(String[] args) {
   int[] ia = new int[]{1, 2, 3};
   System.out.println(ia[-1]);   // throws ArrayIndexOutOfBoundsException

   int[] arrNegativeSize = new int[-5];  // throws NegativeArraySizeException
   List l = new ArrayList(-2); //throws IllegalArgumentException: Illegal Capacity: -2
}
```

Facebook qrupumuzda bu mövzu ilə bağlı çox maraqlı bir sual paylaşılmışdı, "screen" edib qeydlərdə saxlamışdım:

```
class JavaClass {

   byte incrementAndGet() {
      byte b = 0b01110; // line 3
      try {
         String[] students = {"C", "C++", "Java", "JavaScript", "Python"};
         System.out.println(students[5]);
      } catch (Exception ex) {
         System.out.println("value of b in catch: " + b);
         return b;
      } finally {
         b++;
         System.out.println("value of b in finally: " + b);
      }
      return b; // line 7
   }

   public static void main(String[] args) {
      int i = new JavaClass().incrementAndGet(); // line 10
      System.out.println("Returned value in main() method: " + i);
   }
}
```

Which of the following statements are true? (Choose all that apply)

A) value of b in catch: 14
value of b in finally: 15
Returned value in main() method: 14

B) value of b in catch: 14
value of b in finally: 15
Returned value in main() method: 15

C) value of b in catch: 14
value of b in finally: 14
Returned value in main() method: 15

D) value of b in catch: 14
value of b in finally: 14
Returned value in main() method: 14

E) Compiler error at line 3

F) Compiler error at line 7

G) Compiler error at line 10

H) RuntimeException

I) None of above

Düzgün cavaba və müzakirələrə aşağıdakı linkdən baxa bilərsiniz, amma əvvəlcə cavabı özünüz tapmağa çalışın:

https://www.facebook.com/groups/javacertification/permalink/902142646501541/

`System.exit(0);` ilə bağlı bir nümunəyə baxaq. Aşağıdakı kodu icra etdikdə ekrana nə çap ediləcək:

```
class A {

   private String value;

   public void go() {
      try {
         System.out.println(value.toString());
         System.out.println("1");
      } catch (NullPointerException e) {
         System.out.println("2");
         System.exit(0);
      } finally {
         System.out.println("3");
      }
      System.out.println("4");
   }

   public static void main(String[] args) {
      new A().go();
   }
}
```

Deməli, `value`nin default dəyəri `null`dur, ona görə də `try` blokunda `NullPointerException` baş verir və `catch` blokunda tutulur. `2` çap edilir və sonra `System.exit(0);` ifadəsi çağırılır. Bu ifadə çağırıldığına görə ondan sonrakı heç bir kod icra edilmir, hətta `final` bloku da. Output: `2`

Sonuncu chapter olduğuna görə burada paylaşmağı uyğun bildim. Deməli, məqalənin adı *"8 type of questions in OCA Java SE 8 certification exam"* olaraq qeyd edilib, maraqlı yazıdır, baxmaqda fayda vardır:

http://ejavaguru.com/examquestiontypes.html

Bölmə 3. İmtahan Təcrübəm

- Hazırlığa necə başladım və hansı kitabdan istifadə etdim?
- Kitabı bir dəfə oxuyub bitirdikdən sonra hansı qərara gəldim?
- Hansı test bankından istifadə etdim?
- İmtahan günü – 94% nəticə ilə imtahanı keçdim
- Jeanne Boyarsky və Scott Selikoff'un kitabı haqqında fikirlərim
- Enthuware test bankı haqqında qeydlərim
- Coderanch forumu haqqında düşüncələrim
- Real imtahan sualları haqqında qısa qeydlərim
- SYBEX və Enthuware testlərində rastlaşmadığım sual tipləri
- Məni ən çox çaşdıran 4 sual

Hazırlığa necə başladım və hansı kitabdan istifadə etdim?

İlk öncə vurğulamaq istəyirəm ki, mən ixtisasca iqtisadçıyam və sertifikat imtahanına hazırlığa başlayana qədər java proqramlaşdırma dili üzrə artıq 2 il təcrübəm var idi. Hər şey Jeanne Boyarsky və Scott Selikoff`un *"OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808"* (bundan sonra SYBEX) kitabını aldıqdan sonra başladı. İlk günlər yaşadığım çətinliklər barədə birinci bölmədə "Addım 2. Hansı kitabdan hazırlaşmalı?" başlığında qeyd etmişdim. Deməli, ingilis dili səviyyəm qənaətbəxş deyildi, ilk günlər 1 saata 1 səhifə ancaq oxuyurdum, lüğətdən istifadə edə-edə. Gün ərzində təxminən 7-8 saat oxumağa vaxt ayırırdım və bu müddət ərzində maksimum 8-10 səhifə oxuya bilirdim. Daha yaxşı yadımda qalması üçün kitabdakı bütün kod nümunələrini yenidən IDE və ya Notepad`də yazaraq test (compile&run) edirdim.

Əvvəlcə *"Introduction"*, sonra isə "AppendixB" bölməsini oxudum. Kitabda *"Assessment Test"* bölməsi var idi, 20 sualdan ibarət idi, mövzulara başlamazdan əvvəl cari biliklərinizi yoxlamaq üçün nəzərdə tutulmuşdu. Demək olar ki, 3-4 sualı çətinliklə cavablandırmışdım, digər sualları isə bilməmişdim. Və başlanğıcda bu cür hallarla rastlaşmaq təəssüf ki, insanda inamsızlıq yaradır, istər-istəməz xoş olmayan təəssüratlar formalaşdırır. Ümumiyyətlə, haşiyəyə çıxaraq qeyd edim ki, imtahanla bağlı indiyə qədər mənə gələn mesajlardan, ismarıclardan belə hiss etmişəm ki, insanların çoxunda sertifikat almaq üçün istək olsa da, bacarmama qorxusu və inamsızlıq var. Bəlkə də bu səbəbdən yoldaşların çoxu hazırlığı yarımçıq saxlayır. Ona görə də motivasiya üçün vurğulamaq istəyirəm ki, mənə də təzə-təzə hər şey çox çətin gəlirdi. Mənim də içimdə bir qorxu var idi ki, onsuz da keçə bilməyəcəm, əbəs yerə əziyyət çəkməyim, elə bu başdan hər şeyi dayandırım. Çünki gələcəkdə, itirəcəyim vaxt və çəkəcəyim əziyyət ilə əlaqədar yarana biləcək hər hansı bir peşmançılıq hissi rahat buraxmırdı. Amma indiyə qədər elə bir hal olmamışdı ki, əziyyət çəkim və müsbət nəticəsini görməyim. Həm də sertifikatı ürəkdən istəyirdim, qarşıma məqsəd qoymuşdum və özümə söz vermişdim, ona görə də başqa çıxış yolum yox idi, davam etməli idim. Bu tərəddüdlər bir ay boyunca davam elədi, hər dəfə də ümidsizliyə qapılanda, "onsuz da başqa çıxış yolum yoxdur" deyib oxumağa davam edirdim. Bir aydan sonra yavaş-yavaş inam formalaşmağa başladı. Ona görə də istənilən halda davam etmək lazımdır, yarımçıq saxlamaq olmaz.

Kitab 6 chapter`dən ibarətdir və hər chapter`in sonunda mövzunu nə dərəcədə mənimsəyib-mənimsəmədiyinizi yoxlamaq üçün testlər var. Kitabı bir dəfə oxuyub bitirdikdən sonra həmin testlər üzrə nəticələrim aşağıdakı kimi olmuşdu (vaxt limiti tətbiq etmədən):

- Chapter 1 – 74%
- Chapter 2 – 55%
- Chapter 3 – 66.7%
- Chapter 4 – 62.1%
- Chapter 5 – 60%
- Chapter 6 – 60%

Sualları cavablandırmaq üçün kifayət qədər çox vaxt sərf edirdim. Amma real imtahanda orta hesabla bir sual üçün 1.5-2 dəqiqə vaxt ayrılır. Sertifikat suallarının formatı adətən maksimum şəkildə yığcamlaşdırılmış, qısadılmış şəkildə olur, kod müəyyən qədər qarmaşa şəklində görünür. Gözün koda öyrəşməsi, kodun təhlil edilməsi istər-istəməz əlavə vaxt alır. Vaxtımı effektiv şəkildə idarə etməyi öyrənməli idim. Ona görə də Coderanch forumundakı bəzi təcrübəli istifadəçilərin məsləhəti ilə Chapter 4`dən başlayaraq IDE istifadə etməyi dayandırdım və Notepad istifadə etməyə başladım. Artıq kodları notepad`də yazaraq cmd`də `javac/java` ilə test (compile&run) edirdim. Bu üsul müəyyən qədər səbr tələb etdiyindən ilk günlər çox çətin gəlirdi. Amma sonra öyrəşdikcə zövq almağa başladım və çox böyük faydasını gördüm. Artıq massiv ilə `length`, `String` ilə `length()` işləndiyini yadda saxlaya bildim.

Kitabı bir dəfə oxuyub bitirdikdən sonra hansı qərara gəldim?

Kitabda məsləhət görülür ki, əgər *"Review questions"*lardan nəticəniz **80%**-dən az olarsa, həmin mövzunu yenidən oxuyun. Testləri vaxt limitini nəzərə almadan etsəm də yenə bütün nəticələrim **80%**-dən az olmuşdu. Ona görə də kitabı yenidən oxumalı idim. Amma kitabı təkrar oxumağa başlamazdan öncə əlavə testlər etməyi qərara aldım. Çünki testlərdə ən çox hansı məqamlara toxunulduğunu görürsən və artıq mövzunu təkrar oxumağa başlayanda hansı hissələrə daha çox fokuslanmalı olduğunu bilirsən. Bundan sonra müvafiq olaraq aşağıdakı testləri etdim:

"OCA Java SE 7 Programmer I Certification Guide", *Mala Gupta,* Sample exam questions:

- Chapter 1 – 63.6%
- Chapter 2 – 70%
- Chapter 3 – 50%
- Chapter 4 – 45.5%
- Chapter 5 – 60%
- Chapter 6 – 70%
- Chapter 7 – 70%
- Full Mock Exam – 82.2%

"OCA/OCP Java SE 7 Programmer I & II Study Guide", *Kathy Sierra* and *Bert Bates,* Self Test:

- Chapter 1 – 88.9%
- Chapter 2 – 75%
- Chapter 3 – 84.6%
- Chapter 4 – 87.5%
- Chapter 5 – 75%
- Chapter 6 – 80%

Testləri bitirdikdən sonra SYBEX kitabını ikinci dəfə oxumağa başladım və vacib hissələrini qeyd etməyi qərara aldım. Kitabı təkrarlayıb bitirdim və Chapter testləri yenidən etdim, amma bu dəfə "online material"dan. Kitabın "Online material" adlı ikinci hissəsi də var və chapter testlər orada da mövcuddur. Testi online materialdan etməyin üstünlüyü ondan ibarətdir ki, orada hər sual üçün sərf etdiyin vaxt da qeyd edilir və sualı düzgün cavablandırmaqdan əlavə, təyin edilmiş vaxt ərzində cavablandırıb cavablandıra bilmədiyini də müəyyən etmiş olursan. Chapter testlər üzrə 2-ci cəhd nəticələrim aşağıdakı kimi oldu (təqribən 3 ay sonra və bu dəfə vaxt ilə):

İkinci cəhd	Birinci cəhd ilə müqayisə
Chapter 1 – 91.3%	74%
Chapter 2 – 100%	55%
Chapter 3 – 84.85%	66.7%
Chapter 4 – 93.1%	62.1%
Chapter 5 – 70%	60%
Chapter 6 – 90%	60%
Assessment test – 65%	

Yay məzuniyyətinə gedəndə noutbukumu özümlə götürməmişdim, amma vaxtımı faydalı keçirmək üçün google play store`dan telefonuma bir neçə java quiz yüklədim və oradakı sualları etməyə başladım. Quiz`dəki sualların demək olar ki, çoxu sadə idi, amma bəzi xırda, incə faktları daha yaxşı yadda saxlamaq, həmçinin sualları daha cəld şəkildə cavablandırmaq baxımından bir xeyli faydası oldu. İstifadə etdiyim digər faydalı mobil proqram isə AIDE oldu. Bu proqramı telefonda kompilyator və jre kimi istifadə edirdim. Beynimdə hər hansı bir qaranlıq sual yaranan kimi dərhal AIDE proqramını açıb həmin kodu test edirdim.

SYBEX online materiallarda hər biri 60 sualdan ibarət 3 ədəd practise exam var, kitabı tamam yekunlaşdırdıqdan sonra onlara başladım**:**

- Practise Exam 1 – 80%, 97 minutes
- Practise Exam 2 – 82%, 83 minutes
- Practise Exam 3 – 83%, 95 minutes

"OCA/OCP Java SE 7 Programmer I & II Study Guide"*(Kathy Sierra and Bert Bates)* kitabının əlavə olunmuş materiallar siyahısında OCA ilə bağlı 2 mock exam var idi, SYBEX`dən sonra onları etdim. Bu mock exam`lər digərlərinə nisbətən daha çətin idi və nəticələrim belə oldu:

- OCA Test 1 – 80%, 116 minutes
- OCA Test 2 – 75%, 110 minutes

Bura qədər əsas çətinlik çəkdiyim suallar:

1) *"Aşağıdakı ifadələrdən hansı doğrudur?"* – tipli suallar. Bu suallar yaxşı ingilis dili diliyi tələb edir. Çünki cümlədəki bircə sözün tərcüməsini bilməmək belə bəzən bütün cümlənin

ümumi anlamını yanlış yozmağa səbəb ola bilər. Ən yaxşı bildiyin bir mövzuya dair ifadəni belə səhv tərcümə səbəbindən yanlış cavablandıra bilərsən.

2) *"Choose all that apply"* – tipli suallar. Bu tip sualların əksəriyyətində ya cavablarımın sayı əksik olur, ya da artıq. Hərdən bir neçə cavab arasında tərəddüd edirsən, əgər doğru cavabların sayı qeyd olunsa, tapmaq nisbətən asan olar.

Hansı test bankından istifadə etdim?

Hələ imtahan hazırlığımın ortalarında sual bankı seçimimi etmişdim və ən sonda **Enthuware** məhsulunu almağı planlaşdırmışdım. Çünki Coderanch forumunda imtahandan uğurla keçənlərin əksəriyyəti öz təcrübələrində Enthuware mock examların onlara çox böyük faydası olduğunu yazırdı. Proqramın əvvəlcə trial versiyasını yüklədim və *"Sample Test"* suallarına baxdım. Suallar çox xoşuma gəldi və tam versiyanı sifariş etdim. Dizaynı çox sadə idi, əvvəlcə öyrəşə bilmirdim, amma sonradan çox bəyəndim. Və ilk olaraq "Standard Tests"ləri etdim.

Enthuware Standard Test`lər üzrə nəticələrim

Foundation Test	– 79%
Test 1	– 88%
Test 2	– 94%
Test 3	– 78%
Test 4	– 87%
Test 5	– 92%
Test 6	– 94%

Yanlış və ya doğru cavablandırmağımdan asılı olmayaraq bütün sualların izahını oxuyurdum və izahlardan bilmədiyim yeni şeylər öyrənirdim. Əgər izahdan da sualı məni qane edəcək səviyyədə başa düşmürdümsə, o zaman "discuss" butonunu klikləyərək forumdakı müzakirələrə (əgər varsa) baxırdım. İngilis dili səviyyəmə görə hərdən yaxşı başa düşmürdüm, başa düşənə qədər – 3 dəfə, 4 dəfə, lazım gəlirdisə lap 5 dəfə təkrar oxuyurdum.

Enthuware Standart Test`ləri bitirdikdən sonra (**Last Day Test`dən başqa**) imtahan üçün qeydiyyatdan keçdim. İmtahana hələ 1 həftə vaxt qalmışdı və həmin müddətdə qeydlərimi təkrarlamağa başladım. 150 səhifəyə yaxın qeydlərim var idi və hər mövzunu təkrarladıqca Enthuware Objective–wise Test`ləri edirdim. İmtahana 2 gün qalmış bütün hazırlığı bitirdim. Şənbə günü sonuncu Standart Test`i etdim:

- Last Day Test – 90%

Bazar günü istirahət etdim. Bazar ertəsi günü isə imtahana getdim.

İmtahan günü – 94% nəticə ilə imtahanı keçdim

Başlanğıcda imtahan həyəcanını boğa bilmədim, digər tərəfdən isə ilkin suallar çətin oldu deyə həyəcan bir az da artdı. 1 saatdan sonra normal vəziyyətimə qayıda bildim. İmtahan vermiş əksər namizədlər deyirdilər ki, real imtahan Enthuware mock exam`lərdən asan olur. Amma mənə düşən suallar heç də mock exam`lardakı suallardan asan suallar deyildi. Demək olar çox sual görünüşcə sadə idi, amma çaşdırıcı idi. Ona görə də gözümdən nəsə qaçmasın deyə hər suala ən azı 2 dəfə baxırdım. Mock exam`lərdə suallara təkrar baxmaq üçün yetəri qədər vaxtım qalırdı, fikirləşirdim real imtahan sualları daha asan olacaq və daha çox vaxtım qalacaq. Amma suallara təkrar nəzər yetirmək üçün 25 dəqiqə vaxtım qaldı və sona 30 saniyə qalmış imtahanı ancaq bitirə bildim. Həmçinin test mərkəzində imtahan üçün verilən kağız və qələm də rahat deyildi. Vərəq sürüşkən idi və qələmin yazması üçün hərdən 2-3 dəfə cəhd etməli olurdum. Bunun özü də müəyyən qədər konsentrasiyanın pozulmasına səbəb olurdu. İmtahanın nəticəsi imtahan bitən kimi açıqlanmır və son pəncərədə qeyd olunur ki, nəticələrlə bağlı məlumat 30 dəqiqə ərzində email ünvanınıza göndəriləcək. Təxminən 15-20 dəqiqə sonra nəticəm çıxdı:

Examination Score Report
Mushfiq Mammadov
Oracle Testing ID: OC1549695
1Z0-808 Java SE 8 Programmer I

Exam Date: 11/30/2015
Registration: 291799046
Center ID: 55176

Your Score: 94% **Passing Score:** 65% **Result:** Pass

Feedback on your performance is printed below. The report lists the objectives for which you answered a question incorrectly.

- Import other Java packages to make them accessible in your code
- Use super and this to access objects and constructors
- Write a simple Lambda expression that consumes a Lambda Predicate expression

95%-dən yuxarı nəticə hədəfləyirdim, amma bu həyəcanla buna da şükür etmək lazım idi. Uzunmüddətli əziyyətlərdən sonra nəhayət ki, məqsədimə çatmışdım.

Təqribən gün yarım sonra elektron poçtuma elektron sertifikat`ın hazır olması ilə bağlı email gəldi. Həmin emaildə həmçinin sertifikatın çap versiyasının əldə edilməsi üçün "online müraciət forması"nın linki var idi. Müraciət formasını doldurараq sorğu göndərdim və təqribən 4 həftə sonra sertifikatın çap forması göndərildi.

Jeanne Boyarsky və Scott Selikoff`un kitabı haqqında fikirlərim

Hazırlıq üçün istifadə etdiyim əsas kitab Jeanne Boyarsky və Scott Selikoff`un müəllifləri olduğu *"OCA: Oracle Certified Associate Java SE 8 Programmer I Study Guide: Exam 1Z0-808"* kitabı olub. İngilis dili səviyyəm yetərli olmadığına görə mənim üçün ilk öncə önəmli olan kitabın dili idi. Kitabı almamışdan öncə də Amazonda "Look inside"dan müəyyən hissələri oxuyaraq kitabın dilinə nəzər yetirmişdim, çox sadə və anlaşıqlı idi. Ondan sonra kitabı almaq qərarına gəldim. Qısa müddət ərzində kitabın dilinə öyrəşə bildim. Çox yığcamdı, əlavə informasiyalar ilə oxucunu çox yükləmir. Əgər real imtahana hansı mövzu ilə əlaqədar suallar düşməyəcəksə, chapter daxilində həmin mövzu geniş izah edilmir, sadəcə informasiyanız olsun deyə bildirilir ki, belə bir şey mövcuddur, amma OCA imtahanına düşməyəcək yaxud OCP imtahanına düşəcək və OCP kitabında geniş izah ediləcək. Beləcə imtahan üçün zəruri olmayan mövzuları oxumağa ehtiyacınız olmur və bu yolla da vaxtınıza qənaət etmiş olursunuz. Kitabın ən çox bəyəndiyim xüsusiyyəti imtahana düşəcək mövzularla/suallarla bağlı təxminləri oldu. Hansı mövzu/bölmə imtahanda soruşulmayacaq yazılmışdırsa, həqiqətən də həmin mövzu soruşulmadı.

Kitabda ən çox çətinlik çəkdiyim mövzu **Garbage Collection** oldu. Əvvəl-əvvəl bu tip sualları tapmaqda çətinlik çəkirdim, Coderanch`da garbage collector ilə bağlı bir neçə qəliz sual çıxmışdı rastıma və edə bilməmişdim. Ona görə bu mövzunu kənar ədəbiyyatdan da oxumağa ehtiyac duydum. Sonra yavaş-yavaş kağız-qələm ilə bu tip sualları daha rahat şəkildə tapmağın üsulunu öyrəndim. Amma imtahanda GC ilə bağlı cəmi bir sual düşdü və həmin sualı tapmaq üçün də təkcə SYBEX`dəki müvafiq mövzunu oxumaq tam yetərli idi.

Kitabda da qeyd edildiyi kimi demək olar ki, **imtahan mövzularını 100% əhatə edirdi**. Real imtahanda SYBEX kitabında olmayan ancaq bir şey rastıma çıxdı, o da *3 parametrli append()* metodu. Amma *bir parametrli append()* metodu kitabda izah edilib.

Errata (http://www.selikoff.net/java-oca-8-programmer-i-study-guide/) səhifəsinə baxsanız, kitabla bağlı ən çox texniki xəta report edən oxucu kimi mənim adımı görə bilərsiniz. Amma buna baxmayaraq mənim kitabla bağlı fikirlərim çox müsbətdir. Həmin texniki səhvlərin əksəriyyəti mövzunu qavramaqda sizə mane olacaq və imtahanda hər hansı bir sualı səhv cavablandırmağınıza səbəb olacaq səhvlər deyil. Xatırladığım 1-2 ciddi səhv var, onlar da səhv xatırlamıramsa Practise Exam`də rastıma çıxmışdı. Amma müsbət tərəfi odur ki, online material olduğu üçün operativ olaraq aradan qaldırılması mümkündür və mərhələli şəkildə bu proses həyata keçirilir.

Ümumilikdə kitab çox gözəl kitabdır, real imtahan suallarına istinadən deyə bilərəm ki, imtahanda düşən bütün sualları (3 parametrli `append()` metodu istisna olmaqla, onu da Enthuware`dən öyrənmişdim) əhatə edirdi. Həqiqətən də SYBEX kitabını çox sevdim. Kitaba başlamazdan öncə Assessment Test`də 3-4 sual güclə tapmışdım, amma kitabı bitirdikdən sonra bir xeyli inkişaf var idi. Əgər online materialların keyfiyyəti bir az daha artırılsa (məsələn, Practise Exam`də əvvəlki suallara qayıtmaq mümkün deyil; testlər üzrə

nəticələriniz yadda saxlanılmır, növbəti girişdə artıq itir; testlərdə vaxt limiti tətbiq edilmir və s.) gələcəkdə imtahana hazırlaşacaq namizədlər üçün daha faydalı bir vasitə olar.

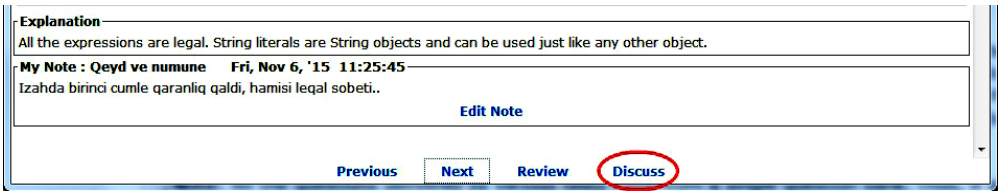
Qeyd: Bəzi məqamlar imtahana hazırlaşdığım dövrdəki mövcud vəziyyətə istinadən yazılıb, zamanla dəyişilə bilər.

Enthuware test bankı haqqında qeydlərim

Enthuware ilə bağlı fikirlərim çox yüksək səviyyədədir. Proqram digər alternativ proqramlarla müqayisədə daha keyfiyyətlidir və sualları da çox gözəldir. Məndən öncə imtahan verən namizədlərin təcrübələrindən də oxumuşdum ki, Enthuware sualları real imtahan suallarına çox bənzəyir. Artıq bu fikirlərə mən də qoşuluram, xüsusilə də **Last Day Test**`i qeyd etmək istəyirəm. İmtahan təcrübəmlə bağlı qeydlərimi yazarkən yadıma düşən suallardan bəzilərini hərdən ayırd edə bilmirdim ki, bu sual mənə imtahanda düşüb yoxsa Last Day Test`də qarşılaşmışam.

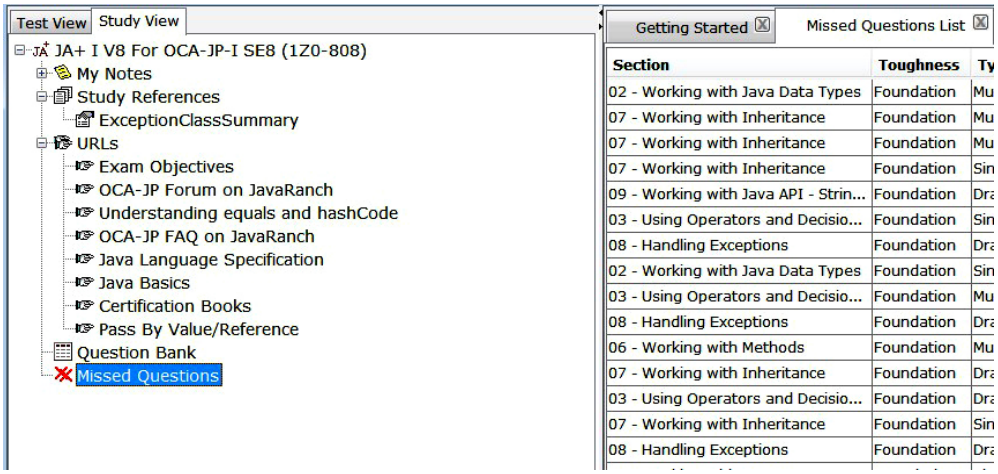
Proqramın strukturu primitiv görünsə də istifadəçinin rahatlığı üçün demək olar ki, bütün imkanlar yaradılıb:

- real imtahan pəncərəsinə bənzər sadə dizayn;
- əvvəlki, sonrakı və ya istədiyin nömrədə olan suala bir addımda getmə imkanı;
- sualı bookmark etmək;
- real imtahandakı kimi sona qalan vaxtla (azalan müddət üzrə) bağlı və həmçinin həmin müddət bitdikdə əlavə xərclənmiş vaxtla bağlı məlumat;
- proqrama istədiyin vaxt fasilə verib (Pause Test), istədiyin vaxt davam edə bilmək (Continue Test) (*amma arada çox fasilə etməyi məsləhət görmürəm, çünki real imtahan 2.5 saat olur və orada fasilə vermək imkanınız olmur, vaxtınız gedir. O səbəbdən mock exam`lərdə həmin bu 2.5 saat ərzində fasiləsiz (və ya az fasilələrlə) özünüzü suallara fokuslamağınızı, gərginlik hiss etdiyinizdə fasilə verməyinizi deyil, konsentrasiyanızı saxlayaraq, səbrlə suallara davam etməyinizi tövsiyə edirəm. Buna öyrəşə bilsəniz real imtahan üçün avantaj sağlamış olacaqsınız*);
- suallarla bağlı qeyd yaza bilmə imkanı (add note);
- başa düşmədiyiniz sualları Enthuware`nin forumunda müzakirə etmək imkanı - əgər izahı oxuduqdan sonra da sualı başa düşməsəniz "discuss" butonuna tıklamaqla forumdakı mövcud müzakirələri oxuya və ya sualınızı soruşa bilərsiniz. Suallar adminlər tərəfindən vaxtlı-vaxtında cavablandırılır;

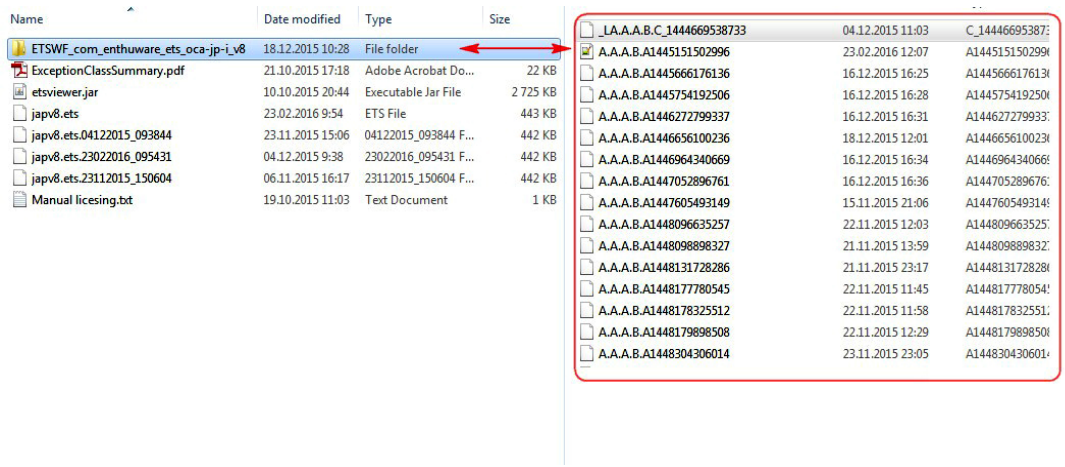


- **performance report** – bütün testlərdə, ayrı-ayrı mövzular üzrə nəticələriniz faizlə göstərilir və bunun əsasında zəif və güclü tərəflərinizi təyin edə bilərsiniz;

- **missed questions** – səhv cavablandırdığınız bütün suallar bu bölmə altında saxlanılır və sonda səhv etdiyiniz sualları tam başa düşdüyünüzə əmin olmaq üçün bu suallara təkrar nəzər yetirə bilərsiniz;



- Objective-wise Tests – ayrı-ayrı mövzulara aid sualları əhatə edir və tövsiyə olunur ki, Standart Test`lərdən sonra hansı mövzu üzrə biliklәriniz zəifdirsə (performance report`da qeyd olunur), həmin mövzunu kitabdan təkrarlayıb bu bölmə üzrə müvafiq sualları yenidən edəsiniz (Standart Tests`dəki suallar təkrarlanır);
- Most missed questions – namizədlər tərəfindən ən çox səhv cavablandırılmış suallara baxmaq imkanı;
- Həm evdə, həm də işdə işləmək imkanı -*"ETSWF_com_enthuware_ets_oca-jp-i_v8"* qovluğunda olan fayları daşımaqla evdə və yaxud işdə qaldığınız yerdən davam edə bilərsiniz.



Real imtahanda ən çox narahat qaldığım sual tipləri əsasən "*Which statements are true?*" tipli suallar idi, çünki tərcümədən dolayı səhv başa düşə bilmə ehtimalım çox idi. Amma imtahanda düşən bu tipli sualların demək olar ki, 90 faizini artıq Enthuware`dən öyrənmişdim, həmin suallara bənzər suallar Enthuware`də var idi.

Bu test bankı haqqında əlavə fikirlərimi "Addım 4. Test bankının seçilməsi və alınması" başlığında qeyd etmişəm.

Coderanch forumu haqqında düşüncələrim

Coderanch forumuna məni Jeanne xanım yönləndirmişdi, SYBEX kitabı ilə bağlı ona email yazarkən. Forumla tanış olduqdan qısa müddət sonra başa düşdüm ki, foruma gəlmək sertifikata hazırlıq boyunca qəbul etdiyim ən doğru qərarlardan biri idi. Çox böyük faydasını gördüm. Foruma heç nə yazmadığım vaxtlarda belə hər gün və ya 1-2 gündən bir daxil olub bütün yeni postları oxuyurdum. Bilmədiyim sualların izahlarını oxuyaraq öyrənirdim, lazım gəldikdə qeydlər götürürdüm. Hələ hazırlığımı bitirmədən öncə imtahan mərkəzləri, imtahana qeydiyyat, sertifikatın əldə olunması ilə bağlı sualları görmüşdüm, ən azından ümumi təsəvvür yaranmışdı və həmin sualları bookmark etmişdim ki, hazırlıq müddəti bitdikdən sonra lazım gələrsə yenidən baxım.

Başa düşmədiyim sualları isə soruşurdum. Düzdür, ingiliscə fikirlərimi yazmaq mənim üçün əlavə əziyyət idi, çünki xeyli vaxtımı alırdı. Amma bilmədiyim sualı öyrənmək xatirinə özümü məcbur edib yazırdım. Bu məcburiyyət eyni zamanda ingilis dilini inkişaf etdirməyə də böyük kömək edirdi. Hətta study guide`lərdə, test banklarında qarşıma çıxan sualların izahı qane etməyəndə, mövzunu tam mənimsəməyəndə ilk növbədə gəlib həmin sualı forumda axtarırdım. Güman edirdim ki, məndən öncə də kimsə o sualı forumda soruşa bilər və çox vaxt da elə olurdu. Forumda əksər suallar oxucu tam başa düşənə kimi dərindən, addım-addım izah edilir və bu da həmin mövzunu tam mənimsəməyə kömək edir.

Bundan əlavə namizədlərin öz təcrübələrini paylaşmalarının da böyük faydası var idi. Onların məsləhət və tövsiyələrindən yararlananda, artıq öncədən imtahan mühiti barədə təsəvvür formalaşırdı və gedib imtahanda hər hansı bir hadisə ilə qarşılaşanda sürpriz olmurdu.

Real imtahan sualları haqqında qısa qeydlərim

- ✓ `instanceof` operatoru, "dangling" else, `enum`, inner class ilə bağlı sual düşməmişdi;
- ✓ `Period` ilə bağlı sual var idi, amma `Duration`, `ZonedDateTime` ilə bağlı sual yox idi;
- ✓ Wrapper konstruktorla bağlı sual yox idi, amma `parseXXX()`, `valueOf()` metodları ilə bağlı sual var idi *(aşağıda ətraflı qeyd olunub);*
- ✓ Digər mövzularla müqayisədə ən çox sual massivlərlə əlaqəli idi (declaration, initialization);
- ✓ Switch case, iç-içə `for` dövrləri, `if` şərtləri ilə bağlı da çox sual var idi;
- ✓ `this()`, `super()` ilə bağlı bir xeyli sual var idi və bu sualları tapmaq üçün bilməli olduğumuz əsas qayda bu idi: `this()` və `super()` ancaq birinci sətirdə (1-ci ifadə kimi) çağırıla bilər;
- ✓ Lambda ilə əlaqəli cəmi bir sual düşmüşdü və həmin tip suala hazırlıq boyunca nə SYBEX, nə də Enthuware`də rastlamamışdım *(aşağıda ətraflı qeyd olunub);*
- ✓ Time API ilə bağlı 3 ya 4 sual düşmüşdü, hamısı da demək olar rahat suallar idi. SYBEX kitabında Dates və Times`lə bağlı mövzular bu sualları demək olar 95-99 faiz əhatə edirdi;
- ✓ *Pass-by-value* ilə əlaqəli təxminən 3 sual var idi;
- ✓ Overloading main metodla bağlı sual var idi;
- ✓ Cavabları *"compilation fails"* olan bir xeyli sual var idi;
- ✓ Casting ilə bağlı nisbətən mürəkkəb suallar var idi;
- ✓ `String`, `StringBuilder`in əksər vacib metodları ilə bağlı suallar düşmüşdü. Həmid metodların hamısı SYBEX kitabında öz əksini tapıb, bircəsindən başqa: 3 parametrli `append()` metodu. SYBEX kitabında ancaq bir parametrli `append()` metodu izah olunub, xoşbəxtlikdən 3 parametrli `append()` metodunu Enthuware suallarından öyrənmişdim. `intern()`, `compareTo()`, `ensureCapacity()`, `insert(4 parametrli)` (amma 2 paramterli düşmüşdü), `setLength()` metodları ilə bağlı suallar yox idi;
- ✓ SYBEX kitabında əhatə olunan Exception adlarından kənar exceptionlar düşməmişdi. Enthuware`də SYBEX-də olmayan bir neçə fərqli exception/error adlarına rast gəlmişdim (`AssertionError`, `IllegalStateException`, `SecurityException` və s.), amma imtahanda heç biri düşmədi, **SYBEX kitabı bu mövzunu tam əhatə edirdi**.

SYBEX və Enthuware testlərində rastlaşmadığım sual tipləri

Qeyd: Real imtahan suallarını paylaşmaq leqal hesab edilmir. Ona görə də aşağıdakı yazıda imtahana istinadən paylaşdığım kod nümunələri dəyişdirilmiş və bənzər kodlarla əvəz edilmişdir.

* İmtahana qədər gördüyüm iç-içə `for` dövrləri adətən bu formada olurdu: ikiölçülü massiv verilirdi, *outer for* iteration üçün massivin birinci elementinin uzunluğu, *inner for* iteration üçün isə massivin ikinci elementinin uzunluğu istifadə olunurdu, təxminən belə:

```
int count=0;
int[][] arr = new int[3][4];
for(int i=0; i<arr.length; i++)
   for(int j=0; j<arr[i].length; j++)
      if(j<2) count++;
System.out.println(count);
```

Amma imtahanda gördüyüm iç-içə `for` suallarında ikiölçülü yox, birölçülü `String` massiv istifadə olunurdu. Inner for iteration üçün massivin indekslərindəki elementlərin uzunluğu istifadə olunurdu, təxminən belə:

```
int count=0;
String[] arr = {"Roel", "Jeanne", "Paul", "Mushfiq"};
for(int i=0; i<arr.length; i++)
   for(int j=0; j<arr[i].length(); j++)
      if(arr[i].length() == 4) count++;
System.out.println(count);
```

Bu stildə, amma nisbətən daha mürəkkəb suallar idi.

* Grid şəklində massiv sualı. XO oyunundan yəqin hamının xəbəri var. Həmin oyundakı xanaları ikiölçülü massivin elementləri kimi təsəvvür etməliyik və `X`-in qalib gəlməsi üçün massivin hansı indeksdə dayanan xanasına `X` çəkilməlidir onu tapmalıyıq;

* Import ilə bağlı bəzi sual nümunələri var idi, ona bənzər nümunələr nə SYBEX, nə də ki Enthuware`də yox idi. Əslində nisbətən bənzər suallar rastıma çıxmışdı, amma əsas fərqlilik cavab variantları ilə əlaqəli idi *(aşağıda ətraflı qeyd olunub).*

Məni ən çox çaşdıran 4 sual

Qeyd: Real imtahan suallarını paylaşmaq leqal hesab edilmir. Ona görə də aşağıdakı yazıda imtahana istinadən paylaşdığım kod nümunələri dəyişdirilmiş və bənzər kodlarla əvəz edilmişdir.

1) 2 sualda iki variant arasında qalmışdım, seçim etməkdə çətinlik çəkirdim.

1-ci sual Wrapper`lə bağlı idi. Təxminən belə bir şey soruşulurdu:

```
public static void main(String[] args) {
   int i1 = Integer.parseInt(args[0]);
   boolean b1 = Boolean.parseBoolean(args[0]);
   System.out.println(i1 + " " + b1);
   int i2 = Integer.valueOf(args[0]);
   boolean b2 = Boolean.valueOf(args[0]);
   System.out.println(i2 + " " + b2);
}
```

`parseXXX()` metodları ilə bağlı problem yox idi, çünki geriyə primitive dəyər qaytarır. Amma `valueOf()` metodu geriyə Reference dəyər qaytarır, fikirləşdim bəlkə kompayl xətası verər, gərək sonunda `intValue()` də olsun: `Integer.valueOf(args[0]).intValue();`

Amma sonra fikirləşdim `intValue()` metodunu çağırmağa ehtiyac yoxdur, `Integer.valueOf(args[0]);` avtomatik olaraq unboxing vasitəsilə primitiv tipə mənimsədilə bilir. Gələn kimi yoxladım, doğru yazmışdım.

2-ci sual isə lambda ilə əlaqəli idi. Ümumiyyətlə, mən lambda mövzusunu çox yaxşı mənimsəmişdim, testlərdə də lambda ilə bağlı əksər sualları doğru cavablandırırdım, bilmədiyim bəzi şeyləri də Enthuware`dən öyrənmişdim. Amma imtahanda qarşıma çıxan lambda sualına bənzər nümunə nə SYBEX, nə də Enhtuware`də qarşıma çıxmamışdı, kifayət qədər çətin sual idi mənə görə. İmtahanın nəticəsi çıxanda, orada həmçinin səhv etdiyiniz sualların hansı mövzuya aid olması ilə bağlı feedback də göndərilir. Feedback`də 3 mövzu qeyd olunmuşdu, həmin mövzulardan biri də lambda idi. Lambda ilə bağlı cəmi bir sual düşdüyündən bildim ki, həmin sualdı. Sual isə təxminən aşağıdakı nümunəyə bənzər sual idi:

```
5:   String names[] = {"Roel", "Jeanne", "Paul", "Mushfiq"};
6:   List<String> list = new ArrayList(Arrays.asList(names));
7:   if(list.removeIf( (String s) -> { return s.contains("s"); } )){
8:      System.out.println(s + " removes");
9:   }
```

`asList()` ilə `removeIf()` metodunu bir yerdə görüncə ilk ağlıma gələn o oldu ki, bu kod `UnsupportedOperationException` fırladacaq, çünki `Arrays.asList()` ilə yaradılan list "fixed size list" hesab olunur və onun ölçüsünü nə artırmaq, nə də azaltmaq mümkün deyil. Amma həmin cavabı seçməzdən öncə lambda ifadəsinin sintaksisinin doğru olub olmadığını

yoxladım, çünki hazırlıqlardan belə başa düşmüşdüm ki, lambda ilə əlaqəli əksər suallar sintaksis ilə bağlı suallardır. `removeIf()` metodu `Predicate` interfeysi tipində parametr qəbul edir və:

- ✓ parametrin tipi aşkar şəkildə göstərilib (String) və buna görə də mötərizə olmalıdır və var;
- ✓ parametr siyahısı ilə gövdəni birləşdirən ox (arrow) var;
- ✓ geriyə `boolean` dəyər qayıtmalı, `return` ifadəsi yazıldığına görə nöqtəli-vergül, fiqurlu mötərizə olmalıdır və var.

Sintaksisdə hər şey qaydasında olduğuna əmin olduqdan sonra *"throws UnsupportedOperationException"* cavabını seçdim. Amma sonda ümumi suallara təkrar bir də nəzər keçirəndə hardasa oxuduğumu xatırladım ki (amma harada onu xatırlaya bilmədim), `new ArrayList` ilə yaradıldığına görə bu list "fixed size" hesab olunmur, ona görə də qərara gəldim ki, bu kod exception fırlatmır və *"`Mushfiq removes`"* cavabını seçdim. Amma bu kodu imtahandan qayıtdıqdan sonra IDE`də test edəndə gördüm ki, gözümdən qaçan, diqqətdən yayınan başqa bir məqam da var imiş. Sintaksis olaraq mən ancaq 7-ci sətiri yoxlamışdım, amma kompayl xətası 8-ci sətirdə idi. Lambda ifadəsinin parametr listində elan edilmiş dəyişən ancaq lambda ifadəsinin gövdəsində işlədilə bilər, 8-ci sətirdə `s` dəyişəninə müraciət etmək mümkün deyil. Cavab *"complation fails"* olmalı idi.

2) İki sualın isə quruluşu qəribə və mürəkkəb idi.

1-ci sual import ilə bağlı idi. Təxminən belə bir sual:

A adında bir proyektimiz var, `B` və `C` paketlərindən ibarətdir:

```
package B;
public class ClassB {}

package C;
public class ClassC {}
```

`ClassC`-nin `ClassB`-nin daxilində istifadə edilə bilməsi üçün hansı importlar tələb olunur? Doğru sintaksisə malik ikisini seçin (*təqribən*):

A. `String name = ClassC.getName();`
B. `import A;`
C. `import C.ClassC;`
D. `import B.ClassB;`
E. `String name = C.ClassC.getName();`

Tələb olunan import bircə **C** bəndidir. Amma 2 cavab seçilməli olduğuna görə digər münasib cavab kimi **D**-ni seçdim. **D** sintaksis olaraq doğru import-dur, amma tələb olunmur, onun yazılmasına ehtiyac yoxdur. Əgər `getName()` metodu `static` olsa idi, **E** bəndi doğru olardı.

Lakin `getName()` metodu ilə bağlı heç bir informasiya yox idi. Həmin vaxtı Coderanch forumunda Roel`in yazdığı bu sözlər yadıma düşdü: *"You should never assume something which is not mentioned in the question or in the option"*. Ona görə də **C** və **D** seçmişdim. Amma sual dəqiqliklə yadımda deyil, təxminən buna bənzər idi.

2-ci sual isə `this()` və `super()` konstruktorlarla bağlı idi. Məzmununa görə Enthuware Standart Test 5`dəki suallardan birinə bənzəyirdi, amma quruluşuna görə çox mürəkkəb idi. Həcmi çox böyük idi, ekrana sığışmırdı, sualı tam şəkildə görmək üçün scrolla aşağı sürüşdürmək lazım idi. Səhv etmirəmsə scroll`a ehtiyac olan yeganə sual idi.

Feedback`ə əsasən səhv etdiyim suallardan birinin lambda ilə əlaqəli sual olduğunu bildim. Amma səhv cavablandırdığım digər 4 sualın hansılar olduğu hələ də mənim üçün maraqlı olaraq qalır.