



OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 11 PROGRAMMER I EXAM FUNDAMENTALS **1Z0-815**

- Covers 100% of exam objectives
- Focuses on mastering concepts
- Includes coding exercises
- Complements Enthuware Mock Exams



Hanumant Deshmukh

OCP Java 11 PART I
EXAM STUDY GUIDE

eN
ENTHUWARE®

Oracle© Certified Professional Java SE 11 Programmer I
Fundamentals

(Exam Code 1Z0-815)

18th Sep 2019 Build 1.7

©Hanumant Deshmukh
www.enthware.com



Ещё больше книг в нашей группе:
<https://vk.com/javatutorial>

For online information and ordering of this book, please contact support@enthuware.com. For more information, please contact:

Hanumant Deshmukh
4A Agroha Nagar, A B Road,
Dewas, MP 452001
INDIA

Copyright © 2019 by Hanumant Deshmukh All Rights Reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under the relevant laws of copyright, without the prior written permission of the Author. Requests for permission should be addressed to support@enthuware.com

Limit of Liability/Disclaimer of Warranty: The author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the author is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. The author shall not be liable for damages arising herefrom.

The fact that an organization or website is referred to in this work as a citation and/or a potential source of further information does not mean that the author endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

Cover Design: Kino A Lockhart, LOXarts Development, <http://www.loxarts.com> .

TRADEMARKS: Oracle and Java are registered trademarks of Oracle America, Inc. All other trademarks are the property of their respective owners. The Author is not associated with any product or vendor mentioned in this book.

To my alma mater,
Indian Institute of Technology, Varanasi

Acknowledgments

I would like to thank numerous individuals for their contribution to this book. Thank you to Liu Yang for being the Technical Editor and Lisa Wright for being the copy editor. Thank you to Carol Davis and Robert Nyquist for technical proof reading. Thank you to Aakash Jangid and Bill Bruening for validating all the code snippets in this book.

Thank you to Maaike Van Putten for her inputs on the book cover design and to Kino A. Lockhart (LOXarts Development, <http://www.loxarts.com>) for designing the cover.

This book also wouldn't be possible without many people at Enthuware, including Paul A Prem, who have been developing mock exams for the past fifteen years. Their experience helped fine tune several topics in this book.

I would also like to thank Bruce Eckel, the author of "Thinking In Java" for teaching Java to me and countless others through his book.

I am also thankful to countless CodeRanch.com and Enthuware.com forum participants for highlighting the topics that readers often find difficult to grasp and also for showing ways to make such topics easy to understand.

Thank you to Edward Dang, Rajashekhar Kommu, Kaushik Das, Gopal Krishna Gavara, Dinesh Chinalachiagari, Jignesh Malavia, Michael Tsuji, Hok Yee Wong, Ketan Patel, Anil Malladi, Bob Silver, Jim Swiderski, Krishna Mangallampalli, Shishiraj Kollengreth, Michael Knapp, Rajesh Velamala, Aamer Adam, and Raghuvir Rawat for putting up with me :)

I would like to thank my family for their support throughout the time I was busy writing this book.

Finally, I am also thankful for the following readers for their help in improving the content of this book through suggestions and by reporting errors:

1. Zheng-Yu Wang

About the Author

Hanumant Deshmukh is a professional Java architect, author, and director of software consultancy firm BetterCode Infosoft Pvt. Ltd. Hanumant specializes in Java based multi-tier applications in financial domain. He has executed projects for some of the top financial companies such as JP Morgan Chase, UBS, and Bank Of America. He started Enthuware.com more than fifteen yrs ago through which he offers training courses and learning material for various Java certification exam. He has also co-authored a best selling book on Java Servlets/JSP certification, published by Manning in 2003.

Hanumant achieved his Bachelor of Technology from Institute of Technology, Banaras Hindu University (now, IIT - Varanasi) in Computer Science in 1997 and his Masters in Financial Analysis from ICFAI in 2010. After spending more than a decade working with amazing people in the United States, he returned back to India to pursue a degree in Law. He is a believer in freedom of speech and expression and works on promoting it in his spare time.

You may reach him at support@enthuware.com

Exam Objectives

The following are the exam objectives as of this writing. Oracle may tweak the objectives at any time so please verify the current objectives published at [OCP Java 11 Certification Part 1 Exam page at Oracle](#) .

Understanding Java Technology and environment

- Describe Java Technology and the Java development environment
- Identify key features of the Java language

Working With Java Primitive Data Types and String APIs

- Declare and initialize variables (including casting and promoting primitive data types)
- Identify the scope of variables
- Use local variable type inference
- Create and manipulate Strings
- Manipulate data using the StringBuilder class and its methods

Working with Java Arrays

- Declare, instantiate, initialize and use a one-dimensional array
- Declare, instantiate, initialize and use two-dimensional array

Creating and Using Methods

- Create methods and constructors with arguments and return values
- Create and invoke overloaded methods
- Apply the static keyword to methods and fields

Reusing Implementations Through Inheritance

- Create and use subclasses and superclasses
- Create and extend abstract classes
- Enable polymorphism by overriding methods
- Utilize polymorphism to cast and call methods, differentiating object type versus reference type
- Distinguish overloading, overriding, and hiding

Handling Exceptions

- Describe the advantages of Exception handling and differentiate among checked exceptions, unchecked exceptions, and Errors
- Create a try-catch block and determine how exceptions alter normal program flow
- Create and invoke a method that throws an exception

Creating a Simple Java Program

- Create an executable Java program with a main class
- Compile and run a Java program from the command line
- Create and import packages

Using Operators and Decision Constructs

- Use Java operators including the use of parenthesis to override operator precedence
- Use Java control statements including if, else, break and continue
- Create and use do/while loops, while loop, and for looping statements including nested loops

Describing Objects and Classes

- Declare and instantiate Java objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
- Define the structure of a Java class
- Read or write to object fields

Applying Encapsulation

- Apply access modifiers
- Apply encapsulation principles to a class

Programming Abstractly Through Interfaces

- Create and implement interfaces
- Distinguish class inheritance from interface inheritance including abstract classes
- Declare and use List and ArrayList instances

- Understanding lambda Expressions

Understanding Modules

- Describe the Modular JDK
- Declare modules and enable access between modules
- Describe how a modular project is compiled and run

Table of Contents

Introduction

- 0.1 Who is this book for?
- 0.2 How is this book different from others?
- 0.3 How is this book organized?
- 0.4 Will this be asked in the exam?
- 0.5 General tips for the exam
- 0.6 Official Exam Details and Exam Objectives
- 0.7 Feedback and Reviews

Chapter 1 Kickstarter for Beginners

- 1.1 Key points in OOP
 - 1.1.1 *A matter of perspective*
 - 1.1.2 *API*
 - 1.1.3 *Type, class, enum, and interface*
- 1.2 Why is something so?
- 1.3 Declaration and Definition
- 1.4 Relation between a class, an object, and a reference
- 1.5 Static and Instance
- 1.6 Stack and Heap
- 1.7 Conventions
 - 1.7.1 *What is a Convention?*
 - 1.7.2 *Conventions in Java*
- 1.8 Compilation and Execution
 - 1.8.1 *Compilation and Execution*
 - 1.8.2 *Running a single file source code program*
 - 1.8.3 *Packaging classes into Jar*
 - 1.8.4 *Compilation error vs exception at run time*
- 1.9 Commonly used terms in Java development
- 1.10 Java Identifiers

Chapter 2 Creating a Simple Java Program

- 2.1 Create an executable Java program with a main class
 - 2.1.1 *The main method*
 - 2.1.2 *Command line arguments*
 - 2.1.3 *The end of main*
- 2.2 Run a Java program from the command line

2.3 Create and import packages

2.3.1 *The package statement*

2.3.2 Quiz

2.3.3 *The import statement*

2.3.4 Quiz

2.4 Exercises

Chapter 3 Working With Java Primitive Data Types

3.1 Data types and Variable types

3.2 Reference variables and primitive variables

3.3 Declare and initialize variables

3.3.1 *Declare and initialize variables*

3.3.2 *Uninitialized variables and Default values*

3.3.3 *Assigning values to variables*

3.3.4 *final variables*

3.4 Wrapper Classes

3.4.1 *What are wrapper Classes?*

3.4.2 *Creating wrapper objects*

3.4.3 *Converting wrapper objects to primitives*

3.5 Exercise

Chapter 4 Describing and Using Objects and Classes

4.1 Declare and instantiate Java objects

4.2 Read or write to object fields

4.2.1 *Accessing object fields*

4.2.2 *What is "this"?*

4.3 Define the structure of a Java class

4.3.1 *Class disambiguated*

4.3.2 *Structure of a java source file*

4.3.3 *Members of a class*

4.3.4 *Relationship between Java source file name and class name*

4.3.5 Quiz

4.4 Identify the scope of variables

4.4.1 *Scope of variables*

4.4.2 *Scope and Visibility*

4.4.3 *Scope and Lifespan*

4.4.4 *Scopes Illustrated*

4.4.5 *Scope for the Exam*

4.4.6 Quiz

- 4.5 Local variable type inference
- 4.6 Explain objects' lifecycles
 - 4.6.1 *Life cycle of an Object*
 - 4.6.2 *Garbage Collection*
 - 4.6.3 *Garbage Collection for the exam*
- 4.7 Exercise

Chapter 5 Working with String APIs

- 5.1 Create and manipulate Strings
 - 5.1.1 *What is a "string"?*
 - 5.1.2 *Creating Strings*
 - 5.1.3 *String interning*
 - 5.1.4 *String immutability*
 - 5.1.5 *Manipulating Strings*
 - 5.1.6 *Comparing strings*
- 5.2 Manipulate data using the `StringBuilder` class and its methods
 - 5.2.1 *Why `StringBuilder`*
 - 5.2.2 *`StringBuilder` API*
- 5.3 Exercise

Chapter 6 Using Operators

- 6.1 Java Operators
 - 6.1.1 *Overview of operators available in Java*
 - 6.1.2 *Expressions and Statements*
 - 6.1.3 *Post and Pre Unary Increment/Decrement Operators*
 - 6.1.4 *String concatenation using + and += operators*
 - 6.1.5 *Numeric promotion and casting*
 - 6.1.6 *Operator precedence and evaluation of expressions*
- 6.2 Exercise

Chapter 7 Using Decision Constructs

- 7.1 Create if and if/else constructs
 - 7.1.1 *Basic syntax of if and if-else*
 - 7.1.2 *Usage of if and if-else in the exam*
- 7.2 The ternary conditional operator ?:
- 7.3 Creating a switch statement
- 7.4 Exercise

Chapter 8 Using Loop Constructs

- 8.1 What is a loop
- 8.2 Create and use while loops
 - 8.2.1 *The while loop*
 - 8.2.2 *Using a while loop*
- 8.3 The do-while loop
- 8.4 Create and use for loops
 - 8.4.1 *Going from a while loop to a for loop*
 - 8.4.2 *Syntax of a for loop*
 - 8.4.3 *Parts of a for loop*
- 8.5 Create and use for each loops
 - 8.5.1 *The enhanced for loop*
 - 8.5.2 *Syntax of the enhanced for loop*
 - 8.5.3 *Enhanced for loop in practice*
- 8.6 Use break and continue
 - 8.6.1 *Terminating a loop using break*
 - 8.6.2 *Terminating an iteration of a loop using continue*
- 8.7 Nested loops
 - 8.7.1 *Nested loop*
 - 8.7.2 *breaking out of and continuing with nested loops*
- 8.8 Comparison of loop constructs
- 8.9 Exercise

Chapter 9 Creating and Using Arrays

- 9.1 Declare, instantiate, initialize and use a one-dimensional array
 - 9.1.1 *Declaring array variables*
 - 9.1.2 *Creating and initializing array objects*
- 9.2 Using arrays
 - 9.2.1 *Array indexing*
 - 9.2.2 *Members of an array object*
 - 9.2.3 *Runtime behavior of arrays*
 - 9.2.4 *Uses of arrays*
- 9.3 Declare, instantiate, initialize and use multi-dimensional arrays
 - 9.3.1 *Multidimensional Arrays*
 - 9.3.2 *Assigning arrays of primitives to Object variables*
- 9.4 Exercise

Chapter 10 Creating and Using Methods

- 10.1 Create methods with arguments and return values
 - 10.1.1 *Creating a method*

- 12.1.2 Inheriting features from a class
- 12.1.3 Inheritance and access modifiers
- 12.1.4 Inheritance of instance members vs static members
- 12.1.5 Benefits of inheritance
- 12.2 Using super and this to access objects and constructors
 - 12.2.1 Object initialization revisited
 - 12.2.2 Initializing super class using "super"
 - 12.2.3 Using the implicit variable "super"
 - 12.2.4 Order of initialization summarized
- 12.3 Create and extend abstract classes
 - 12.3.1 Using abstract classes and abstract methods
 - 12.3.2 Using final classes and final methods
 - 12.3.3 Valid combinations of access modifiers, abstract, final, and static
- 12.4 Enable polymorphism by overriding methods
 - 12.4.1 What is polymorphism
 - 12.4.2 Overriding methods
 - 12.4.3 Invalid overrides
- 12.5 Utilize polymorphism to cast and call methods
 - 12.5.1 Type of reference and type of an object
 - 12.5.2 Bridging the gap between compile time and run time
 - 12.5.3 When is casting necessary
 - 12.5.4 The instanceof operator
 - 12.5.5 Invoking overridden methods
 - 12.5.6 Impact of polymorphism on == and equals method
- 12.6 Overriding and Hiding
- 12.7 Exercise

Chapter 13 Programming Abstractly Through Interfaces

- 13.1 Create and implement interfaces
 - 13.1.1 Using interfaces
 - 13.1.2 Implementing an interface
 - 13.1.3 Extending an interface
 - 13.1.4 Instantiating abstract classes and interfaces
- 13.2 Difference between Interface and Abstract Class
- 13.3 Declare and use List and ArrayList instances
 - 13.3.1 Introduction to Collections and Generics
 - 13.3.2 Generics
 - 13.3.3 Quiz
 - 13.3.4 Collection and List API

13.3.5 ArrayList API

13.3.6 ArrayList vs array

13.3.7 Map and HashMap

13.3.8 Quiz

13.4 Exercise

Chapter 14 Lambda Expressions

14.1 Understanding Lambda Expressions

14.1.1 Lambda Expressions

14.1.2 Parts of a Lambda expression

14.1.3 Using Predicate

14.1.4 Functional Interfaces

14.1.5 Using Functional Interfaces with Collections API

14.1.6 Scope of variables in a lambda expression

14.1.7 Quiz

14.2 Exercise

Chapter 15 Handling Exceptions

15.1 Create try-catch blocks and determine how exceptions alter program flow

15.1.1 Java exception handling

15.1.2 Fundamentals of the try/catch approach

15.1.3 Pieces of the exception handling puzzle

15.2 Differentiate among checked, unchecked exceptions, and Errors

15.2.1 Checked and Unchecked exceptions

15.2.2 Commonly used exception classes

15.3 Create and invoke a method that throws an exception

15.3.1 Creating a method that throws an exception

15.3.2 Throwing exceptions from initializers and constructors

15.3.3 Invoking a method that throws an exception

15.3.4 Using multiple catch blocks

15.4 Exercise

Chapter 16 Understanding Modules

16.1 Module Basics

16.1.1 What are modules?

16.1.2 Declaring a module

16.1.3 Directory structure of a module

16.2 Describe how a modular project is compiled and run

16.2.1 Compiling a module

16.2.2 Running a module

16.3 Declare modules and enable access between modules

16.3.1 Enabling access between modules

16.3.2 Qualified exports

16.3.3 Transitive dependencies

16.4 Advanced module compilation and execution

16.4.1 Compiling multiple modules at once

16.4.2 Module jar vs regular jar

16.4.3 Summary of command line switches used for compilation

16.4.4 Summary of command line switches used for execution

16.5 Describe the modular JDK

16.5.1 Modular JDK

16.5.2 Organization of the modular JDK

16.5.3 Benefits of the modular JDK

16.6 Exercise

Chapter 17 Understanding Java Technology and environment

17.1 Java Technology and key features of the Java language

Introduction

I believe you have already got your feet wet with Java programming and are now getting serious about your goal of being a professional Java programmer. First of all, let me commend your decision to consider Java certification as a step towards achieving that goal. I can assure you that working towards acquiring **Oracle's Java Certification** will be very rewarding. Irrespective of whether you get extra credit for being certified in your job hunt or not, you will be able to speak with confidence in technical interviews and the concepts that this certification will make you learn, will improve your performance on the job.

The Java SE 11 Programmer I exam (Exam code **1Z0-815**), aka OCP JP-I exam, is the first of the two exams that you need to pass in order to become an Oracle Certified Professional - Java SE 11 Developer. This exam focuses on the fundamental aspects of Java and is not particularly tough to pass. If you go through a decent book and practice a few good mock exams, you should be able to pass it with a couple of months of preparation. However, the topics covered in this certification form the groundwork for the second step of professional certification, i.e., the Java SE 11 Programmer II exam (Exam code **1Z0-816**), aka OCP JP-II exam. The OCPJP-II is a very tough exam. It is a lot tougher than the OCPJP-I exam. You will have trouble passing that exam if your fundamentals are weak. For this reason, it is very important to not think of just passing the OCPJP-I exam with the bare minimum marks required (63%) but to set a score of 90% as your target. My objective with this book is to help you achieve 90% plus score on the OCPJP-1 exam.

About the mock exams

Mock exams are an essential preparation tool for achieving a good score on the exam. However, having created mock exams for several certifications, I can tell you that creating good quality questions is neither easy nor quick. Even after multiple reviews and quality checks, it takes years of use by thousands of users for the questions to shed all ambiguity, errors, and mistakes. I have seen users come up with plausible interpretations of a problem statement that we could never imagine. A bad quality mock exam will easily eat up your valuable time and may also shake your confidence. For this reason, I have not created new

mock exams for this book. We have a team that specializes in developing mock exams and I will recommend you to buy the exam simulator created by this team from our website Enthuware.com. It is priced quite reasonably (only 9.99 USD) and has stood the test of time.

0.1 Who is this book for?

This book is for Java SE 11 Programmer - I exam (1Z0-815) takers who know how to program and are at least aware of the basic Java terminology. Before proceeding with this study guide, please answer the following questions.

Remember that you don't have to be an expert in the topic to answer yes. The intention here is to check if you are at least familiar with the basic concepts. It is okay if you don't know the details, the syntax, or the typical usage. I will go through all that in this book, but I will not teach the basics of programming in this book.

1. Do you know what OS, RAM, and CPU are?
2. Do you know what a command line is?
3. Do you know basic OS commands such as `dir`, `cd`, and `mkdir` (or if you are a Linux/Mac user - Do you know how to use `ls`, `cd`, and `md`)?
4. Can you write a simple `Hello World` program in Java and run it from the command line?
5. Do you know what variables are?
6. Do you know what loops (such as for loop and while loop) are and what they are used for?
7. Are you aware of arrays?
8. Are you aware that Java has classes and interfaces?
9. Are you aware that classes and interfaces have methods?
10. Have you installed JDK 11 on your computer?

If you answered no to any of the above, this book is not for you. It would be better if you go through a programming book or a computer book for beginners first, and then come back to this book. Alternatively, be open to google a term if you are not sure about it at any time before proceeding further while reading this book.

0.2 How is this book different from others?

With so many certification books around, I think this question is worth answering at the outset. This book is fundamentally different from others in the following respects:

1. **Focus on concepts** - I believe that if you get your basic concepts right, everything else falls in place nicely. While working with Java beginners, I noticed several misconceptions, misunderstandings, and bad short cuts that would affect their learning of complex topics later. I have seen so many people who manage to pass the exam but fail in technical interviews because of this reason. In this book, I explain the important stuff from different perspectives. This does increase the length of the book a bit but the increase should be well worth your time.
2. **No surgical cuts** - Some books try to stick very close to the exam objectives. So close that sometimes a topic remains nowhere close to reality and the reader is left with imprecise and, at times, incorrect knowledge. This strategy is good for answering multiple choice questions appearing on the OCPJP-I exam but it bites the reader during technical interviews and while writing code on the job. I believe that answering multiple choice questions (MCQs) should not be your sole objective. Learning the concepts correctly is equally important. For this reason, I go beyond the scope of exam objectives as, and when, required. Of course, I mention it clearly while doing so.
3. **Exercises** - "Write a lot of code" is advice that you will hear a lot. While it seems quite an easy task for experienced programmers, I have observed that beginners are often clueless about what exactly they should be writing. When they are not sure about what exactly a test program should do, they skip this important learning step altogether. In my training sessions, I give code writing exercises with clear objectives. I have done the same in this book. Instead of presenting MCQs or quizzes at the end of a topic or chapter, I ask you to write code that uses the concepts taught in that topic or chapter.

Besides, a question in the real exam generally requires knowledge of multiple topics. The following is a typical code snippet appearing in the exam:

```
int i = 10;  
Long n = 20;  
float f = 10.0;  
String s = (String) i+n++;
```

To determine whether this code compiles or not, you need to learn four topics - wrappers, operators, String class, and casting. Thus, presenting an MCQ at the end of a topic, that focuses only on that one topic, creates a false sense of confidence. I believe it is better to focus on realistic MCQs at the end of your preparation.

4. **Not being pedantic** - If you are preparing for the OCPJP-I exam, I believe you have already been through many academic exams in your life. You already know what to expect in an exam. So, I won't advise you on the amount of water you should drink before the exam to avoid a restroom break, or on how much sleep you should get before the exam, or to check the exam center location a day before. If you have not taken any computer-based exam containing multiple choice questions, I strongly suggest you use Enthuware's exam simulator to get familiar with this style. It closely mimics the user interface of the real exam.

0.3 How is this book organized?

This book consists of seventeen chapters plus this introduction at the beginning. Other than the first chapter "Kickstarter for the Beginners", the chapters correspond directly to the official exam objectives. The sections of a chapter also correspond directly to the items of exam objectives in most cases. Each chapter lists the exam objectives covered in that chapter at the beginning and includes a set of coding exercises at the end. It would be best to read the book sequentially because each chapter incrementally builds on the concepts discussed in the previous chapters. I have included simple coding exercises throughout the book. Try to do them. You will learn and remember the concept better when you actually type the code instead of just reading it. If you have already had a few years of Java development experience, you may go through the chapters in any order.

Conventions used in this book

This book uses certain typographic styles in order to help you quickly identify important information. These styles are as follows:

Code font - This font is used to differentiate between regular text and Java code appearing within regular text. Code snippets containing multiple lines of code are formatted as Java code within rectangular blocks.

Red code font - This font is used to show code that doesn't compile. It could be because of incorrect syntax or some other error.

Output code font - This font is used to show the output generated by a piece of code on the command line.

Bold font - I have highlighted important words, terms, and phrases using bold font to help you mentally bookmark them. If you are cruising through the book, the words in bold will keep you oriented besides making sure you are not missing anything important.

Note -

Things that are not completely related to the topic at hand are explained in notes. I have used notes to provide additional information that you may find useful on the job or for technical interviews but will most likely not be required for the exam.

Exam Tip:

Exam Tips contain points that you should pay special attention to during the exam. I have also used them to make you aware of the tricks and traps that you will encounter in the exam.

Asking for clarification

If you need any clarification, have any doubt about any topic, or want to report an error, feel free to ask on our dedicated forum for this book - <http://enthuware.com/forum>. If you are reading this book on an electronic

device, you will see this icon  beside every topic title. Clicking on this icon will take you to an existing discussion on that particular topic in the same forum. If the existing discussion addresses your question, great! You will have saved time and effort. If it doesn't, post your question with the topic title in the subject line. We use the same mechanism for addressing concerns about our mock exam questions and have received tremendous appreciation from the users about this feature.

0.4 Will this be asked in the exam?

While going through this book, you will be tempted to ask this question many times. Let me answer this question at the beginning itself. I do talk about concepts in this book that are not explicitly listed in the official exam objectives but wherever I deviate from the official exam objectives, I clearly specify so. You are free to ignore that section and move on. But I suggest you do not skip such sections because of the following reasons.

1. While discussing a rule of the language, I may have to refer to some terms and concepts for the sake of completeness and technical accuracy. For

example, let's say we are talking about public classes in a file. If I state that you cannot have more than one public class in a file, it is fine for the purpose of the exam but it is technically incorrect because you can have any number of public nested classes in a file. Thus, it would be better to state that you cannot have more than one top-level class in a file. How about one public top-level class and one public interface? Nope, you can't do that either. Thus, the statement is still incorrect. The correct statement would be that you cannot have more than one public top-level reference type in a file. As you can see, it is imperative for me to mention the meaning of the terms reference type, nested class, and top level class, even though you won't be tested on them in the exam.

If you absolutely do not want to spend any time learning about anything that is not part of the exam, then this book is not for you. I have tried to stick to the objectives as much as possible but, if I believe you need to know something, I talk about it even if it is beyond the scope of the exam.

2. I have noticed that many of the certification aspirants are new Java programmers who are either in school or want to start their career with Java programming. They want to get certified because they ultimately want to land a job as a Java programmer. These programmers will be facing a lot of technical interviews as well. I want these programmers to do well on technical interviews.

Certification may get you a foot in the door but you will need to back it up with strong knowledge of fundamentals in the interview. Therefore, if I believe that something is important for you to know or that something will be helpful to you in your technical interview, irrespective of whether it will be asked in the exam or not, I discuss it.

3. Official exam objectives are neither detailed nor exhaustive. They list top level topics that you need to study but leave out finer details. You will be asked questions that require you to know those concepts.
4. Oracle adds new questions to the exam before formally adding a new topic in the official exam objectives. These questions may not be included in your final score, i.e., your answers on such questions are not counted towards your score on the exam. However, test takers do not know if a question is unscored and so, they must attempt it as if it will be counted towards their

final score.

Since we, at Enthuware, conduct classroom training as well, we get to interact with a lot of test takers. We receive feedback from test takers about getting questions on topics that are not there in the exam objectives. After receiving such multiple reports, we may decide to add that topic to our content. We clearly specify the reason for their inclusion.

5. Official exam objectives are not constant. Although not frequently, Oracle does add and remove topics from the objectives from time to time. This may render some of the content not relevant for the exam. I will update the content as soon as possible.

If you are interested in getting your basics right, then I suggest you do not worry too much about the exam objectives while following this book. Even if you spend a little more time (not more than 10%, I promise) in your preparation because of this extra content, it will be worth your while.

0.5 General tips for the exam

Here is a list of things that you should keep in mind while preparing for the exam -

1. **Code Formatting** - You may not find nicely formatted code in the exam. For example, you may expect a piece of code nicely formatted like this:

```
if(flag){  
    while(b<10){  
    }  
}else if(a>10) {  
    invokeM(a);  
}  
else{  
    System.out.println(10);  
}
```

But you may get the same code formatted like this:

```

if(flag){
    while(b<10){ }
} else
if(a>10) { invokeM(a); }
else { System.out.println(10); }

```

They do this most likely to save space. But it may also happen inadvertently due to variations in display screen size and resolution.

2. **Assumptions** - Several questions give you partial code listings, aka "code snippets". For example, what will the following code print?

```

ArrayList al = new ArrayList();
al.remove(0);
System.out.println(al);

```

Obviously, the code will not compile as given because it is just a code fragment. You have to assume that this code appears in a valid context such as within a method of a class. You also need to assume that appropriate import statements are in place.

You should not fret over the missing stuff. Just focus on the code that is given and assume that everything else is irrelevant and is not required to arrive at the answer.

3. **Tricky Code** - You will see really weird looking code in the exam. Code that you may never even see in real life. You will feel as if the exam is about puzzles rather than Java programming. To some extent, that is correct. If you have decided to go through the certification, there is no point in questioning the relevance. If you feel frustrated, I understand. Please feel free to vent out your anger on our forum and get back to work!
4. **Number of correct options** - Every question in the exam will tell you exactly how many options you have to select to answer that question correctly. Remember that there is no negative marking. In other words, marks will not be deducted for answering a question incorrectly. Therefore, do not leave a question unanswered. If you don't know the answer, select the required number of options anyway. There is a slight chance that you will have picked the correct answer.
5. **Eliminate wrong options** - Even better than not leaving a question unanswered is make intelligent guesses by eliminating obviously incorrect options. You may see options that are contradictory to each other. This makes it a bit easy to narrow down the correct options.

That's about it. Hope this book helps you become a better Java programmer besides getting you the certification.

0.6 Official Exam Details and Exam Objectives

The following are the official exam details published by Oracle as of 1st July 2019. As mentioned before, Oracle may change these details at any time. They have done it in the past. Several times. Therefore, it would be a good idea to check the official exam page at https://education.oracle.com/java-se-11-programmer-i/pexam_1Z0-815 during your preparation.

Exam Details

Duration : 180 Minutes

Number of Questions: 80

Passing Score: 63%

Format: Multiple Choice

Exam Price: USD 245 (varies by country)

Exam Objectives

Understanding Java Technology and environment

- Describe Java Technology and the Java development environment
- Identify key features of the Java language

Working With Java Primitive Data Types and String APIs

- Declare and initialize variables (including casting and promoting primitive data types)
- Identify the scope of variables
- Use local variable type inference
- Create and manipulate Strings
- Manipulate data using the StringBuilder class and its methods

Working with Java Arrays

- Declare, instantiate, initialize and use a one-dimensional array
- Declare, instantiate, initialize and use two-dimensional array

Creating and Using Methods

- Create methods and constructors with arguments and return values
- Create and invoke overloaded methods
- Apply the static keyword to methods and fields

Reusing Implementations Through Inheritance

- Create and use subclasses and superclasses
- Create and extend abstract classes
- Enable polymorphism by overriding methods
- Utilize polymorphism to cast and call methods, differentiating object type versus reference type
- Distinguish overloading, overriding, and hiding

Handling Exceptions

- Describe the advantages of Exception handling and differentiate among checked exceptions, unchecked exceptions, and Errors
- Create a try-catch block and determine how exceptions alter normal program flow
- Create and invoke a method that throws an exception

Creating a Simple Java Program

- Create an executable Java program with a main class
- Compile and run a Java program from the command line
- Create and import packages

Using Operators and Decision Constructs

- Use Java operators including the use of parenthesis to override operator precedence
- Use Java control statements including if, else, break and continue
- Create and use do/while loops, while loop, and for looping statements

including nested loops

Describing Objects and Classes

- Declare and instantiate Java objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
- Define the structure of a Java class
- Read or write to object fields

Applying Encapsulation

- Apply access modifiers
- Apply encapsulation principles to a class

Programming Abstractly Through Interfaces

- Create and implement interfaces
- Distinguish class inheritance from interface inheritance including abstract classes
- Declare and use List and ArrayList instances
- Understanding lambda Expressions

Understanding Modules

- Describe the Modular JDK
- Declare modules and enable access between modules
- Describe how a modular project is compiled and run

0.7 Feedback and Reviews

This is the first draft of the book and we are currently reviewing the content and the code snippets for technical correctness. If you have any query regarding the contents of this book or if you find any error, please do let me know on [Enthuware Forum](#).

I hope you enjoy reading this book. If you learn a few things and find it interesting, I would be very grateful if you would consider leaving a review on

Amazon or Google Play with a few kind words. If you have received a review copy of this book, please mention so, in your review.

thank you,
Hanumant Deshmukh

Chapter 1 Kickstarter for Beginners

This section is for Java beginners. It does not directly relate to any exam objective but is meant to provide a solid grounding that will help you to easily understand the concepts taught in later chapters. The concepts covered in this section are important because they kind of repeat over and over throughout this course. If we get these repetitions over with now, you will be happier later on!

1.1 Key points in OOP

1.1.1 A matter of perspective

A couple of years ago, while I was visiting India, I had a tough time plugging in my laptop charger in the 3-pin sockets. Even the international socket adapter kit, which had adapter pins of various sizes, was not of much help. Sometimes the receiver would be a bit too small and the pins wouldn't make steady contact or the pins would be a bit too wide and won't go into the receiver. I had to finally cut my cord and stick the bare copper wire ends directly into the sockets. I wondered, why do all these sockets in the same country have slight differences. During my stay, I observed that such minor variations were present in other things as well. Doors that wouldn't completely close, nuts that wouldn't turn properly, taps that wouldn't stop leaking, and other differences. Most of the time, people there take the trial and error approach when replacing parts. They work with the expectation that even if they get a part with the right size, it still may not fit perfectly. In other words, minor variations are expected and well tolerated.

This was unimaginable to me in the US, where everything just fits. I could buy a bolt from one shop and a nut from another, and it would work perfectly. Everything, from screws, nuts, and bolts, to wood panels, electrical parts, packing boxes, is standardized. One can easily replace a part with another built by a totally different company. You just have to specify the right "size".

This experience led me to a potential cause of why some OOP learners find OOP

concepts confusing. Especially, beginners from non-western background find it really tough to grasp the fundamental concepts because they do not know the rationale behind so many rules of OOP. This is reflected in their application design.

In the US, and I imagine in other developed countries as well, things are extremely well defined. Products clearly specify how they should be used and in what cases they will fail. People can and do rely on these specifications because products work as expected and fail as defined. At the same time, people expect products to come with detailed specifications. Ready to assemble furniture is a prime example of how detailed these specifications can be. It's because of detailed and clear specifications that people feel comfortable in buying complicated ready-to-assemble furniture.

In short, people know exactly what they are getting when they acquire something. I think of it as the society being naturally "object-oriented".

Object orientation is just a name for the same natural sense of things fitting nicely with each other. A piece of code is not much different from the physical things I mentioned earlier. If you code it to a specification, it will fit just as nicely as your .16 inch nut on a .16 inch bolt, irrespective of who manufactured the nut and who manufactured the bolt. The point here is that the source of the concept of object-oriented programming is the physical world. If you want to grasp OOP really well, you have to start thinking of your piece of code as a physical component...a "thing" that has a well defined behavior and that can be replaced with another component that adheres to the same behavior. You would not be happy if you bought a tire that doesn't fit on your car's wheel even though you bought same 'size', would you? You should think about the same issues when you develop your software component. Someone somewhere is going to use it and they won't be happy if it fails at run time with an exception that you didn't say it would throw in a particular situation.

1.1.2 API

You probably know that **API** stands for **Application Programming Interface**. But do you understand what it really means? This goes back to my previous

observation about relating programming concepts to real life. When you operate a switch do you really care about what exists inside the switch? Do you really care how it works? You just connect the switch to a light bulb and press it to switch the bulb on or off. It is the same with a car. A car provides you with a few controls that allow you to turn, accelerate, and brake. These controls are all you need to know how to drive a car.

You should think about developing **software components** in the same way. A software component doesn't necessarily mean a bunch of classes and packages bundled together in jar file. A software component could be as simple as a single class with just one method. But while developing even the smallest and the simplest of software components, you should think about how you expect the users to use it. You should think about various ways a user can use the component. You should also think about how the component should behave when a user doesn't use it the way it is expected to be used. In other words, you should specify an **interface** to your component, even before you start coding for it. Here, by interface, I do not mean it in the strict sense of a Java interface but a specification that details how to interact with your component. In a physical world, the user's manual of any appliance is basically its interface. In the software world, the specification of the publicly usable classes, methods, fields, enums, et cetera of a software component is its interface. As an **application programmer**, if you want to use a component developed by someone else, you need to worry only about the interface of that component. You don't need to worry about what else it might contain and how it works. Hence the phrase 'Application Programming Interface'.

In the Java world, a collection of classes supplied by a provider for a particular purpose is called a **library** and the **JavaDoc** documentation for those classes describes its API. When you install the **Java Runtime Environment** (JRE), it includes a huge number of classes and packages. These are collectively called the standard Java library and the collection of its public classes and interfaces is called the **Java API**.

The Java API contains a huge amount of ready-made components for doing basic programming activities such as file manipulation, data structures, networking, dates, and formatting. When you write a Java program, you actually build upon the Java API to achieve your goal. Instead of writing all the logic of your application from scratch, you make use of the functionality already provided to you, free of cost, by the Java library and only write code that is

specific to your needs. This saves a lot of time and effort. Therefore, a basic understanding of the Java API is very important for a Java programmer. You don't need to know by heart all the classes and their methods. It is practically impossible to know them all, to be honest. But you should have a broad idea about what the Java API provides at a high level so that when the need arises, you know where to look for the details. For example, you should know that the standard Java library contains a lot of classes for manipulating files. Now, when you actually need to manipulate a file, you should be able to go through the relevant Java packages and find a Java class that can help you do what you want to do.

The **OCPJP 11 Part 1** exam requires that you know about only a few packages and classes of the Java API. I will go through them in detail in this book.

If you keep the above discussion in mind, I believe it will be very easy for you to grasp the concepts that I am going to talk about throughout this book.

1.1.3 Type, class, enum, and interface [↳](#)

A **type** is nothing but a name given to a behavior. For example, when you define how a bank account behaves when you interact with it, you are defining a type and if you give this behavior a name, say Account, then you have essentially defined the Account type.

From this perspective, a **class**, an **enum**, and an **interface** help you define certain kinds of behaviors and are thus, types of types.

A **class** allows you to combine the description of a behavior and the implementation that is used to realize this behavior. The implementation includes logic as well data. For example, an account allows you to withdraw and deposit money, which is the description of its behavior, and uses "account balance", which is the data that it manipulates to realize this behavior. Thus, Account could be a class. Once you define the behavior of an account and also provide the implementation to realize this behavior, you can have as many accounts as you want.

An **enum** , which is a short form for enumeration, also allows you to combine the description of a behavior and the implementation that is used to realize this behavior. However, in addition, it provides a fixed number of instances of this type and restricts you from creating any new instances of this type. For example, there are only 7 days in a week (from Monday to Sunday). Thus, if you define DayOfWeek, you wouldn't want to create a day other than those predefined 7 days. Thus, DayOfWeek could be an enum with 7 predefined unchangeable instances.

An **interface** allows you to define just the behavior without any implementation for it. For example, if you want to describe something that moves, you can call it Movable. It doesn't tell you anything about how that entity moves. A cow, a car, or even a stock price all move, but obviously, they move very differently. Thus, Movable could be an interface.

The key point about an interface is that you cannot have an instance of an interface because it is just a description of the way you can interact with something and is not the description of the thing itself. For example, you cannot really have just a Movable. You must have a cow or a car or something else that exhibits the behavior described by Movable. In that sense, an interface is always **abstract** . It cannot exist on its own. You need a class to **implement** the behavior described by an interface.

Besides the above three, there is something called **abstract class** . An abstract class lies somewhere in between a class and an interface. Just like a class, it defines behavior as well as implementation but the implementation that it provides is not complete enough for you to create instances of it. Therefore, just like an interface, it cannot exist on its own. For example, if you define the behavior that is common to animals along with some implementation that is common to all animals in a class. But you know that you can't really have just an Animal. You can have a cat, or a dog, or a cow, which are all animals, but not just an animal.

1.2 Why is something so

1.2.1 Why is something so?

Why does Java not have **pointers**? Why does Java permit static fields and methods? Why does Java not have **multiple inheritance**? Why does this work but that doesn't? While learning Java, curious minds get such questions very often. Throughout the book, you will come across rules and conventions that will trigger such questions. Most of the times the reason is not too complicated. I will explain four points below that will help you answer most of such questions. I will also refer to them throughout the book wherever warranted.

1. **To help componentize the code** - As discussed earlier in the API section, while writing Java code, you should think about developing **components** instead of writing just **programs**. The difference between the two is in the way they allow themselves to be used interchangeably. Can you imagine a 3 pin socket that has the ground pin on the left instead of on the top? No one makes such a thing because it won't allow any other plug to be plugged in. In that sense, Components are like generic Lego blocks. You can mix and match the blocks with basic functionality and build even bigger blocks. You can take out one block and replace it with another block that has the same connectors. It is the same with software components.

A well-developed software component is as **generic** as possible. It is built to do one thing and allows other components to make use of it without making them dependent on it. Dependency here means that you should be able to easily replace this component with another component that does the same thing. Indeed, you should be able to replace a 3 pin socket from one manufacturer with another without needing to replace the entire appliance!

A program, on the other hand, is a monolithic pile of code that tries to do everything without exposing generic and clear interfaces. Once you start using a program, it is almost impossible to replace it with another one without impacting all other pieces that work with that program. It is very much like a proprietary connector that connects a device to a computer. You have to buy the whole new PC card to support that connector. If the connector wire goes missing, you are dependent on the maker of that proprietary connector to provide you with a replacement, at which point, you will wish that you had bought a device with a USB connector instead. Only a few companies can pull this stunt off on their customers.

Java is designed with this in mind. You will see that many seemingly confusing rules are there precisely because they promote the development of interchangeable components. For example, an overriding method cannot throw a more generic exception than the one declared by the overridden method. On the other hand the constructor of a subclass cannot throw only a more specific exception than the one thrown by the constructor of the superclass. Think about that.

2. **To eliminate the scope for bugs** - Java designers have tried to limit or eliminate features that increase the possibility of bugs in a piece of code. **Pointer arithmetic** and **goto** are examples of that. They have also tried to add features that help writing bug-free code. **Automatic Garbage Collection** and **Generics** are examples of that.
3. **Make life easier for the programmer** - Many older languages such as **C/C++** were built with the flexibility and power to do various kinds of things. Putting restrictions on what a programmer can do was thought of as a bad idea. On the contrary, how to add features that will let the programmer do more and more was the focus. Every new language added more new features. For example, C++ has pointer arithmetic, global functions, operator overloading, extern declarations, preprocessor directives, unsigned data types, and so many other "features" that Java simply does not have. These are some really powerful tools in the hands of a C++ programmer. So, why doesn't Java have them? Java has actually gone in reverse with respect to features. Java does not have a lot of features that are found in languages that came before Java. The reason is simple. Java follows the philosophy of **making life simpler for the programmer**. Having more and more features is not necessarily a good thing. For example, having pointer arithmetic and manual allocation and deallocation of objects is powerful but it makes life hell for the programmer. Thus, unlike C++, there is no need to allocate memory in Java because all objects are created on the heap. Why should a programmer have to worry about something that can be taken care of by the language? Instead of focusing on mastering complicated features, the programmers should be spending more time in developing business logic. Thus, unlike C++, there is no need to deallocate memory in Java because Java performs garbage collection automatically.
Furthermore, the cost of maintaining complicated code cascades very quickly. A piece of code is written once but is read and is overwritten

numerous times. What is "clever" for one programmer becomes a nightmare for the one who follows that programmer.

Java has therefore, introduced several restrictions (I consider them features, actually) that make Java development substantially simpler overall. For example, in C++, there is no restriction on the file name in which a class exists. But in Java, a public class has to be in a file by that name. This is a restriction that doesn't seem to make sense at first because after compilation all classes are in the class files with the same names as that of the classes. But when you think about the organization of your source code and the ease of locating the code for a class, it makes perfect sense. Forcing every class to be coded in its own independent file would have been impractical and letting any class to be in a file by any name would have been too chaotic. So, forcing a public class to be in a file by that name is a nice balance.

4. **To become commercially successful** - "If Java is an Object-Oriented language, then why does it allow XYZ?" I see this question asked so many times. The answer is simple. Java was designed by pragmatic folks rather than idealistic ones. Java was designed at a time when C/C++ was extremely popular. Java designers wanted to create a language that remained faithful to OOP as much as possible but at the same time did not alienate the huge community of C/C++ programmers. This community was seen as potential Java developers and several compromises were made to make it easier for them to program in Java. Furthermore, not all non-OOP features are completely useless. Features that add value in certain commonly occurring situations find a place in Java even if they are not strictly OOP. Static methods is one such feature.

Then there is a matter of a "judgement call". Java designers are a bunch of smart people. Some things may not make complete sense from a purely logical or technical perspective but that's how they designed those things anyway. They made the decisions based on their experience and wisdom. For example, it is technically possible to design a compiler that can figure out the value of a non-final local variable with 100% certainty in the following code but the Java compiler does not flag an error for "unreachable code" here:

```
int x = 0;
```

```
if(x==0){
    throw new Exception();
}
x = 20; //unreachable code here but no compilation error
```

Sometimes there is a logical explanation for a seemingly confusing rule but the reason is not very well known. For example, the following code compiles fine even though the compiler knows that the code is unreachable:

```
class ConditionalCompilation {
    public static final boolean DEBUG = false;
    public void method(){
        if(DEBUG){
            System.out.println("debug statement here")
        }
    } //works
}
```

But a similar code causes the compiler to flag "unreachable code" error:

```
class ConditionalCompilation{
    public static final boolean DEBUG = false;
    public void method(){
        while(DEBUG){ //doesn't work
            System.out.println("debug statement her
e");
        }
    }
}
```

The reason is that historically, developers have used the combination of a boolean variable and an 'if' statement to include or exclude debug statements from the compiled code. A developer has to change the value of the flag at just one place to eliminate all debug statements. The 'if' statement in the code above works because Java designers decided to permitted this type of unreachable code so that conditional compilation could occur.

In conclusion, if you ever find yourself in a situation where you have to explain the reason behind a weird Java rule or concept, one of the above four would be your best bet. For example, reason 3 answers the question that you asked in the previous section, "why does Java allow fields and methods to be defined in an interface?". Why doesn't Java allow multiple inheritance? Reason 3. Why are all objects in Java rooted under Object class?. Reason 3.

1.3 Declaration and Definition

1.3.1 Declaration and Definition

In a technical interview, you should always know what you are talking about. A smart interviewer will catch you in no time if you talk loose. If you answer imprecisely, your credibility will evaporate faster than water in a frying pan. The certification exam requires the same attitude. You will lose marks for not knowing the basics.

It is surprising how many people use the terms **declaration** and **definition** incorrectly. So, let's just get this straight from the get-go. A declaration just means that something exists. A definition describes exactly what it is. For example,

```
class SomeClass //class declaration
```

```
//class definition starts
```

```
{  
    public void m1()//method declaration  
  
    //method definition starts  
  
    {  
    }  
    //method definition ends  
  
}  
//class definition ends
```

As you can see, a declaration provides only partial information. You can't make use of partial information. The definition makes it complete and thus, usable.

In terms of variables, Java doesn't have a distinction between declaration and definition because all the information required to define a variable is included in the declaration itself. For example,

```
int i; //this declaration cum definition is complete in itself
```

However, Java does make a distinction between variable declaration and variable initialization. Initialization gives a value to a variable. For example, `int i = 10;` Here `i` is defined as an `int` and also initialized to 10. `Object obj = null;` Here `obj` is defined as an `Object` and is also initialized to `null`. I will discuss more about declaration and initialization later.

The above is a general idea but you should be aware that there are multiple viewpoints with minor differences. Here are some links that elaborate more. You should go through at least the first link below.

[StackOverflow - Difference between declaration and definition in Java](#)

[JavaRanch - Declaration and Definition](#)

Can you now answer the question what does an interface contain - method declarations or method definitions?

Well, there was a time when interfaces contained only method declarations, but since Java 8, interfaces contain method declarations as well as definitions.

1.4 Object and Reference

1.4.1 Relation between a class, an object, and a reference

A **class** is a template using which **objects** are created. In other words, an object is an instance of a class. A class defines what the actual object will contain once created. You can think of a class as a cookie cutter. Just as you create cookies out of dough using a cookie cutter, you create objects out of memory space using a class.

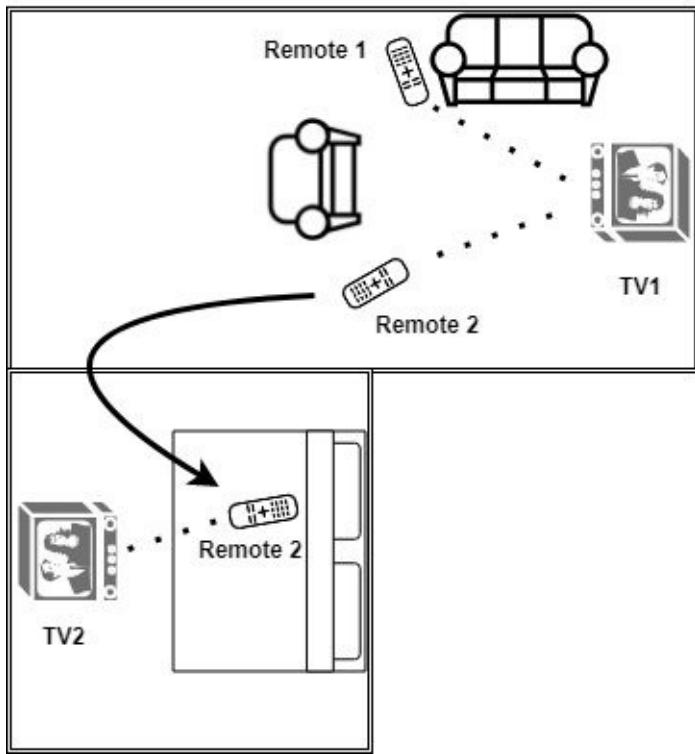
To access an object, you need to know exactly where that object resides in memory. In other words, you need to know the "address" of an object. Once you know the address, you can call methods or access fields of that object. It is this "address" that is stored in a reference variable.

If you have trouble understanding this concept, try to imagine the relationship between a Television (TV) and a Remote. The TV is the object and the Remote is the reference variable pointing to that object. Just like you operate the TV using the remote, you operate on an object using **a** reference pointing to that object. Notice that I did not say, "you operate on an object using **its** reference". That's because an object doesn't have any special reference associated with it. Just as a TV can have multiple remotes, an object can have any number of references pointing to it. One reference is as good as any other for the purpose of accessing that object. There is no difference between two references pointing to the same object except that they are two different references. In other words, they are mutually interchangeable.

Now, think about what happens when the batteries of a remote die. Does that

mean the TV stops working? No, right? Does that mean the other remote stops working? Of course not! Similarly, if you lose one reference to an object, the object is still there and you can use another reference, if you have it, to access that object.

What happens when you take one remote to another room for operating another TV? Does it mean the other remote stops controlling the other TV? No, right? Similarly, if you change one reference to point to some other object, that doesn't change other references pointing to the that object. The following picture illustrates the situation:



Let me now move to an example that is closer to the programming world. Let's say, you have the following code:

```
String str = "hello";
```

"hello" is the actual object that resides somewhere in the program's memory. Here, `str` is the remote and "hello" is the TV. You can use `str` to invoke methods on the "hello" object.

A program's memory can be thought of as a long array of bytes starting with `0` to `NNNN`, where `NNNN` is the location of last byte of the array. Let's say, within this memory, the object "`hello`" resides at the memory location `2222`. Therefore, the variable `str` actually contains just `2222`. It doesn't contain "`hello`". It is no different from an int variable that contains `2222` in that sense.

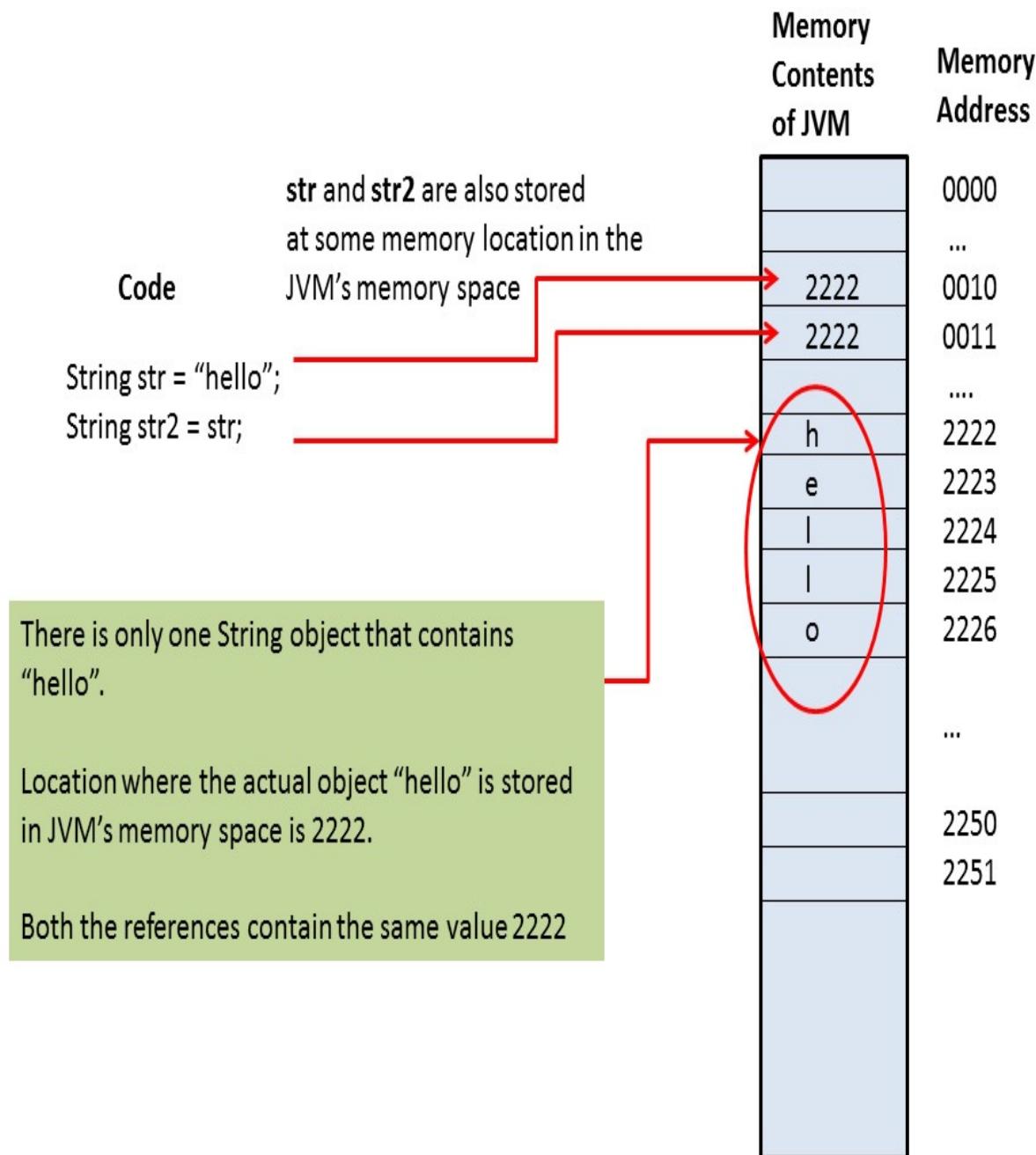
But there is a fundamental difference in the way Java treats **reference variables** and non-reference variables (aka **primitive variables**). If you print an `int` variable containing `2222`, you will see `2222` printed. However, if you try to print the value `str`, you won't see `2222`. You will see "`hello`". This is because the JVM knows that `str` is defined as a reference variable and it needs to use the value contained in this variable to go to the memory location and do whatever you want to do with the object present at that location. In case of an `int` (or any other primitive variable), the JVM just uses the value contained in the variable as it is. Since this is an important concept, let me give you another example to visualize it. Let us say Paul has been given `2222` dollars and Robert has been given bank locker number `2222`. Observe that both Paul and Robert have the same number but Paul's number denotes actual money in his hands while Robert doesn't have actual money at all. Robert has an address of the location that has money. Thus, Paul is like a primitive variable while Robert is like a reference variable.

Another important point is that you cannot make a reference variable point to a memory location directly. For example, you can set the `int` variable to `2250` but you can't do that to `str` i.e. you can't do `str = 2250`. It will not compile. You can set `str` to another string and if that new string resides at a memory location `2250`, `str` will indeed contain `2250` but you can't just store the address of any memory location yourself in any reference variable.

As a matter of fact, there is no way in Java to see and manipulate the exact value contained in a reference variable. You can do that in C/C++ but not in Java because Java designers decided not to allow messing with the memory directly.

You can have as many references to an object as you want. When you assign one reference to another, you basically just copy the value contained in one reference into another. For example, if you do `String str2 = str;` you are just copying `2222` into `str2`. Understand that you are not copying "`hello`" into `str2`. There is only one string containing "`hello`" but two reference

variables referring to it. Figure 1 illustrates this more clearly.



If you later do `str = "goodbye";` you will just be changing `str` to point to a different string object. It does not affect `str2`. `str2` will still point to the string "`hello`".

The question that should pop into your head now is what would a reference

variable contain if it is not pointing at any object? In Java, such a variable is said to be **null**. After all, as discussed above, a reference variable is no different from a primitive variable in terms of what it contains. Both contain a number. Therefore, it is entirely possible that a reference that is not pointing to any object may actually contain the value 0. However, it would be wrong to say so, because a reference variable is interpreted differently by the JVM. A particular implementation of JVM may even store a value of -1 in the reference variable if it does not point to any object. For this reason, a reference variable that does not point to any object is just null. At the same time, a primitive variable can never be **null** because the JVM knows that a primitive variable can never refer to an object. It contains a value that is to be interpreted as it is. Therefore,

```
String str = null; //Okay  
int n = 0; //Okay
```

```
String str = 0; //will not compile  
int n = null; //will not compile.
```

1.5 static and instance

1.5.1 Static and Instance [↳](#)

You will read the word "**static**" a lot in Java tutorials or books. So, it is better to form a clear understanding of this word as soon as possible. In English, the word static means something that doesn't change or move. From that perspective, it is a misnomer. Java has a different word for something that doesn't change: **final**. I will talk more about "final" later.

In Java, static means something that belongs to a class instead of belonging to an instance of that class. As we discussed in the "Object and Reference" section, a class is just a template. You can instantiate a class as many times as you want and every time you instantiate a class you create an instance of that class. Now, recall our cookie cutter analogy here. If a class is the cookie cutter, the fields defined in the class are its patterns. Each instance of that class is then the cookie and each field will be imprinted on the cookie - except the fields defined as

static. In that sense, a static member is kind of a tag stuck to a cookie cutter. It doesn't apply to the instances. It stays only with the class.

Consider the following code:

```
class Account {  
    String accountNumber;  
    static int numberofAccounts;  
}  
...  
//Create a new Account instance  
  
Account acct1 = new Account();  
  
//This Account instance has its own accountNumber field.  
  
acct1.accountNumber = "A1";  
  
//But the numberofAccounts fields does not belong to the instance, it belongs to the Account class  
  
Account.numberofAccounts = Account.numberofAccounts +  
1;  
  
//Create another Account instance  
  
Account acct2 = new Account();  
  
//This instance has its own accountNumber field.  
  
acct2.accountNumber = "A2";
```

//the following line accesses the same class field and therefore, numberOfAccounts is incremented to 2

```
Account.numberOfAccounts = Account.numberOfAccounts +  
1;
```

Important points about static -

1. static is considered a non object-oriented feature because as you can see in the above code, static fields do not belong to an object. So, why does Java have it? Check out the "Why is something so?" section.
2. Here is a zinger from Java designers - even though static fields belong to a class and should be accessed through the name of the class, for example, `Account.numberOfAccounts`, it is not an error if you access it through a variable of that class, i.e., `acct1.numberOfAccounts`. Accessing it this way doesn't change its behavior. It is still static and belongs to the class. Therefore, `acct2.numberOfAccounts` will also refer to the same field as `acct1.numberOfAccounts`. This style only causes confusion and is therefore, strongly discouraged. Don't write such code. Ideally, they should have disallowed this usage with a compilation error.
3. Just like fields, methods can be static as well. A static method belongs to the class and can be accessed either using the name of the class or through a variable of that class.
4. The opposite of static is instance. There is no keyword by that name though. If a class member is not defined as static, it is an instance member.

1.6 Stack and Heap

1.6.1 Stack and Heap

When you execute a program, the Operating System (OS) allocates and gives memory to that program. This memory is used by the program to keep its variables and data. For example, whenever you create a variable, its value needs

to be preserved as long as the program wants to use it. The program uses its allocated memory to keep it. A program may ask the OS for more memory if it requires and the OS will oblige if the OS has free memory available. A program may also release some memory that it does not want back to the OS. Once the OS gives out a chunk of memory to the program, it is the responsibility of the program to manage it. Once the program ends, this memory is released and goes back to the OS. This is basically how any executable program works.

Now, think about the following situation. Your program has a method that prints "hello" 100 times. Something like this:

```
public class Test{
    private String str = new String("hello"); //Using new is not a good way to create strings, but bear with me for a moment.

    public void print(){
        int i = 0;
        while(i++<100){
            System.out.println(this.str);
        }
    }

    public static void main(String[] args){
        Test t = new Test();
        t.print();
    }
}
```

In the above class, it is the main method that calls the print method but there could also be another class, which could make use of the same print method to print **hello** a 100 times. When the print method is called, it creates the variable **i** to keep track of the number of times the while loop has iterated. This variable needs to be kept somewhere as long as the print method runs. Similarly, it uses the variable **str** to print the string that you want the print method to print.

The question is, what happens when the print method ends? The variable **i** has

served its purpose and is not required anymore. It is not used anywhere except within this method. Therefore, it need not be kept longer than the execution of the print method. But the variable `str` still can be used whenever the print method is called. Therefore, the value of `str` needs to be kept irrespective of the execution lifetime of the print method.

It should now be clear that a program needs two kinds of memory spaces to keep the stuff. One for temporary stuff that can be cleaned up as soon as a method call ends and one for permanent stuff that remains in use for longer than a single method call. The space for storing the temporary stuff is called **Stack space** and the space for storing all other stuff is called **Heap space**. The reason why they are called Stack and Heap will be clear soon.

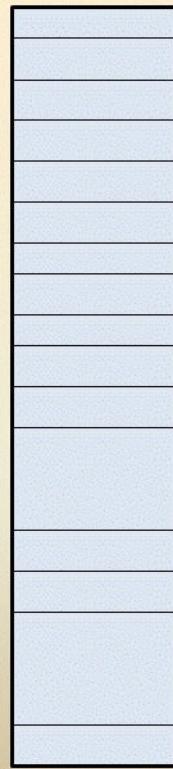
In Java, each thread is given a fixed amount of stack space. In the above example, when you execute the program, a main thread is created with a fixed amount of stack space. All this space is initially empty. When this thread invokes the main method, all the temporary variables created by this method are kept on this stack. In the above example, the main method gets one reference variable named `args` and inside the method, it creates another reference variable named `t`. (Note that since `args` and `t` are reference variables, they contain the address of the location where actual objects referred to by `args` and `t` respectively reside). Therefore, the stack fills up by the amount of space required by these reference variables. Before the main method ends, it calls the print method on the reference `t`. Since print is an instance method, a variable named "`this`" is automatically put on the stack for it so that the method can access the instance fields on this object. The variable `this` is also a reference variable and it contains the address of the location where the Test object actually resides. The print method creates one more temporary variable `i`. This variable is also kept on the same stack on top of `this`. Thus, the stack fills up a little more by amount of space required for storing two variables. When the print method ends, the space used for `this` and `i` is reverted back to the stack and the stack is thus, emptied out a little. The control goes back to the main method. This method also ends and the space used for storing `args` and `t` is cleaned up. There is nothing left on the stack anymore at this point. Thus, the stack is completely empty again. The following five figures illustrate this process.

1

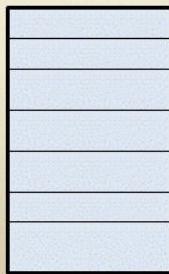
Main thread's
Stack space



Heap Space
of JVM



Some other
thread's Stack space



JVM's Memory Space

2

Main thread's
Stack space

0010 (t)
3333 (args)

Location where
Test Object
resides.

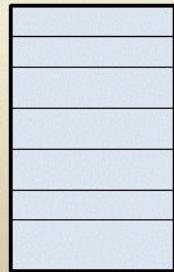
Heap Space
of JVM

0011(str)
h
e
l
l
o

0010
0011
0012
0013
0014
0015

...

Some other
thread's Stack space



JVM's Memory Space

3

Main thread's Stack space
100 (i)
0010 (this)
0010 (t)
3333 (args)

Location where
Test Object
resides.

Heap Space
of JVM

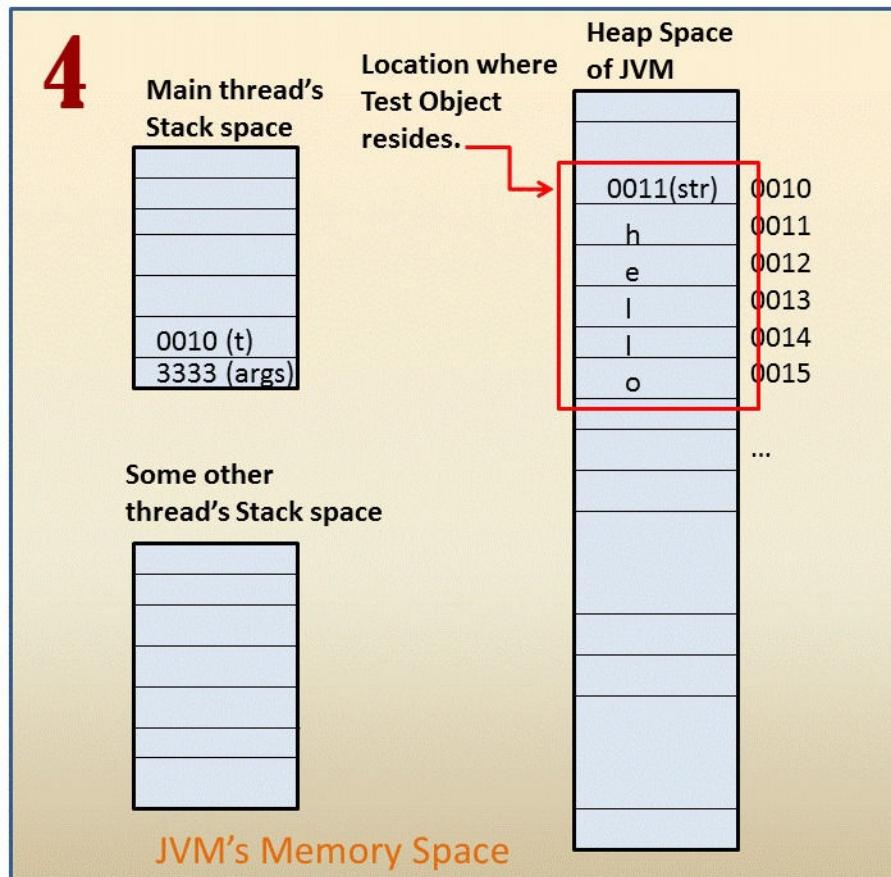
0011(str)	0010
h	0011
e	0012
l	0013
l	0014
o	0015
	...

Some other
thread's Stack space

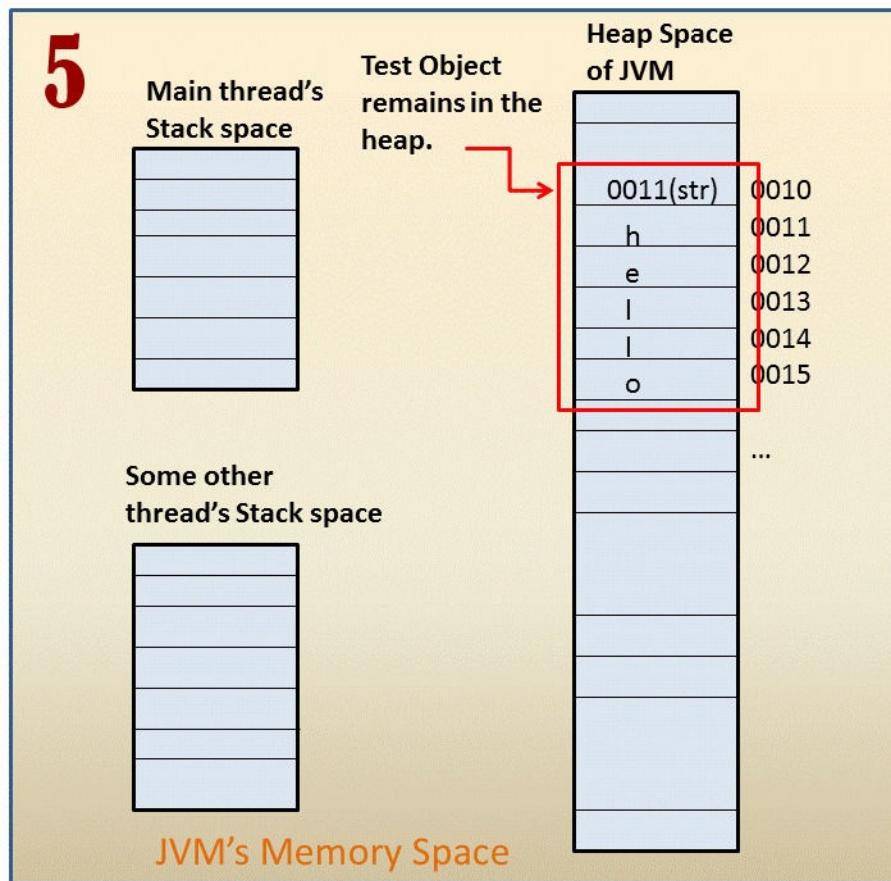


JVM's Memory Space

4



5



As you can observe, the stack space looks like a stack of chips that are kept one on top of the other. The temporary variables created by a method are added on top of the stack one by one as and when they are created. As soon as the method ends, all those variables are removed from the top. Observe that they are removed only after the method ends. If the method calls another method, then the variables created by the called method are pushed on to the same stack on top of the variables stored by the caller method. When a thread dies, its stack space is reverted back to the JVM. Since this space behaves like a stack, it is called **stack space**.

The **heap space**, on the other hand, is, well, like a heap! Objects lie in a heap just as they please. JVM goes to great lengths to organize the heap space. Organization of the heap space is an advanced topic and is very important when you analyse the performance of an application. However, it is not relevant for the exam and so, I will not discuss it. From the program perspective, there is not much of an organization in a heap.

Whenever any object is created anywhere in the code (i.e. whether in a method or in a class), the JVM allocates space for that object on the heap and puts its contents in that space. In Java, a program never releases this space explicitly. It is managed by the JVM. Again, recall that an object can only be accessed using its reference. For a method to access an object, it must use a reference that points to that object. It could get that reference either from a variable kept on its stack space (if the object was created in this method itself) or through a reference to another object whose reference is kept on the stack space (if that object has a reference to the required object). In either case, a method has to start with a reference that exists on its stack space. If there is no reference on any stack space through which an object can be accessed directly or indirectly, that object is considered garbage. It is cleaned up automatically by the JVM using a garbage collector.

*Recall from our discussion on **References and Objects** that a reference is merely a variable that stores the address of the location where the actual object is stored. In that sense, a reference variable is no different than an int variable. They both store a number. A reference variable stores a number that indicates the memory location where you can find the actual object, while an int variable stores a number that is interpreted as a number. It doesn't indicate anything else. If you create a variable in a method, whether a reference variable or a primitive variable, it is kept on the stack but when you create an object, that object is stored on the heap.*

Typically, an object is created using the new keyword. But Java treats Strings as special, and so, you can create String objects even without the new keyword. Thus, whether you do "hello" or new String("hello"), in both the cases, a String object containing "hello" is created on the heap.

Points to remember:

- Local variables are always kept on the stack. Objects are always stored in the heap. (An optimizing JVM may allocate an object on the stack space, but it is an internal detail of the JVM and you need not worry about it. For all we care, objects are always on the heap.)
- JVM may have several threads. Each thread is given a fixed amount of stack space that is dedicated completely and exclusively to that thread. No one but that thread can access its stack space. This is called "**stack semantics**". A thread accesses its stack space by creating and using variables. There is no other special way of accessing the stack space.
- Heap space is shared among all threads. Any thread can use space on a heap by creating objects. Since heap space is shared, it is possible for one thread to access objects created by another if it has a reference to that object. This is called "**heap semantics**".
- Stack space is limited for a program. So, if you have a huge chain of method calls where each method creates a lot of temporary variables (**recursion** is a good example), it is possible to run out of stack space. In Java, the default stack space size is **64KB** but it can be changed at the time of executing the program using command line option **-Xss**. Heap space is unlimited from the program's perspective. It is limited only by the amount of space available on your machine.
- Only temporary variables, i.e., variables created in a method (also known as local variables and automatic variables) are created on the stack space. Everything else is created on the heap space. If you have any doubt, ask yourself this question - is this a temporary variable created in a method? Yes? Then it is created on the stack. No? Then it is on the heap. Actual objects are **ALWAYS** created on the heap.
- When a method is invoked by a thread, it uses the thread's stack space to keep its temporary variables.
- Variables added to the stack space by a method are removed from the stack when that method ends. Everything else created by a method is left on the heap even after the method ends.

[Here is a good discussion on Stack and Heap and why its understanding is important .](#)

1.7 Conventions

1.7.1 What is a Convention?

You add a 15% tip to your bill at a restaurant. There is no law about that. Nobody is going to put you in jail if you add nothing for a tip. But you still do it because it is a convention. A lot of things in the world are based on convention. In India, you drive on the left side of the road. This is a convention. It has nothing to do with being technically correct. Indeed, people are fine driving on the right side of the road in the US. But if you drive on the right side of the road in India, you will cause accidents because that is not what other people expect you to do.

It is the same in the programming world. As a programmer, you are a part of the programmer community. The code that you write will be read by others and while developing your code, you will read and use code written by others. It saves everyone time and effort in going through a piece of code if it follows conventions. It may sound ridiculous to name **loop variables** as `i`, `j`, or `k`, but that is the convention. Anyone looking at a piece of code with a variable `i` will immediately assume that it is just a temporary variable meant to iterate through some loop.

If you decide to use a variable named `i` for storing some important program element, your program will work fine but it will take other people time to realize that and they will curse you for it.

If you are still unconvinced about the importance of conventions in programming, let me put it another way. If I ask you to write some code in an interview and if you use a variable named `hello` as a loop variable, I will not hire you. I can assure you that most interviewers will not like that either. Conventions are that important.

1.7.2 Conventions in Java

Some of the most important **conventions in Java** are as follows:

1. **Cases** - Java uses "Camel Case" everywhere with minor differences.
 1. Class names start with an uppercase letter. For example, `ReadOnlyArrayList` is a good name but `Readonlyarraylist` is not.
 2. Package names are generally in all lowercase but they also may be in camel case starting with a lower case letter. For example, `datastructures` is a good package name but `DataStructures` is not.
 3. variable names start with a lower case and may include underscores. For example, `current_account` is a good variable name.
2. **Naming** - Names should be meaningful. A program with a business purpose should not have variables with names such as `foo`, `bar`, and `fubar`. Although, such nonsensical names are used for illustrating or explaining code in sample programs where names are not important.
3. **Package names** use a reverse domain name combined with a group name and/or application name. For example, if you work at Bank of America's Fixed Income Technologies division and if you are developing an application named FX Blotter, all your packages for this application may start with the name `com.bofa.fit.fxblotter`. The full class name for a class named `ReadOnlyArrayList` could be -
`com.bofa.fit.fxblotter.dataStructures.ReadOnlyArray`
The reason for using a reverse domain name is that it makes it really easy to come up with globally unique package names. For example, if a developer in another group also creates his own `ReadOnlyArrayList`, the full name of his class could be
`com.bofa.derivatives.dataStructures.ReadOnlyArrayList`. There would be no problem if a third developer wants to use both the classes at the same time in his code because their full names are different. The important thing is that the names turned out to be different without any of the programmers ever communicating with each other about the name of their classes. The names are unique globally as well because the domain names of companies are unique globally.

1.8 Compilation and Execution

1.8.1 Compilation and Execution [↳](#)

Let us go over the basics really quickly. You know that a Java source file is compiled into a Java class file and a class file is what is executed by the JVM. You also know that you can organize your Java classes into packages by putting a package statement at the top of a Java source file. The package name plus the class name is called **Fully Qualified Class Name** or **FQCN** for short, of a Java class. For example, consider the following code:

```
package accounting;

public class Account{

    private String accountNumber;

    public static void main(String[] args){

        System.out.println("Hello 1 2 3 testing...");

    }

}
```

In the above code, `accounting.Account` is the fully qualified class name of the class. This long name is the name that you need to use to refer to this class from a class in another package. Of course, you can "import" `accounting` package and then you can refer to this class by its short name `Account`. The purpose of packages is to organize your classes according to their function to ease their maintenance. It is no different from how you organize a physical file cabinet where you keep your tax related papers in one drawer and bills in another.

Packaging is meant solely for ease of maintenance. The Java compiler and the JVM don't really care about it. You can keep all your classes in one package for all that matters.

Let us create **Account.java** file and put it in your work folder (for example, **c:\javatest**). Copy the above mentioned code in the file and compile it as follows:

```
c:\javatest>javac Account.java
```

You should see **Account.class** in the same folder. Now, let us try to run it from the same folder:

```
c:\javatest>java Account
```

You will get the following error:

```
Exception in thread "main" java.lang.NoClassDefFoundError:  
    at Account (wrong name: accounting/Account)
```

Of course, you need to use the long name to refer to the class, so, let's try this:

```
c:\javatest>java accounting.Account
```

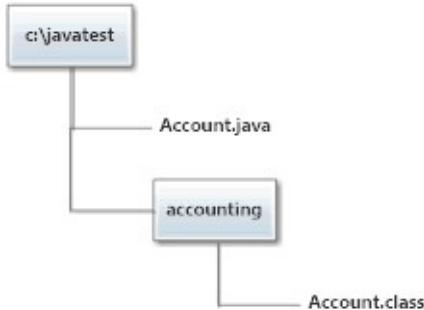
You will now get the following error:

```
Error: Could not find or load main class accounting.Ac  
count
```

Okay, now delete the **Account.class** file and compile the Java code like this:

```
c:\javatest>javac -d . Account.java
```

You should now have the directory structure as shown below:



Now, run it like this:

```
c:\javatest>java -classpath . accounting.Account
```

You should see the following output:

```
Hello 1 2 3 testing...
```

What is going on? Well, by default the Java compiler compiles the Java source file and puts the class file in the same folder as the source file. But the Java command that launches the JVM expects the class file to be in a directory path that mimics the package name. In this case, it expects the **Accounting.class** file to be in a directory named **accounting**. The accounting directory itself may lie anywhere on your file system but then that location must be on the classpath for the JVM to find it.

One of the many command line options that **javac** supports is the **-d** option. It directs the compiler to create the directory structure as per the package name of the class and put the class file in the right place. In our example, it creates a directory named **accounting** in the current directory and puts the class file in that directory. The dot after **-d** in the javac command tells the compiler that the dot, i.e., the current directory is the target directory for the resulting output. You

can replace dot with any other directory and the compiler will create the new package based directory structure there. For example, the command `c:\javatest>javac -d c:\myclassfiles Account.java` will cause the accounting directory to be created in `c:\myclassfiles` folder.

Now, at the time of execution you have to tell the JVM where to find the class that you are asking it to execute. The `-classpath` (or its short form `-cp`) option is meant exactly for that purpose. You use this option to specify where your classes are located. You can specify multiple locations here. For example, if you have a class located in `c:\myclassfiles` directory and if that class refers to another class stored in `c:\someotherdirectory`, you should specify both the locations in the classpath like this:

```
c:\java -classpath c:\myclassfiles;c:\someotherdirectory accounting.Account
```

Observe that when you talk about the location of a class, it is not the location of the class file that you are interested in but the location of the directory structure of the class file. Thus, in the above command line, `c:\myclassfiles` should contain the `accounting` directory and not `Account.class` file.

`Account.class` should be located inside the `accounting` directory. The JVM searches in all the locations specified in the `-classpath` option for classes.

*Note: On *nix based systems, you need to use colon (:) instead of semi-colon (;) and forward slash (/) instead of back slash (\).*

The Java command scans the current directory for class files (and packages) by default, so, there is usually no need to specify "dot" in the -classpath option. I have specified it explicitly just to illustrate the use of the -classpath option.

Compiling multiple source files at once ↗

Let's say you have two source files **A.java** and **B.java** in **c:\javatest** directory with the following contents:

Contents of A.java:

```
package p1;
import p2.B;
public class A{
    B b = new B();
}
```

Contents of B.java:

```
package p2;
public class B{}
```

Open a command prompt, **cd** to **c:\javatest**, and compile **A.java**. You will get a compilation error because class A depends on class B. Obviously, the compiler will not be able to find **B.class** because you haven't compiled **B.java** yet! Thus, you need to compile **B.java** first. Of course, as explained before, you will need to use the **-d .** option while compiling **B.java** to make javac create the appropriate directory structure along with the class file in **c:\javatest** directory. This will create **B.class** in **c:\javatest\p2** directory. Compilation of **A.java** will now succeed.

The point is that if you have two classes where one class depends on the other, you need to compile the source file for the independent class first and the source file for the dependent class later. However, most non-trivial Java applications are composed of multiple classes coded in multiple source files. It is impractical to determine the sequence of compilation of the source files manually. Moreover, it is possible for two classes to be circularly dependent on each other. Which source file would you compile first in such a case?

Fortunately, there is a simple solution. Just let the compiler figure out the dependencies by specifying all the source files that you want to compile at once. Here is how:

```
javac -d . A.java B.java
```

But again, specifying the names of all the source files would also be impractical. Well, there is a solution for this as well:

```
javac -d . *.java
```

By specifying `*.java`, you are telling the compiler to compile all Java files that exist in the current directory. The compiler will inspect all source files, figure out the dependencies, create class files for all of them, and put the class files in an appropriate directory structure as well. Isn't that neat?

If your Java source files refer to some preexisting class files that are stored in another directory, you can state their availability to `javac` using the same `-classpath` (or `-cp`) option that we used for executing a class file using the `java` command.

I strongly advise that you become comfortable with the compilation process by following the steps outlined above.

1.8.2 Running a single file source code program [↳](#)

Java designers felt that the two step compilation and execution of Java programs is too tedious when you are trying to execute simple test programs. To make it simple, Java 11 allows you to directly execute a Java source file using the `java` command. For example, consider the following contents of `TestClass.java` file in `C:\javatest` directory:

```
public class TestClass {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

You can execute this file directly from the command line using the following command:

```
java TestClass.java
```

It will print **Hello World!**.

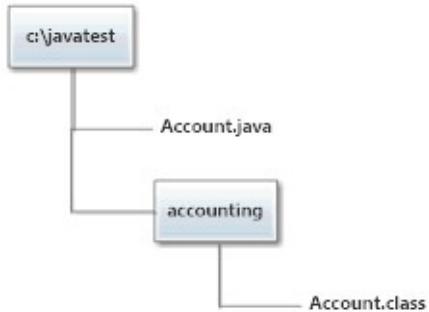
The only restriction with this approach is that your Java code must not refer to code in any other Java file. You can have as many classes in the file as you want but the first class that appears in the file must contain the main method because that is the main method that the JVM will pick to execute.

Although this compilation cum execution technique is **not on the exam**, I have included it here because it will save you a lot of time while trying out various concepts using single file programs. You have to be careful about distinguishing between compilation failure and exception at run time though. If you see **error: compilation failed** on the console, it is a compilation error, otherwise, it is an exception during execution.

1.8.3 Packaging classes into Jar ↗

It is undoubtedly easier to manage one file than multiple files. An application may be composed of hundreds or even thousands of classes and if you want to make that application downloadable from your website, you cannot expect the users to download each file individually. You could zip them up but then the users would have to unzip them to be able to run the application. To avoid this problem, Java has created its own archive format called "Java Archive", which is very much like a zip file but with an extension of jar.

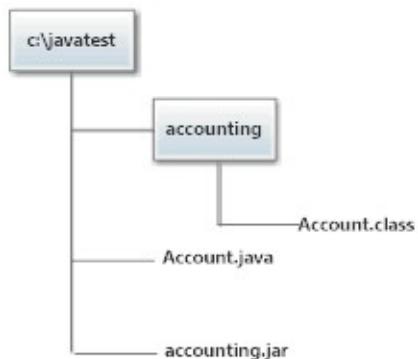
Creating a jar file that maintains the package structure of class files is quite easy. Let us say you have the directory structure shown below:



Go to the command prompt, **cd** to **c :\javatest** directory and run the following command:

```
jar -cvf accounting.jar accounting
```

This command tells the jar utility to create **accounting.jar** file and include the entire directory named **accounting** in it along with its internal files and directories. You should now have the directory structure shown below:



Assuming that you are still in **c :\javatest** directory on your command prompt, you can now run the class through the jar file like this:

```
java -classpath .\accounting.jar accounting.Account
```

Note that you must maintain the package structure of the class while creating the jar file. If you open **accounting.jar** in **WinZip** or **7zip**, you will see that this jar contains **Account.class** under **accounting** directory.

Besides the class files, the Jar file allows you to keep information about the contents of the jar file within the jar file itself. This information is kept in a

special file is called **MANIFEST.MF** and is kept inside the **META-INF** folder of the jar file. (This is just like airlines using a "manifest" to document the cargo or a list of passengers on a flight.) For example, you can specify the entry point of an application which will allow you to run a Jar file directly (from the command line or even by just double clicking the jar file in your file explorer) without having to specify the class name containing the main method on the command line. Typical contents of this file are as follows -

```
Manifest-Version: 1.0  
Created-By: 1.7.0_09-b05 (Oracle Corporation)  
Main-Class: accounting.Account
```

You can actually go ahead and create **mymanifest.txt** file with the above mentioned contents in **C:\javatest** directory and use the following command to create the jar:

```
jar -cvfm accounting.jar mymanifest.txt accounting
```

c is for create, **v** is for verbose (i.e. display detailed information on command line), **f** is for the output file, and **m** is the name of the file the contents of which have to be included in the jar's manifest. Notice that the name of the manifest file on the command line is not important. Only the contents of the file are important. This command will automatically add a file named **MANIFEST.MF** inside the **META-INF** folder of the jar file.

Once you have this information inside the jar file, all you need to do to run the program is to execute the following command on the command line:

```
java -jar accounting.jar
```

Although packaging classes into Jar files is not on the exam as such, you will need to know about it from the perspective of execution of modules, which is on the exam.

1.8.4 Compilation error vs exception at run time [↳](#)

Understanding whether something will cause a failure during compilation or will cause an exception to be thrown at run time is important for the exam because a good number of questions in the exam will have these two possibilities as options. Beginners often get frustrated while trying to distinguish between the two situations. It will get a little easier if you keep the following three points in mind:

1. First and foremost, it is the compiler's job to check whether the code follows the syntactical rules of the language. This means, it will generate an error upon encountering any syntactical mistake. For example, Java requires that the package statement, if present, must be the first statement in the Java code file. If you try to put the package statement after an import statement, the compiler will complain because such a code will be syntactically incorrect. You will see several such rules throughout this book. Yes, you will need to memorize all those. If you use an IDE such as Eclipse, NetBeans, or IntelliJ, you should stop using it because you need to train your brain to spot such errors instead of relying on the IDE. Using Notepad to write and using the command line to compile and run the test programs is very helpful in mastering this aspect of the exam.
2. Besides being syntactically correct, the compiler wants to make sure that the code is logically correct as well. However, the compiler is limited by the fact that it cannot execute any code and so, it can never identify all the logical errors that the code may have. Even so, if, based on the information present in the code, the compiler determines that something is patently wrong with the code, it raises an error. It is this category of errors that causes the most frustration among beginners. For example, the statement **byte b = 200;** is syntactically correct but the compiler does not like it. The compiler knows that the value **200** is too big to fit into a **byte** and it believes that the programmer is making a logical mistake here. On the other hand, the compiler okays the statement **int i = 10/0;** even though you know just by looking at the code that this statement is problematic.

3. The JVM is the ultimate guard that maintains the integrity and type safety of the Java virtual machine at all times. Unlike the compiler, the JVM knows about everything that the code tries to do and it throws an exception (I am using the word exception in a general sense here and not referring to the `java.lang.Exception` class) as soon as it determines that the action may damage the integrity or the type safety of the JVM. Thus, any potentially illegal activity that escapes the compiler will be caught by the JVM and will result in an exception to be thrown at run time. For example, dividing a number by zero does not generate any meaningful integral value and that is why the JVM throws an exception if the code tries to divide an integral value by zero.

Honestly, this is not an easy topic to master. The only way to get a handle on this is to know about all the cases where this distinction is not so straightforward to make. If you follow this book, you will learn about all such rules, their exceptions, and the reasons behind them, that are required for the exam.

1.9 Nomenclature

1.9.1 Commonly used terms in Java development

During your programming career you will be reading a lot. It could be books, articles, blogs, manuals, tutorials, and even discussion forums. You will also be interacting with other Java developers in various roles such as interviewers, team members, architects, and colleagues. To make the most out of these interactions, it is very important to form a clear and precise understanding of commonly used terms.

I will explain the commonly used phrases, names, and terminology in the Java world.

1. **Class** - Unless stated otherwise or unless clear from the context, the term class includes class, interface, and enum. Usually, people mean "type" when they say "class". You should, however, always try to be precise and use the term class only for class.

2. **Type** - Type refers to classes, interfaces, enums, and also primitive types (byte, char, short, int, long, float, double, and boolean).
3. **Primitive types** - byte, char, short, int, long, float, double, and boolean are called primitive types because they just hold data and have no behavior. You can perform operations on them but you cannot call methods on them. They do not have any property or state other than the data value that they contain. You access them directly and never through references.
4. **Reference types** - Classes, Interfaces, and Enums are called reference types because you always refer to them through references and never directly. Unlike primitive types, reference types have behavior and/or state.
5. **Top-level reference types** - Classes, interfaces, or enums that are defined directly under a package are called top-level classes, interfaces, or enums.
6. **Nested reference types** - Classes, interfaces, and enums that are defined inside another class, interface, or an enum are called nested classes, interfaces, or enums.
7. **Inner reference types** - Non-static nested classes, interfaces, and enums that are called inner classes, interfaces, or enums.
8. **Local reference types** - Nested reference types that are defined inside a method (or inside another code block but not directly inside a class, interface, or enum) are called local classes, interfaces, or enums.
9. **Anonymous classes** - This is a special case of a nested class where just the class definition is present in the code and the complete declaration is automatically inferred by the compiler through the context. An anonymous class is always a nested class and is never static.
10. **Compile time vs run time (i.e. execution time)** - You know that there are two steps in executing Java code. The first step is to compile the Java code using the Java compiler to create a class file and the second step is to execute the JVM and pass the class file name as an argument. Anything that happens while compiling the code such as generation of compiler warnings or error messages is said to happen during "compile time".

Anything that happens while executing the program is said to happen during the "run time". For example, syntax errors such as a missing bracket or a semicolon are caught at compile time while any exception that is generated while executing the code is thrown at run time. It is kind of obvious but I have seen many beginners posting questions such as, "why does this code throw the following exception when I try to compile it?", when they really mean, "why does this code generate the following error message while compilation?" Another common question is, "why does this code throw an exception even after successful compilation?" Successful compilation is not a guarantee for successful execution! Although the compiler tries to prevent a lot of bugs by raising warnings and error messages while compilation, successful compilation really just means that the code is syntactically correct.

11. **Compile-time constants** - Normally, it is the JVM that sets the values of variables when a program is executed. The compiler does not execute any code and it has no knowledge of the values that a variable might take during the execution of the program. Even so, in certain cases, it is possible for the compiler to figure out the value of a variable. If a compiler can determine the value that a variable will take during the execution of the program, then that variable is actually a compile-time constant. For example, if you define an int variable as `final int x = 10;` then `x` is a compile time constant because the compiler knows that `x` will always have a value of 10 at run time. Similarly, literals such as the numbers 1, 2, and 3, or the characters written in code within single quotes such as '`a`', or boolean values `true` and `false`, are all compile time constants because the compiler knows that these values will never change.

I will refer to these terms and will also discuss the details of these terms throughout the course so, it will be helpful if you keep the basic idea of these terms in mind.

1.10 Java Identifiers

1.10.1 Java Identifiers

Java has specific rules to name things such as variables, methods, and classes. All these names belong to a category of names called "identifiers".

Java defines an identifier as an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. An identifier cannot have the same spelling as a Java keyword or a literal (i.e. `true`, `false`, or `null`).

For example, the following variable names are invalid:

`int int; //int is a keyword`

`String class; //class is keyword`

`Account 1a; //cannot start with a digit`

`byte true; //true is a literal`

Java letters include uppercase and lowercase ASCII Latin letters A-Z (\u0041-\u005a), and a-z (\u0061-\u007a), and, for historical reasons, the ASCII underscore (_ or \u005f) and dollar sign (\$ or \u0024). The "Java digits" include the ASCII digits 0-9 (\u0030-\u0039).

Older versions of the exam tested candidates on identifying valid identifiers. However, the current exam has moved away a bit from making the candidate a human compiler and does not include this topic. You should still have a basic idea about what an identifier is though because this concept applies to all kind of names in Java.

Chapter 2 Creating a Simple Java Program

Exam Objectives

- Create an executable Java program with a main class
- Compile and run a Java program from the command line
- Create and import packages

2.1 Create an executable Java program with a main class

2.1.1 The main method

Let's get one thing out of the way first. Java classes are not **executables** . You cannot "execute" Java classes. The **Java Virtual Machine** (JVM) is an executable. You execute the JVM. You actually pass the FQCN of a Java class as an argument to the JVM. When the JVM runs, it loads the given class and looks for a specific method in that class. If it finds that method, it passes control to that method and this method then becomes the in-charge from there onward. If the JVM doesn't find that specific method, it errors out. In common parlance, we call it as executing or running a Java class or a program.

The method that the JVM is hardwired to look for in the class is called **the "main" method** and this method has a very specific signature - its name must be **main** and it must take exactly one parameter of type **String array** . In addition to this, it must return **void** , must be **public** and must also be **static** . It is free to declare any exception in its **throws** clause. If your class has such a method, the JVM can invoke this method and therefore, it is possible to execute the class. The meaning of **void** , **public** , **static** , and **throws** will be clear as you proceed through the book but for now, just assume that this is how

the main method has to be.

Examples of a valid main method:

1. `public static void main(String[] args){ }` - This is the version that you will see most of the time.
2. `public static void main(String... args){ }` - Note that `String...` is the same as `String[]` as far as the JVM is concerned. The three dots syntax is called varargs. I will talk more about it later.
3. `public static void main(String args[]) throws Exception{ throw new Exception(); }` - The main method is allowed to throw any exception.

Examples of an invalid main method:

1. `static void main(String abc[]){ }` - Invalid because it is not public.
2. `public void main(String[] x){ }` - Invalid because it is not static.
3. `public static void main(String[] a, String b){ }` - Invalid because it doesn't take exactly one parameter of type String array.
4. `static void Main(String[] args){ }` - Invalid because it is not public and the name starts with a capital M. Remember that Java is case sensitive.

Note that all of the above methods are valid methods in their own right. It is not a compilation error if you have these methods in your class. But they cannot be accepted as the "main" method. JVM will complain if you try to execute a class on the basis of these methods. JVM has gotten smarter over the years and in Java 11, it gives out a very helpful error message that explains the problem with your main method. For example, if it is not static, you will see the following message:

```
Error: Main method is not static in class TestClass,  
please define the main method as:  
public static void main(String[] args)
```

Examples of really **weird looking** main methods:

1. `public static native void main(String[] args);` - Out of scope for the exam, but good to know. This is a valid main method. A native method means you are going to implement this method in a separate executable library which will be linked at run time. In this case, the JVM will look for the implementation of your main method in a dynamically linked library. If it finds an implementation, all is good. If not, then it will throw an error saying it is unable to find the implementation to this native method.
2. `public abstract static void main(String[] args);` - Invalid because static methods cannot be abstract.

Just like with any other method, it is possible to have **overloaded** main methods in a Java class. I will talk overloading in detail later, but for now, it means having multiple methods with same name but different parameters. The JVM looks for a specific main method as described above. All other main methods have no special meaning for the JVM.

Many questions in the certification exam assume the presence of the main method. You may be given a code snippet and asked to determine the output. If you don't see any main method in the given code you need to assume that there is a main method somewhere that is invoked by the JVM and the given code or a method is invoked through that method.

2.1.2 Command line arguments

It is possible to provide any number of arguments while executing a class by specifying them right after the name of the class on the command line. The arguments must be separated by a space character. For example, if you want to pass three arguments to your class named TestClass, your command would be:

`java TestClass a b c`

The JVM passes on the arguments specified on the command line to the main method through its `String[]` parameter. In other words, the `String[]` parameter of the main method contains the arguments specified on the command line. An important implication of this is that all the arguments are passed to the main method as Strings. For example, if your command line is `java TestClass 1`, the main method will get a String array with one String element containing 1 and not an int 1.

Let me now present to you the following program to explain how to use command line arguments. This simple program prints the arguments that were passed to it from the command line.

```
public class TestClass{
    public static void main(String[] args) throws Exception{
        for(int i=0; i<args.length; i++){
            System.out.println("args["+i+"] = \\""+args[i]
]+\\"");
        }
    }
}
```

The output of this program will tell you all you need to know about the command line arguments. The following is a table containing the command line used to execute the program and the corresponding output generated by the program:

Command line used	Output	Inference
<code>java TestClass</code>		If no argument is specified, args contains a String array of length 0. Observe that a <code>NullPointerException</code> is not raised for <code>args.length</code> . That means <code>args</code> is not null . In this case <code>args</code> refers to a <code>String</code> array of length 0.
<code>java TestClass a</code>	<code>args[0] = "a"</code>	The first argument is stored at index 0. The first argument is NOT <code>"java"</code> or even the name of the class, i.e., <code>"TestClass"</code> .

java TestClass a b c	args[0] = "a" args[1] = "b" args[2] = "c"	Arguments can be separated by one or more than one white characters. All such separator characters are stripped away.
java TestClass "a" b	args[0] = "a " args[1] = "b"	If you put quotes around the value that you want to pass, everything inside the quotes is considered one parameter. Quotes are not considered as part of the argument. Observe that the first argument is "a ", i.e., a String containing character 'a' followed by a space character.
java TestClass "\""	args[0] = """"	To pass a quote character as an argument, you have to escape it using a backslash.

By the way, can you guess why the name of class is not passed in as an argument to the main method? Remember that unlike an executable program, you cannot change the name of a Java class file. The name of a Java class file will always be the same as the name given to the class in the Java source file. Therefore, the main method of a class always knows the name of its containing class.

2.1.3 The end of main

As discussed before, once the JVM passes control to the **main method** of the class that you are trying to execute, it is the main method that decides what to do next. As far as the JVM is concerned, your application has been "launched" upon invocation of the main method. In that sense, the main method is just an entry point of your application. So, what happens when the main method ends? Does the application end as well? Well, the answer is it depends on what the main method does.

A simple Java program such as the one we used earlier to print arguments may have all its code in the main method. Once the main method ends, there is nothing else to do and so, the program ends. While you can write all your application code within the main method, Java applications are usually composed of multiple classes. At run time, an application consists of instances of several classes that interact with each other by calling methods on each other. A

Java application may also perform multiple activities in parallel, so, even if an activity implemented by one method ends, another activity implemented by some other method may still be going on. The code in main itself is just an activity. The end of the main method implies the end of only that activity. It doesn't mean the end of all the activities that may be going on in an application.

If it helps, you may think of your Java application as a fast food restaurant and the main method as its manager opening the restaurant in the morning. The restaurant need not close immediately after opening if there are no customers lined up. After opening the restaurant, the manager kicks off a lot of activities such as preparing the food, setting up the dining area, and waiting for customers. Such activities may continue side by side throughout the course of the day. When the last customer of the day is gone and when all such actives end, the restaurant closes for the day. The same thing happens in a Java application. The main method may kick off other activities that run side by side and the application ends only when all such activities come to an end.

Java allows executing activities in parallel using threads. This topic is beyond the scope of this exam so, I will not discuss it anymore in this book. But you should know that in a nutshell, an application doesn't end until all the threads started by the application, including the thread that executes main, end.

2.2 Compile and run a Java program from the command line

2.2.1 Run a Java program from the command line [↳](#)

We have already seen the basics of how to compile and execute a Java program in Compilation and Execution section under Kickstarter for Beginners. I will just summarize the important points that you need to know for the exam here.

1. The standard Oracle JDK comes bundled with a Java compiler. The executable for the compiler is named **javac**. In other words, the program you need to use to compile your Java code is called "javac". You may compile a Java source file named **TestClass.java** using the command

- javac TestClass.java

Notice that you have to specify the full file name including the extension. javac does support multiple options to fine tune the compilation process but none of these options are required for the exam.

2. Compilation of a Java file results in one or more class files depending on the contents of the Java source file.
3. The standard Oracle JDK comes bundled with a Java Virtual Machine (JVM). The executable for the JVM is named java. In other words, the program you need to launch the JVM and to execute your Java program is called "**java**".
4. To execute a Java class, you can use the following command - **java TestClass**

Notice the absence of file extension **.class** while specifying the class name. To compile a Java source file, you must specify the extension of the file, i.e., **.java** though. Just like with javac, Java command can take multiple options to fine tune the execution of a Java program. You do not need to know any of these options for the exam.

5. Java 11 allows what is called "single file source code programs", which means, if your program is present in a single source file, you can run that file directly using the **java** command instead of compiling it first using **javac**. For example, **java TestClass.java**

2.3 Create and import packages

2.3.1 The package statement ↗

Every Java class belongs to some or the other package. The name of this package is specified using the package statement contained in a source file.

There can be at the most one package statement in the entire source file and, if present, it must be the first statement (excluding comments, of course) in the file. All top level types defined in this file belong to this package. If there is no package statement in a Java file, then the classes defined in that file belong to an unnamed package which is also known as the "**default**" package. In other words, if you have two Java files without any package statement, classes defined in those two files belong to the same unnamed package.

Important points about the unnamed package [↳](#)

1. The unnamed package has no name. Duh!
2. Default is not the name of the unnamed package. There is no package named **default**. You cannot even create a package named default by specifying default as the package name for your class though because default is a keyword.
3. Since the unnamed package has no name, it is not possible to refer to this package. In other words, it is not possible to import classes belonging to the unnamed package into classes belonging to another package. You can't do `import *;` in your Java file. This is one reason why it is not recommended to create classes without a package statement.

You can name your package anything but it is recommended that you use the reverse domain name format for package. For example, if you work at **Amazon**, you should start your package name with **com.amazon**. You should then append the group name and application name to your package name so, as to make your class unique across the globe. For example, if the name of your group is sales, and the name of the application is itemMaster, you might name your package **com.amazon.sales.itemMaster**. If the name of your class is **Item**, your **Item.java** source file will look like this:

```
package com.amazon.sales.itemMaster;  
public class Item{  
}
```

Although you can use non-ascii characters in your package name, the exam will not ask you questions about package names with such characters.

2.3.2 Quiz

Q1. Which of the following code snippets are valid?

Select 1 correct option.

A.

//in Test.java

```
package;
public class Test{}
```

B.

//in Test.java

```
package mypackage;
public class Test{}
```

C.

//in Test.java

```
package x;
public class Test{}
package y;
class AnotherTest{}
```

D.

//in Test.java

```
package x;
package y;
public class Test{
}
```

Correct answer is B.

A is incorrect because you must specify the package name along with the keyword package.

C and **D** are incorrect because you cannot have more than one package statement in a Java source file. Moreover, **C** is incorrect also because the package statement must be the first statement in a Java file if it exists in the file.

2.3.3 The import statement

If all of your classes are in the same package, you can just use the simple class name of a class to refer to that class in another class. But to refer to a class in one package from another, you need to use its "**fully qualified class name**" or **FQCN** for short. FQCN of a class is basically the package name + dot + the class name. For example, if the package statement in your class **Test** is `package com.enthuware.ocp;`, the FQCN of this class is `com.enthuware.ocp.Test`.

If you want to refer to this class from another class in a different package, say `com.xyz.abc`, you need to use the FQCN, i.e., `com.enthuware.ocp.Test`. For example,

```
com.enthuware.ocp.Test t = new  
com.enthuware.ocp.Test();
```

If you try to use just the simple class name, i.e., `Test t = new Test();`, the compiler will assume that you mean to use the Test class from the same package, i.e., `com.xyz.abc` and if it doesn't find that class in `com.xyz.abc`

package, it will complain that it doesn't understand what you mean by "Test". FQCN tells the compiler exactly which class you intend to use.

If you refer to this class several times in your code, you can see that it will lead to too many repetitions of `com.enthuware.ocp` in the code. The import statement solves this problem. If you add an import statement `import com.enthuware.ocp.Test;`, you can use just the simple class name `Test` in your class to refer to `com.enthuware.ocp.Test` class.

If your class refers to multiple classes of the same package, you can use either use one import statement for each class or you can use just one import statement with a wild card character * for the whole package. For example, `import com.enthuware.ocp.*;` The compiler will try to find the simple class names used in your code in the imported package(s). You can have as many import statements as you need. You can also have redundant imports or imports that are not needed.

Although importing all the classes with a wildcard looks like good idea but I assure you that it is not. In practice, if a class uses several classes from different packages, it becomes difficult to figure out the package to which a class referred to in the code belongs. For this reason, well written, professional code always uses import statements for specific classes instead of using the wildcard format. Most IDEs even have a feature to clean up import statements of a class. In NetBeans, you can do it with Control+Shift+i, for example.

The `import static` statement

Sometimes you need to define values that are to be used in various classes of your application. For example, if you are developing an application for tax computation, you may want to define a value for tax rate that is to be used by all other classes. Since all code in Java must be a part of a reference type (i.e. a class or an interface or an enum), you may define a class named Values and add this value to this class as follows:

```
package taxes;  
public class Values{  
    public static double TAX_RATE = 0.15;  
}
```

Now, if you want to use this value in some other class, you have three options. You know the first two options already:

Option 1 - Don't import anything and just use `taxes.Values.TAX_RATE`; in your class.

Option 2 - Add `import taxes.Values;` or `import taxes.*;` and then use `Values.TAX_RATE` in your class.

Option 3 - Java 7 onwards, you have a third option called "**import static**". To eliminate typing the class name multiple times, you can simply import the static members of any class using the import static statement. In this example, you can add `import static taxes.Values.TAX_RATE;` or `import static taxes.Values.*;` (the wild card format imports all static members of the class) to the list of import statements of the class and then use just `TAX_RATE` in your code.

You can import static fields as well as static methods through this statement. So, for example, if you have static utility method named `apply` in `Values` class, you could directly use the name `apply` instead of `Values.apply` if you include `import static taxes.Values.apply;` or `import static taxes.values.*;`

Remember that `import static` does not import a class. It imports only the static member(s) of a class. Thus, you cannot use the simple name `Values` in your code if you haven't imported `taxes.*` or `taxes.Values` already using the regular import statement.

The word "import" is really a misnomer here. The import statement doesn't import anything into your class. It is merely a hint to the compiler to look for classes in the imported package. You are basically telling the

compiler that the simple class names referred in this code are actually referring to classes in the packages mentioned in the import statements. If the compiler is unable to resolve a simple class name (because that class is not in the same package as this class), it will check the import statements and see if the packages mentioned there contain that class. If yes, then that class will be used, if not, a compilation error will be generated.

Important points about the import statement -

1. You can import each class individually using `import packagename.classname;` statement or all the classes of a package using `import packagename.*;` or any combination thereof.
2. import statements are optional. You can refer to a class from another package even without using import statements. You will have to write FQCN of the class in your code in that case.
3. You can import any number of packages or classes. Duplicate import statements and redundant import statements are allowed. You can import a class even if you are not using that class in your code. Remember, an import statement is just a shortcut for humans. It doesn't actually import anything in your class.
4. `java.lang` package is imported automatically in all the classes. You don't need to write `import java.lang.*;` in your class even if you use classes from `java.lang` package. But it is not wrong to import it anyway because redundant imports are allowed.

What you cannot do:

1. There is no way to import a "subpackage" using the import statement. For example, `import com.enthuware.*;` will import all the class in package `com.enthuware` but it will not import any class under `com.enthuware.ocp` package. Furthermore, `import`

`com.enthware.*.*`; is illegal. This essentially means that technically, there is no concept of "subpackage" in Java. Each package must be imported separately.

2. You cannot import a package or a class that doesn't exist. For example, if you try to use some random package name such as `import xyz.*`; the compiler will raise an error saying,

```
error: package xyz does not exist
import xyz.*;
^
1 error
```

How does the compiler know whether a package exists or not, you ask? Well, if the compiler doesn't find any class in its classpath that belongs to the package that you want to import, it draws the inference that such a package does not exist.

3. Unpackaged classes (the phrases "unpackaged classes" and "classes in the default or unnamed package" mean the same thing, i.e., classes that do not have any package statement) cannot be imported in any other package. You cannot do something like `import *`;
4. If a class by the same name exists in multiple packages and if you import both the packages in your code, you cannot use the simple class name in your code because using the simple name will be ambiguous. The compiler cannot figure out which class you really mean. Therefore, you have to use FQCN in such a case. You may import one package or class using the import statement and use simple name for a class in that package and use FQCN for classes in the other package.

The requirement to use two classes with same name but from different packages typically used to arise a lot while using JDBC. JDBC related classes are in `java.sql` package and classes in this package use `java.sql.Date` class instead of `java.util.Date`. But

the non-JDBC related code of the application uses `java.util.Date` . In such a situation, it is preferable to use FQCN of each class in the code to avoid any confusion to the reader even though you can import one package and use simple name `Date` to refer to the class of that package.

However, Java 8 onward, you should use the new Date/Time classes of the `java.time` package anyway, which eliminates this annoyance.

2.3.4 Quiz

Q. You have downloaded two Java libraries. Their package names are `com.xyz.util` and `com.abc.util` . Both the packages have a class named `Calculator` and both the classes have a static method named `calculate()` .

You are developing your class named `MyClass` in `com.mycompany.app` package and your class code needs to invoke calculate methods belonging to both of the Calculator classes as follows:

```
public class MyClass{
    public static void main(String[] args){
        //call xyz's calculate()

        //call abc's calculate()

    }
}
```

Which of the following approaches will work?

- A. Add `import com.*;` to your class. Then use `xyz.util.Calculator.calculate();` and `abc.util.Calculator.calculate();`
- B. Add `import com.xyz.util.Calculator;` Then use `Calculator.calculate();` and `com.abc.util.Calculator.calculate();`
- C. Do not use any import statement. In the code, use `com.xyz.util.Calculator.calculate();` and `com.abc.util.Calculator.calculate();`
- D. This cannot be done.

Correct answer is B and C.

Option A is incorrect because you cannot import partial package names. While using a class, you can either use simple class name (if you have imported the class or package using the import statement) or use Fully Qualified Class Name. You cannot use partial package name to refer to a class.

2.4 Exercises

1. Create classes in two different named packages. Define static and instance fields in one of those classes and access those fields from the other class. See what happens when both the classes try to access the fields of each other.
Hint: If you have trouble compiling classes, check out "Compilation and Execution" section in Kickstarter for Beginners".
2. Define a local variable in a method. Update this variable in a while loop and print it out after the while loop ends. Check what happens when you define a variable by the same name within the while loop.

3. Create a class in package foo and another class in package foo.bar with a static method. Invoke the static method from the class in package foo using different import statements.
4. Create a class with a main method and execute the class with a few arguments. Print the number of arguments.
5. Which Java feature (or lack of thereof) annoys you most. Why?

Chapter 3 Working With Java Primitive Data Types

Exam Objectives

- Declare and initialize variables (including casting and promoting primitive data types)
- Identify the scope of variables (*moved to the next chapter)
- Use local variable type inference (*moved to the next chapter)

3.1 Data types in Java

Java has support for two kinds of Data.

3.1.1 Data types and Variable types

A **data type** is essentially a name given to a certain kind of data. For example, integer data is given the name "int" in Java. Boolean data is given the name "boolean" in Java. Classifying data into different data types allows you to treat data of the same kind in the same way. It also allows you to define a set of operations that can be performed on data of the same kind. For example, if you are given data of type int and of type boolean, you know that you can do addition operation on the int data but not on the boolean data. Data type also determines the space required to store that kind of data. For example, a byte requires only 8 bits to store while an int requires 32 bits.

Data types are important for a programming language because they allow you tell the compiler the kind of data you want to work with. For example, when you say `int i;` you are telling the compiler that `i` is of type `int`. The compiler will then allow you to store only an integer value in this variable. Ignoring a few

exceptions, it is not possible to store data of one type into a variable of another type because of the difference in the amount of space required by different data types and/or because of their compatibility.

Languages where the data type of a variable is defined at compile time and cannot change during run time are called "statically typed" languages. Java is, therefore, a statically typed language. Languages that allow the data type of a variable to change at run time are called dynamically typed languages. JavaScript is an example of a dynamically typed language. A variable in JavaScript code may contain an integer value in one statement and may point to a String at the next statement. There are advantages and disadvantages to each approach but discussing that would be way out of scope. You should, however, read about it online for interview purposes.

By defining a variable of a certain type, you automatically get the right to perform operations that are valid for that type on that variable. For example, if **i** is defined to be of type the **int** , the compiler will allow you to perform only mathematical and bit wise operations on this variable. If **b** is defined as a **boolean** , the compiler will allow you to perform only logical operations on this variable.

Java has two fundamental kinds of data types: **primitive** and **reference** .

Primitive data types are designed to be used when you are working with raw data such integers, floating point numbers, characters, and booleans. Java (by Java, I mean, the Java compiler and the Java Virtual Machine) inherently knows what these data types mean, how much space they take up, and what can be done with them. You don't need to explain anything about them to Java. Primitive data types are the most basic building blocks of a Java program. You combine primitive data types together in a class to build more complicated data types.

Reference data types , on the other hand, are designed to be used when you are working with data that has a special meaning for code that Java has no knowledge of. For example, if you are developing an application for student

management, you might define entities such as Student, Course, and Grade. Java has no knowledge of what Student, Course, and Grade mean. It doesn't know how much space they take, what operations they support, or what properties they have. Java will expect you to define all these things. Once you define them, you can use them to implement the business logic of your application. When you write a class, interface, or enum, you are essentially defining a reference data type. Reference data types are built by combining primitive data types and other reference data types.

In Java, primitive data types include **integral data types** (`byte`, `char`, `short`, `int`, `long`), **floating point data types** (`float`, `double`), and **boolean data type** (there is only one - `boolean`). While reference data types include all the **classes**, **interfaces**, and, **enums**, irrespective of who defines them. If something is a class, an interface, or an enum, it is a reference data type. Yes, `String` too is a reference data type because all strings are instances of type `java.lang.String` class :) I will talk more about Strings later.

Note that **integral** and **floating point** data types are collectively called **numeric data types**.

The following table lists out the details of primitive data types:

You will not be asked the details of the sizes of data types in the exam. However, it is important to know about them as a Java programmer.

Primitive data Types in Java

Data Type	Size (in bits)	Range of values	Sample Values	Operations supported
byte	8	-2 ⁷ to 2 ⁷ -1, i.e., -128 to 127	-1, 0, 1	mathematical, bitwise
char	16	0 to 2 ¹⁶ -1, i.e., 0 to 65,535	0, 1, 2, 'a', '\u00061'	mathematical, bitwise
		-2 ¹⁵ to 2 ¹⁵ -1, i.e., -32,768 to		Mathematical,

short	16	32,767	-1, 2, 3	bitwise
int	32	-2 ³¹ to 2 ³¹ -1	-1, 2, 3	mathematical, bitwise
long	64	-2 ⁶³ to 2 ⁶³ -1	-1, 2, 3	mathematical, bitwise
float	32	approximately ±3.40282347E+38F	1.1f, 2.0f	mathematical
double	64	approximately ±1.79769313486231570E+308	1.1, 2.0	mathematical
boolean	1	true or false	true, false	logical

Notes:

1. **byte, char, short, int, and, long** are called **integral data types** because they all store precise integral values.
2. **char** is also an integral type that stores numbers just like byte, short, int and long. But it cannot store a negative number. The number stored in a char variable is interpreted as a **unicode character**.
3. **float** and **double** store large but imprecise values. Java follows IEEE 754 standard. You may go through it to learn more but it is not required for the exam.
4. A **boolean** stores only two values and therefore, requires only one bit of memory. Officially however, its size is not defined because the size depends on the smallest chunk of memory that can be addressed by the operating system. On 32 bit systems, a **boolean** may even use 4 bytes.

A word on void

void is a keyword in Java and it means "nothing". It is used as a return type of a method to signify that the method never returns anything. In that sense, void is a type specification and not a data type in itself. That is why, even though you can declare a method as

returning `void` but you cannot declare a variable of type `void` .

Difference between `null` and `void`

`null` is also a keyword in Java and means "nothing". However, `null` is a value. It is used to signify that a reference variable is currently not pointing to any object. It is not a data type and so, it is not possible to declare a variable of type `null` .

Note that `null` is a valid value for a reference variable while `void` is not. When a method invocation returns `null` , it means that only that particular invocation of the method did not return a valid reference. It does not mean that the method never returns a valid reference. On the other hand, `void` means that a method never returns anything at all. Therefore, you cannot use `void` in a return statement. In other words, `return null;` can be a valid return statement for a method but `return void;` is never valid. A method that declares `void` as its return type, can either omit the return statement altogether from its body or have an empty return statement, i.e., `return;` .

Types of variables

Java has two types of variables to work with the two types of data types, namely primitive variables and reference variables. Primitive variables let you work with primitive data, while reference variables let you work with reference data. Thus, when you define `int i;` `i` is a variable of the primitive data type `int` , but when you define, `String str;` `str` is a variable of the reference data type `java.lang.String` .

It is very important to understand the fundamental difference between the two types of variables. A primitive variable contains primitive data within itself, while a reference variable stores only the address to the location where the actual data is stored. For example, if you do `i = 10;`, `i` will contain the value 10. But if you do `str = "hello";`, `str` will only contain the address of the memory location where the string "hello" resides. You can now understand why they are called "reference" variables. Because they are merely references to the actual data! When you perform any operation on a reference, the operation is actually performed on the object that is located somewhere else. In that sense, you can think of a reference variable as a "remote control" of a TV. (If you have trouble understanding this, you should go through the "Kicker for Beginners" chapter before moving forward.)

Both types of variables support the assignment operation, i.e., they allow you to assign values to them. For example, the statement `i = 20;` assigns the value 20 to the variable `i`.

In case of a reference variable, you cannot assign the address of an object directly. You can only do so indirectly. For example, in statement the `String str = "hello";` you are assigning the address of the memory location at which the string "hello" is stored to `str` variable. `str`, therefore, now contains the address of a memory location. Similarly, in statement `String str2 = str;` you are assigning the value stored in `str` to `str2`. You are not copying "hello" to `str2`. You are just copying the address stored in `str` to `str2`. You cannot assign a memory address to a reference variable directly because you don't know the actual address. Only the JVM knows where an object is stored in memory and it assigns that address to the variable while executing the assignment operation. The only "address" you can assign to a reference variable directly is `null`.

Size of variables ↗

Since a primitive variable stores actual data within itself, the size of a primitive variable depends on the size of the primitive data. Thus, a byte variable requires 1 byte while an int variable requires 4 bytes and so on.

Since a reference variable stores only the address of a memory location, the size of a reference variable depends on the addressing mechanism of the machine. On

a system with 32 bit OS, a reference variable will be of 4 bytes, while on a 64 bit systems, it will be of 8 bytes.

Size of reference data types

Size of a reference data type such as a class can be easily determined at compile time by looking at the instance variables defined in that class. Since every instance variable will either be a primitive variable or a reference variable, and since you know the sizes of each of those types, the size of an instance of that class will simply be the sum of the sizes of its instance variables.

This size never changes for a given class. All instances of a given class always take exactly the same amount of space in memory, no matter what values its internal variables hold.

Thus, there is never a need to calculate the size of memory space taken by an instance of a class at run time. And for this reason, there is no such operator as "sizeof" in Java.

3.2 Difference between reference variables and primitive variables

3.2.1 Reference variables and primitive variables

In the "Object and Reference" lesson, we discussed the relationship between a class, an object, and a reference. I explained the fundamental difference between an object reference and a primitive. To recap, there is no difference between an object reference and a primitive variable from the program memory perspective. In memory, both just store a raw number. The difference is in how the JVM interprets that raw number. In the case of a reference variable, the JVM interprets the number as an address of another memory location where the actual object is stored, but in the case of a primitive variable, it interprets the raw

number as a primitive data type (i.e. a byte, char, short, int, long, float, double, or boolean). In that sense, primitives do not have references. There is nothing like a primitive "reference" because there is no object associated with a primitive variable.

Another crucial point to understand here is that it is the objects that support methods and have fields, not the references. Therefore, when you invoke a method (or access a field) using a reference, the JVM invokes that method on the actual object referred to by that variable and not on the variable itself.

Since primitives are not objects, you cannot "invoke" any method on a primitive variable. But you can perform mathematical (+, -, *, /, and, %), logical (||, &&, !, |, and, &), and bitwise(â^¼, |, and, &) operations on the primitive variables themselves.

The following image explains the above with some code. The code assumes that there is a class named Student defined as follows:

```
public class Student{  
    int id;  
}
```

```
int i1 = 1234;  
int i2 = i1;
```

Contents of i1 are copied into i2.
Therefore, i2 now contains 1234 as well.

```
i2 = 0;
```

i2 is set to 0. This doesn't affect i1,
which still contains 1234.

```
System.out.println(i1);
```

prints the contents of i1 i.e. 1234

...

```
Student s1 = new Student();
```

The new keyword creates a new Student object and places it on the heap.
The assignment operator assigns the address of the location of the Student object to the variable s1. If the address is, for example, 1001, s1 now contains 1001.

```
s1.id = 1;
```

JVM knows that s1 is a reference type (and not a primitive type), therefore, it goes to the object stored at the address contained in s1 and sets that object's id field to 1.

```
Student s2 = s1;
```

Contents of s1, i.e. 1001, are copied into s2.
Therefore, s2 now contains 1001 as well.

```
s2.id = 2;
```

JVM knows that s2 is a reference type and so it goes to the object stored at the address contained in s2 and sets that object's id field to 2.

```
System.out.println(s1.id);
```

Prints 2 because the previous statement modified the same object.

As you can observe in the above flow diagram, whenever you assign one variable to another, the JVM just copies the value contained in the variable on the right-hand side of the assignment operator to the variable on the left-hand side. It does this irrespective of whether the variable is a primitive variable or a reference variable. In case of a primitive variable, the value happens to be the actual primitive value and in case of a reference variable, the value happens to be the address of an object. In both the cases, it is the value that is copied from one variable to another. For this reason, it is also said that Java uses "**pass by value**" semantics instead of "**pass by reference**". We will revisit this later when we discuss about passing variables to method calls.

This concept is very important and you will see many questions that require you to have a clear understanding of it. The only thing that you need to remember is that a variable, be it of any kind, contains just a simple raw number. Assigning one variable to another simply copies that number from one variable to another.

It is the JVM's job to interpret what that number means based on the type of the variable. Everything else just follows from this fundamental rule.

3.3 Declare and initialize variables

3.3.1 Declare and initialize variables [↳](#)

For better or for worse, Java has several ways of declaring and initializing variables. The exam expects that you know them all. Although Oracle have substantially reduced the number of questions that are based solely on quirky syntax, you may still see weird syntax used in code snippets in questions that test you on something else.

So, let's go through them one by one starting with the most basic - declarations without initialization.

1. `int x;
String str;
Object obj;`
2. `int a, b, c; //a, b, and c are declared to be of type int`

`String s1, s2; //s1 and s2 are declared to be type String`

The following are ways to declare as well as initialize at the same time:

1. `int x = 10; //initializing x using an int literal 10`
2. `int y = x; //initializing y by assigning the value of another variable x`
3. `String str = "123"; //initializing str by creating a new String`

4. `SomeClass obj = new SomeClass();` //initializing *obj* by creating a new instance of *SomeClass*
5. `Object obj2 = obj;` //initializing *obj2* using another reference
6. `int a = 10, b = 20, c = 30;` //initializing each variable of same type with a different value
7. `String s1 = "123", s2 = "hello";`
8. `int m = 20; int p = m = 10;` //resetting *m* to 10 and using the new value of *m* to initialize *p*

Mixing the two styles mentioned above:

1. `int a, b = 10, c = 20;` //*a* is declared but not initialized. *b* and *c* are being declared as well as initialized
2. `String s1 = "123", s2;` //Only *s1* is being initialized

And the following are some illegal ones:

1. `int a = 10, int b;` //You can have only one type name in one statement.
2. `int a, Object b;` //You can have only one type name in one statement.
3. `int x = y = 10;` //Invalid, *y* must be defined before using it to initialize *x*.

Observe that there is no difference in the way you declare a primitive variables and a reference variables. A reference variable, however, has one additional way of initialization - you can assign null to a reference variable. You can't do that to a primitive variable. For example, `int i = null;` is invalid. But `String s1 = null;` is valid.

Naming rules for a variable

A variable name must be a valid Java identifier. Conventionally however, a variable name starts with a lower case letter and names for constant variables are in upper case. Variables created by code generation tools usually start with an

underscore or a dollar (_ or \$) sign.

3.3.2 Uninitialized variables and Default values [↳](#)

Given just this statement - `int i;` - what will be the value of `i`?

If you are from C/C++ world, you may say that the value is indeterminate, i.e., `i` may have any value. Java designers didn't like this undefined behavior of uninitialized variables because it is a common source of bugs in applications. A programmer may forget to initialize a variable and that may cause unintended behavior in the application. Uninitialized variables don't serve any purpose either. To use a variable, you have to assign it a value anyway. Then why leave them uninitialized?

For this reason, Java designers simply outlawed the use of uninitialized variables altogether in Java. In fact, they went even further and made sure that if a programmer doesn't initialize a variable, the JVM initializes them with known pre-determined values. Well, in most cases!

Try compiling the following code:

```
public class TestClass{  
    static int i;  
    int y;  
    public static void main(String[] name){  
        int p;  
    }  
}
```

It compiles fine without any issues. It will run fine as well but will not produce any output. Now, try the same code with a print statement that prints `i` and `y`.

```
public class TestClass{  
    static int i;  
    int y;  
    public static void main(String[] name){  
        int p;
```

```
        System.out.println(i+" "+new TestClass().y);
    }
}
```

This also compiles fine. Upon running, it will print **0 0**. Now, try the following code that tries to print **p**.

```
public class TestClass{
    static int i;
    int y;
    public static void main(String[] name){
        int p;
        System.out.println(p);
    }
}
```

This doesn't compile. You will get an error message saying:

```
TestClass.java:6: error: variable p might not have been initialized
        System.out.println(p);
```

You can draw the following conclusions from this exercise:

1. Java doesn't have a problem if you have uninitialized variables as long as you don't try to use them. That is why the first code compiles even though the variables have not been initialized.
2. Java initializes static and instance variables to default values if you don't initialize them explicitly. That is why the second code prints **0 0**.
3. Java doesn't initialize local variables if you don't initialize them explicitly and it will not let the code to compile if you try to use such a variable. That is why the third code doesn't compile.

The first point is straightforward. If a variable is not used anywhere, you don't have to initialize it. It is possible that a smart optimizing Java compiler may even eliminate such a variable from the resulting class file.

Let us look at the second and third points now. To make sure that variables are always initialized to specific predetermined values before they are accessed,

Java takes two different approaches .

The **first approach** is to let the JVM initialize the variables to predetermined values on its own if the programmer doesn't give them any value explicitly. This approach is taken for **instance** and **static** variables of a class. In this approach, the JVM assigns **0** (or **0.0**) to all numeric variables (i.e. byte, char, short, int, long, float, and double), **false** to boolean variables, and **null** to reference variables. These values are called the default values of variables. The following code, therefore, prints **0**, **0.0**, **false**, and **null**.

```
public class TestClass{
    static int i; //i is of numeric type and is therefore, initialized to 0

    static double d; //d is a floating numeric type and is therefore, initialized to 0.0

    static boolean f; //f is of boolean type and is therefore, initialized to false

    static String s; //s is of reference type and is therefore, initialized to null

    public static void main(String[] args){
        System.out.println(i);
        System.out.println(d);
        System.out.println(f);
        System.out.println(s);
    }
}
```

Observe that since **s** is a reference variable, it is initialized to **null**. You will learn in the next chapter that an array is also an object, which means that an array variable, irrespective of whether it refers to an array of primitives or objects, is a reference variable, and is, therefore, treated the same way.

The above code uses only static variables. You will get the same result with instance variables:

```
public class TestClass{  
    int i;  
    double d;  
    boolean f;  
    String s;  
    public static void main(String[] args){  
        TestClass tc = new TestClass();  
        System.out.println(tc.i);  
        System.out.println(tc.d);  
        System.out.println(tc.f);  
        System.out.println(tc.s);  
    }  
}
```

The **second approach** is to make the programmer explicitly initialize a variable before the variable is accessed. In this approach, the compiler raises an error if it finds that a variable may be accessed without being initialized. This approach is used for local variables (i.e. variables defined in a method or a block).

Basically, the compiler acts as a cop that prevents you from using an uninitialized variable. If at any point the compiler realizes that a variable may not have been initialized before it is accessed, the compiler flags an error. This is called the principle of "**definite assignment**". It means that a local variable must have a definitely assigned value when any access of its value occurs. For example, the following code compiles fine because even though the variable **val** is not initialized in the same line in which it is declared, it is definitely assigned a value before it is accessed:

```
public class TestClass {  
    public static void main(String[] args) throws Exception {  
        int val; //val not initialized here
```

```

    val = 10;
    System.out.println(val); //compiles fine

}

}

```

A compiler must perform flow analysis of the code to determine whether an execution path exists in which a local variable is accessed without being initialized. If such a path exists, it must refuse to compile the code. A compiler is only allowed to consider the values of "**constant expressions**" in its flow analysis. The Java language specification does formally define the phrase "constant expression" but I will not go into it here because it is outside the scope of the exam. The basic idea is that a compiler cannot execute code and so, it cannot make any inferences based on the result of execution of the code. It has to draw inferences based only on the information that is available at compile time. It can take into account the value of a variable only if the variable is a **compile time constant**. This is illustrated by the following code:

```

public class TestClass {
    public static void main(String[] args) throws Exception {
        int val;
        int i = 0; //LINE 4

        if(i == 0){
            val = 10;
        }
        System.out.println(val); //val may not be initialized

    }
}

```

This code will not compile. Even though we know that `i` is 0 and so, `i == 0` will always be true, the compiler doesn't know what the actual value of the variable `i` will be at the time of execution because `i` is not a compile time

constant. Therefore, the compiler concludes that if the `if` condition evaluates to `false`, the variable `val` will be left uninitialized. In other words, the compiler notices one execution path in which the variable `val` will remain uninitialized before it is accessed. That is why it refuses to accept the print statement. If you change line 4 to `final int i = 0;` the compiler can take the value of `i` into account in its flow analysis because `i` will now be a compile time constant. The compiler can then draw the conclusion that `i==0` will always be true, that the `if` block will always be executed, and that `val` will definitely be assigned a value before it is accessed.

Similarly, what if we add the `else` clause to the `if` statement as shown below?

```
int val;
int i = 0; //i is not final

if(i == 0){
    val = 10;
} else{
    val = 20;
}
System.out.println(val);
```

Now, `i` is still not a compile time constant but the compiler doesn't have to know the value of `i`. If-else is one statement and the compiler is now sure that no matter what the value of `i` is, `val` will definitely be assigned a value. Therefore, it accepts the print statement.

Let us now change our if condition a bit.

```
if(i == 0){
    val = 10;
}
if(i != 0){
    val = 20;
}
```

It doesn't compile. It has the exact same problem that we saw in the first version. We, by looking at the code, know that `val` will definitely be initialized in this

case. We know this only because we executed the code mentally. As far as the compiler is concerned, these are two independent **if** statements. Since the compiler cannot make inferences based on the results of execution of expressions that are not compile time constants, it cannot accept the argument that **val** will definitely be assigned a value before it is accessed in the print statement.

In conclusion, Java initializes all static and instance variables of a class automatically if you don't initialize them explicitly. You must initialize local variables explicitly before they are used.

3.3.3 Assigning values to variables [↳](#)

Java, like all languages, has its own rules regarding assigning values to variables. The most basic way to assign a value to a variable is to use a "literal".

Literals [↳](#)

A literal is a notation for representing a fixed value in source code. For example, **10** will always mean the number **10**. You cannot change its meaning or what it represents to something else in Java. It has to be taken literally, and hence it is called a literal. Since it represents a number, it is a numeric literal. Similarly, **true** and **false** are literals that represent the two boolean values. '**a**' is character literal. "**hello**" is a string literal. The words **String** and **name** in the statement **String name;** are not literals because Java does not have an inherent understanding of these words. They are defined by a programmer.

*The word **int** in **int i;** or the word **for** in **for(int i=0; i<5; i++);** are kinda similar to literals because they have a fixed meaning that is defined by the Java language itself and not by a programmer. They are actually a bit more than literals because they tell the compiler to treat the following code in a particular way. They form the instruction set for the Java compiler using which you write a Java program*

and are therefore, called "keywords".

Let me list a few important rules about literals:

1. To make it easy to read and comprehend large numbers, Java allows underscores in numeric literals. For example, **1000000.0** can also be written as **1_000_000.0**. You cannot start or end a literal with an underscore. You can use multiple underscores consecutively. You need not worry about the rules governing the usage of underscores in hexadecimal, binary, and octal number formats.
2. A number without a decimal is considered an **int literal**, whereas a number containing a decimal point is considered a **double literal**.
3. A **long literal** can be written by appending a lowercase or uppercase **L** to the number and a **float literal** can be written by appending a lowercase or uppercase **f**. For example, **1234L** or **1234.0f**.
4. A **char** literal can be written by enclosing a single character within single quotes, for example, '**a**' or '**A**'. Since it may not always be possible to type the character you want, Java allows you to write a **char** literal using the hexadecimal Unicode character format ('**\uxxxx**'), where **xxxx** is the hexadecimal value of the character as defined in unicode charset. For some special characters, you can also use escape character ****. For example, a new line character can be written as '**\n**'.
Note that writing character literals using a unicode or escape sequence is not on the exam. I have presented this brief information only for the sake of completeness.
5. There are only two boolean literals: **true** and **false**.
6. **null** is also a literal. It is used to set a reference variable to point to nothing.
7. Java allows numeric values to be written in hexadecimal, octal, as well as binary formats. In hexadecimal format (aka hex notation), the value must start with a **0x** or **0X** and must follow with one or more hexadecimal digits. For example, you could write **0xF** instead of **15**. In octal format,

the number must start with a 0 and must follow with one or more octal digits. For example, `017` is `15` in octal. In binary format, the number must start with a `0b` or `0B` and must follow with one of more binary digits (i.e. zeros and ones). Understanding of these formats is not required for the exam and so, I will not discuss these formats any further.

Assignment using another variable [↳](#)

The second way to assign a value to a variable is to copy it from another variable. For example, `int i = j;` or `String zipCode = zip;` or `Student topper = myStudent;` are all examples of copying the value that is contained in one variable to another. This works the same way for primitive as well as reference variables. Recall from the "Object and Reference" section that a reference variable simply contains a memory address and not the object itself. Thus, when you assign one reference variable to another, you are only copying the memory address stored in one variable to another. You are not making a copy of the actual object referred to by the variable.

Assignment using return value of a method [↳](#)

The third way to assign a value to a variable is to use the return value of a method. For example, `Student topper = findTopper();` or `int score = evaluate();` and so on.

Assigning value of one type to a variable of another type [↳](#)

In all of the cases listed above, I showed you how to assign a value of one type to a variable of the same type, i.e., an `int` value to an `int` variable or a `Student` object to a `Student` variable. But it is possible to assign a value of one type to a variable of another as well. This topic is too broad to be covered fully in this chapter because the rules of such assignments touch upon multiple concepts. I will cover them as and when appropriate. Let me list them here first:

1. Simple assignments involving primitive types - This includes the assignment of compile time constants and the concept of casting for

primitive variables. I will discuss this topic next.

2. Primitive assignments involving mathematical/arithmetic operators - This includes values generated using binary operators as well as compound operators, and the concept of implicit widening and narrowing of primitives. I will discuss this topic in the "Using Operators" chapter.
3. Assignments involving reference types - This expands the scope of casting to reference types. I will discuss this in the "Reusing Implementations Through Inheritance" chapter.

Primitive assignment [↳](#)

If the type of the value can fit into the type of the variable, then no special treatment is required. For example, you know that the size of a **byte** (8 bits) is smaller than the size of an **int** (32 bits) and a **byte** can therefore, fit easily into an **int**. Thus, you can simply assign a **byte** value to an **int** variable. Here are a few similar examples:

```
byte b = 10; //b is 8 bits
```

```
char c = 'x'; //c is 16 bits
```

```
short s = 300; //c is 16 bits
```

```
int i; //i is 32 bits
```

```
long l; //l is 64 bits
```

```
float f; //f is 32 bits
```

```
double d; //d is 64 bits  
  
//no special care is needed for any of the assignments  
//below  
  
i = b;  
i = s;  
l = i;  
f = i;  
d = f;  
//observe that the type of the target variable is larger  
//than the type of the source variable in all of the  
//assignments above.
```

Assigning a smaller type to a larger type is known as "**widening conversion**" . Since there is no cast required for such an assignment, it can also be called "**implicit widening conversion**" . It is analogous to transferring water from one bucket to another. If your source bucket is smaller in size than the target bucket, then you can always transfer all the water from the smaller bucket to the larger bucket without any spillage.

What if the source type is larger than the target type? Picture the bucket analogy again, what will happen if you transfer all the water from the larger bucket to the smaller one? Simple! There may be spillage :) Similarly, when you assign a value of a larger type to a variable of a smaller type, there may be a loss of information. The Java compiler does not like that. Therefore, in general, it does not allow you to assign a value of a type that is larger than the type of the target variable. Thus, the following lines will cause a compilation error:

//assuming variable declarations specified above

```
c = i;  
i = l;  
b = i;  
f = d;
```

//observe that the type of the target variable on the left is smaller than the type of the source variable(on the right) in all of the assignments above.

But what if the larger bucket is not really full? What if the larger bucket has only as much water as can be held in the smaller bucket? There will be no spillage in this case. It follows then that the compiler should allow you to assign a variable of larger type to the variable of a smaller type if the actual value held by the source value can fit into the target value. It does, but with a condition.

The problem here is that the compiler does not execute any code and therefore, it cannot determine the actual value held by the source variable unless that variable is a compile time constant. For example, recall that the number **10** is actually an **int literal**. It is not a **byte** but an **int**. Thus, even though an **int** is larger than a byte, **byte b = 10;** will compile fine because the value 10 can fit into a **byte**. But **byte b = 128;** will not compile because a **byte** can only store values from -128 to 127. 128 is too large to be held by a **byte**.

Similarly, **final int i = 10; byte b = i;** will also compile fine because **i** is now a compile time constant. Being a compile time constant, its value is known to the compiler and since that value is small enough to fit into a byte, the compiler approves the assignment.

Thus, you can assign a source variable that is a compile time constant to a target variable of different type if the value held by source variable fits into the target variable. This is called "**implicit narrowing**". The compiler automatically narrows the value down to a smaller type if it sees that the value can fit into the smaller type. The compiler does this only for assignments and not for method calls. For example, if you have a method that takes a **short** and if you try to pass an **int** to this method, then the method call will not compile even if the value being passed is small enough to fit into a short.

What if the source variable is not a constant? Since the compiler cannot determine the value held by the variable at run time, it forces the programmer to make a promise that the actual value held by the source variable at run time will fit into the target variable. This promise is in the form of a "**cast**". Java allows you to cast the value of one primitive type to another primitive type by

specifying the target type within brackets. For example, `int i = (int) 11.1;` Here, I am casting the floating point value `1.1` to an `int`. You can use a cast to assign any primitive integral (i.e. `byte`, `char`, `short`, `int`, `long`) or floating point type (i.e. `float` and `double`) value to any integral or floating point variable. You cannot cast a `boolean` value to any other type or vice versa.

Here are a few more examples of assignments that can be done successfully with casting:

```
int i = 10;  
char c = (char) i; //explicitly casting i to char
```

```
long l = 100;  
i = (int) l; //explicitly casting l to int
```

```
byte b = (byte) i; //explicitly casting i to byte
```

```
double d = 10.0;  
float f = (float) d; //explicitly casting d to float
```

A cast tells the compiler to just assign the value and to not worry about any spillage. This is also known as "**explicit narrowing**".

But what will happen if there is spillage?, i.e., what will happen if the actual value held by the source variable is indeed larger than the size of the target variable? What will happen to the extra value that can't fit into the target? For example, what will happen in this case - `int i = 128; byte b = (byte) i;`? The explicit cast should simply assign the value that can fit into the target variable and throw away the extra. Thus, it should just assign `127` to `b` and ignore the rest, right? Wrong! If you print the value of `b`, you will see `-128` instead of `127`. There doesn't seem to be any relation between `127` and `-128`! Understanding why this happens is not required for the exam. You will not be asked about the values assigned to variables in such cases. But I will discuss it

briefly because it is useful to know.

Casting of primitives is pretty much like shoving an object of one shape into a mould of another shape. It may cause some parts of the original shape to be cut off. To understand this, you need to look at the bit patterns of `int i` and `byte b`. The size of `i` is 32 bits and the value that it holds is 128, therefore, its bit pattern is: `00000000 00000000 00000000 10000000`. Since you are now shoving it into a `byte`, which is of only 8 bits, the JVM will simply cut out the extra higher order bits that can't fit into a `byte` and assign the lowest order 8 bits, i.e., `10000000` to `b`. Thus, `b`'s bit pattern is `10000000`. Since `byte` is a signed integer, the topmost bit is the sign bit (1 means, it is a negative number). Since negative numbers are stored in **two's complement** form, this number is actually `-128` (and not `-0`!). This process happens in all the cases where the target is smaller than the source or has a different range than the source. As you can see, determining the value that will actually be assigned to the target variable is not a simple task for a human. It is, in fact, a common source of bugs. This is exactly why Java doesn't allow you to assign just about any value to any variable very easily. By making you explicitly cast the source value to the target type, it tries to bring to your attention the potential problems that it might create in your business logic. You should, therefore, be very careful with casting.

Assigning `short` or `byte` to `char` ↗

As you know, the sizes of `short` and `char` are same, i.e., 16 bits. The size of `int` and `float` are also the same, i.e., 32 bits. Thus, it should be possible to assign a `short` to a `char` and a `float` to an `int` without any problem. However, remember that a `char` is unsigned while a `short` is not. So, even though their sizes are the same, their ranges are different. A `char` can store values from `0` to `65535`, while a `short` can store values from `-32768` to `32767`. Thus, it is possible to lose information while making such assignments. Similarly, you cannot assign a `byte` to a `char` either because even though `byte` (8 bits) is a smaller type than `char`, `char` cannot hold negative values while `byte` can.

Here are a few examples that make this clear:

```
char c1 = '\u0061'; //ok, unicode for 'a'
```

`short s1 = '\u0061'; //ok, no cast needed because '\u0061' is a compile time constant that can fit into a short.`

`short s2 = c1; //will not compile - c1 is not a compile time constant, explicit cast is required.`

`char c2 = '\uFEF0'; //ok, unicode for some character.`

`short s2 = '\uFEF0'; //will not compile, value is beyond the range of short.`

`short s3 = (short) '\uFEF0'; //ok because explicit cast is present.`

`char c3 = 1; //ok, even though 1 is an int but it is a compile time constant whose value can fit into a char`

`char c4 = -1; //will not compile because -1 cannot fit into a char`

`short s4 = -1;
char c5 = (char) s4; //ok because explicit cast is present`

Assigning **float** to **int** or **double** to **long** and vice-versa [↗](#)

The same thing happens in the case of `int` and `float` and `long` and `double`. Even though they are of same sizes their ranges are different. `int` and `long` store precise integral values while `float` and `double` don't. Therefore, Java requires an explicit cast when you assign a `float` to a `int` or a `double` to a `long`.

The reverse, however, is a different story. Although `float` and `double` also do lose information when you assign an `int` or a `long` to them respectively, Java allows such assignments without a cast nonetheless. In other words, Java allows implicit widening of `int` and `long` to `float` and `double` respectively.

Here are a few examples that make this clear:

```
int i = 2147483647; //Integer.MAX_VALUE
```

```
float f = i; //loses precision but ok, implicit widening of int to float is allowed
```

```
long g = 9223372036854775807L; //Long.MAX_VALUE;
```

```
double d = g; //loses precision but ok, implicit widening of long to double is allowed
```

```
i = f; //will not compile, implicit narrowing of float to int is NOT allowed
```

```
g = d; //will not compile, implicit narrowing of double to long is NOT allowed
```

You can, of course, assign a `float` or a `double` to an `int` or a `long` using an explicit cast.

3.3.4 final variables

A **final variable** is a variable whose value doesn't change once it has had a value assigned to it. In other words, the variable is a constant. Any variable can be made final by applying the keyword **final** to its declaration. For example:

```
class TestClass{
    final int x = 10;
    final static int y = 20;

    public static void main(final String[] args){

        final TestClass tc = new TestClass();

        //x = 30; //will not compile.

        //y = 40; //will not compile.

        //args = new String[0]; //will not compile

        //tc = new TestClass(); //will not compile

        System.out.println(tc.x+" "+y+" "+args+" "+tc);
    }
}
```

Observe that in the above code, I have made an instance variable, a static variable, a method parameter, and a local variable final. It prints **10 20 [Ljava.lang.String;@52d1fadb TestClass@35810a60** when compiled and run.

You cannot reassign any value to a final variable, therefore, the four statements

that try to modify their values won't compile.

Remember that when you make a reference variable final, it only means that the reference variable cannot refer to any other object. It doesn't mean that the contents of that object can't change. For example, consider the following code:

```
class Data{  
    int x = 10;  
}  
public class TestClass {  
    public static void main(String[] args){  
        final Data d = new Data();  
  
        //d = new Data(); //won't compile because d is final  
  
        d.x = 20; //this is fine because we are not changing d here.  
    }  
}
```

In the above code, we cannot make **d** refer to a different **Data** object once it is initialized because **d** is final, however, we can certainly use **d** to manipulate the **Data** object to which it points. If you have any confusion about this point, go through the "Object and Reference" section in "Kickstarter for Beginners" chapter.

There are several rules about the initialization of final variables but they depend on the knowledge of initializers and constructors. I will revisit this topic in the "Reusing Implementations Through Inheritance" chapter.

3.4 Wrapper Classes

Wrapper classes were on the OCAJP 8 exam but, surprisingly, they have been removed from the OCP Java 11 Part 1 exam. You may, therefore, ignore the following discussion if you are short on time. However, it is almost necessary for a Java developer to have a basic understanding of wrapper classes because they are widely used.

3.4.1 What are wrapper Classes?

Before I start talking about wrapper classes, let me recall three topics that are quite fundamental to understanding wrapper classes - 1. Object and Reference, 2. Stack and Heap, and 3. Difference between reference variables and primitive variables. I suggest you go through them first if you are not clear on those three concepts.

As you are aware, Java is considered an object-oriented language. Pretty much everything in Java is about objects. You also know that all objects reside on the heap space and can only be accessed through their references, which are cached in reference variables. This is all good but from a performance perspective, heap space is bit heavy as compared to stack space. It is also more permanent than stack space. Recall that JVM performs garbage collection for getting rid of unused objects from time to time. For many simple programming activities such as loops, decision constructs, and temporary data storage, objects seem like too much of a hassle.

To address this concern, Java has the provision of "primitive" data types. They are called primitives because they don't have any behavior associated with them. They are just raw data. If you want to run a loop 10 times, all you need is a counter that can count from 0 to 9. The counter doesn't serve any purpose after the loop is over and so, you don't need any permanent storage to store the counter value. A simple int variable that is forgotten as soon as the loop is done, is sufficient for this purpose. The stack space is perfect for keeping such values because a stack space is wiped clean as soon as the thread that owns that stack space is finished. There is no need of any garbage collection to happen here. Furthermore, a program can directly manipulate the value of a primitive variable without having to go through the indirection of a reference.

The problem with having two different kinds of data types is that it creates a dichotomy between data represented by primitives and data represented by

objects. Due to the difference between how they are stored, passed, and accessed, a program cannot treat primitives and objects the same way. For example, let's say you have developed some logic to process different types of data and you've captured this logic into a method. What type of input parameters would you use to pass the data to this method? If you decide to use Object, you cannot pass any primitive to this method and if you choose a primitive, you cannot pass an object to it. You will either need to write different methods for each kind or have the caller wrap primitive data into objects and pass the objects to your method.

A more concrete example would be a class that manages a collection of data. Java has several standard classes for managing collections. The one most commonly used, and which is also included in the exam objectives, is `java.util.ArrayList`. I will discuss this class in detail later, but basically, you use an `ArrayList` to collect a bunch of objects. If you have a primitive value that you want to keep in the collection managed by an `ArrayList`, you need to wrap that value into an object because `ArrayList` only works with objects.

Java designers realized this problem and added ready-made wrapper classes for each of the primitive data types to Java core library. These classes are - `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `Boolean`. All of these classes are in `java.lang` package.

Wrapper classes meant for the types that are used to represent numeric data, i.e., `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`, extend from a common base class named `java.lang.Number`.

Further, `Number`, `Character`, and `Boolean`, classes extend from `java.lang.Object`.

3.4.2 Creating wrapper objects

There are three ways to create objects of wrapper classes. Let us look at them one by one.

1. **Using constructors** - Like any other object, objects of wrapper classes can be created using their constructors. For example,

```
Integer i1 = new Integer(10);
Integer i2 = new Integer("10");

Character c = new Character('c');

Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean("true");

float f = 10.2f;
Float f1 = new Float(f);
Float f2 = new Float(10.2); //valid, even though 10.2 is a double

Float f3 = new Float("10.2");

short s = 10;
Short s1 = new Short(s);
Short s2 = new Short("10");

Short s3 = new Short(10); //this will not compile.
```

Observe that in the above code, I have created instances of wrapper classes using primitive values as well as **String** values. All wrapper classes except **java.lang.Character** provide two constructors each - one that takes the relevant primitive type and the second one that takes a **String**. **java.lang.Character** class provides only the **char** constructor and does not provide the **String** constructor. Furthermore, **java.lang.Float** class provides a third constructor that takes a **double**, which is why the line **Float f2 = new Float(10.2);** works even though the literal **10.2** is a **double**.

Further observe that **Short s3 = new Short(10);** will not compile

because 10 is an `int` and, as explained above, `Short` does not have any constructor that takes as `int`. It has only two constructors - one that takes a `String` and one that takes a `short`. (Recall that "implicit narrowing" does not happen for method or constructor arguments.)

2. **Using the `valueOf` methods** - All wrapper classes have two static `valueOf` methods each. One that takes the relevant type as a parameter, and the second one that takes a `String`. `Character` class is an exception because it has only one `valueOf` method that takes a `char`. Unlike constructors, `Float` class does not have a third `valueOf` method that takes a `double`! Here are a few examples:

```
float f = 10.2f;
Float f1 = Float.valueOf(f);
Float f2 = Float.valueOf("10.2");

Float f2 = Float.valueOf(10.2); //will not compile
                                because 10.2 is a double
```

```
Integer i1 = Integer.valueOf(10);

Byte b = Byte.valueOf("10");

Boolean bool = Boolean.valueOf("true");
```

In the case of the `String` versions of `valueOf` methods (and also of the constructors), you have to be careful about the value that you pass to the method because if you pass a `null` or a value that cannot be converted into the required wrapper type, a `NumberFormatException` will be thrown. For example, `Integer i = Integer.valueOf("10.2");` will throw a `NumberFormatException` because `10.2` cannot be parsed into an integer.

It is interesting to know that you can pass `null` or any string to `Boolean`'s constructor or `valueOf` method without any exception. All such values will cause it to create a `Boolean` object containing `false`. To get a `Boolean` containing `true`, you can pass "`true`" any case (upper, lower,

or even mixed).

So, what's the point of having `valueOf` methods if you can do the same thing using constructors?

Well, the difference between the two is that a constructor will always create a new object, while the `valueOf` method **may** return a **cached** object. The `valueOf` methods are therefore, more efficient than the constructors and should be used when you don't need to use separate wrapper objects for the same value.

3. **Through auto-boxing** - Up until Java 1.4, every time you wanted to use a wrapper object, you would have had to either use a constructor or a `valueOf` method to create it. Since wrapper objects are used quite often, this was considered too much of a typing effort for such a mundane thing.

To make things a little easier and cleaner, Java 1.5 introduced the concept of "auto-boxing". All it means is that if you assign a primitive value to a wrapper variable, the compiler will automatically box the primitive value into a wrapper object. So, basically, instead of writing `Integer i = Integer.valueOf(100);` you can just write `Integer i = 100;`.

Similarly, if a method expects an object as an input parameter, you can just pass in the primitive and the compiler will automatically box it into a wrapper object. For example, instead of writing
`myList.add(Integer.valueOf(100));`, you can write,
`myList.add(100);`.

Observe that I have used the `valueOf` method instead of a constructor to illustrate the equivalency of the explicit creation of wrapper objects and autoboxing. This is deliberate. The reason is that autoboxing for `byte`, `short`, `char`, `int`, `long`, and `boolean` uses cached objects instead of creating new instances just like the `valueOf` methods. Thus, `i1` and `i2` in the following code will refer to the same `Integer` wrapper object:

```
Integer i1 = 100;  
Integer i2 = 100; //i2 will refer to the same  
object as i1.
```

Just like the `valueOf` methods, autoboxing of values from `-128` to `127`, `true` and `false`, and `'\u0000'` to `'\u007f'` will result in cached objects. Wrappers for other values may also be cached but that is not guaranteed.

3.4.3 Converting wrapper objects to primitives [↳](#)

There are two ways to get primitive values from wrapper objects:

1. **Using `xxxValue` methods** - All wrapper classes provide an instance method that returns the value wrapped by that wrapper object as a primitive. The name of this method follows the pattern `<type>Value`. For example, `Integer` class has `intValue`, `Boolean` class has `booleanValue`, and `Character` class has `charValue` that returns `int`, `boolean`, and `char` respectively.

Recall that wrapper classes for numeric types (i.e. `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`) have a common base class called `Number`. This class defines `byteValue` and `shortValue` methods and declares `intValue`, `longValue`, `floatValue`, and `doubleValue` methods. Therefore, it is possible to get a primitive value of any of these types from any of numeric wrapper class objects.

2. **Through unboxing** - This is just the opposite of autoboxing. You can assign (or pass as an argument) any wrapper object to a variable of primitive type directly and the compiler will automatically extract the primitive value from it and assign it to the target. For example,

```
Integer i1 = 10; //autoboxing int value 10 into an Integer object.  
int i2 = i1; //unboxing Integer object and assigning the resulting value to an int variable.
```

Remember that unboxing will compile only if the type of the target is wide enough to accept the type of the wrapper. For example, `byte b = i1;` will not compile because `int` is wider than `byte`, while `float f = i1;` will compile fine because `float` is wide enough to hold any `int` value.

Besides letting you convert wrapper objects to primitives, wrapper classes also contain `parseXXX` methods that let you get primitive values from Strings. For example, `Integer` has `parseInt` method that takes in a `String` as an argument and returns an `int`. Of course, the argument must have a valid value that can be parsed as an `int` otherwise a `NumberFormatException` will be thrown. There are several variations of this method but you need not memorize them for the exam.

Wrapper classes contain many useful methods and even though they are not on the exam, I suggest you quickly browse through their API descriptions, as that will help you on the job.

3.5 Exercise

1. Identify all the primitive and reference data types as well as primitive and reference variables used in the following code:

```
public class Person {  
    int id;  
    String name;  
    java.util.Date dob;  
    boolean VIP;  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p2 = p1;  
        int id = p2.id;  
        p1.name = args[0];
```

```
    }  
    public String getName(){ return name; }  
}
```

2. What are the values of the primitive variables used in the above code. What operations can be performed on these variables?
3. Identify the methods that can be invoked using the reference variables used in the above code and also identify the objects on which those methods will be invoked.
4. Change the declaration of various instance members of Person class to include initial values.

Chapter 4 Describing and Using Objects and Classes

- Declare and instantiate Java objects
- Define the structure of a Java class
- Read or write to object fields
- Identify the scope of variables
- Use local variable type inference
- Explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)

4.1 Declare and instantiate Java objects

4.1.1 Declare and instantiate Java objects

In the previous chapter you saw the difference between primitive variables and reference variables and how to declare both kind of variables. The basic syntax of declaring primitive and reference variables is the same:

```
<type name> <variable name>;
```

In case of reference variables, the type name can be a simple name or a FQCN depending on whether the class (or its package) has been imported using an appropriate import statement or not. For example:

```
String str;  
Object obj;  
SomeClass scRef; //assuming com.abc.SomeClass has been imported  
  
com.xyz.SomeOtherClass socRef; //assuming com.xyz.Som
```

eClass hasn't been imported

You can declare multiple variables of the same type in a single statement:

```
SomeClass scRef1, scRef2;
```

As with primitive variables, it is possible to assign a value to a reference variable at the time of declaration itself:

```
SomeClass scRef1 = null; //scRef1 is initialized with null.
```

```
SomeClass scRef2 = scRef1, scRef3; //scRef2 is initialized with the same value as scRef1 but scRef3 remains uninitialized.
```

```
Object a = null, b = a; //a is initialized to null and then a is assigned to b
```

Here are a couple of invalid declarations:

```
String str1, Object o1; //can't change the type within a statement
```

```
Object a = b, b = null; //b is being assigned to a but b hasn't been defined at that point
```

You will not get overly tricky questions on declaration of variables in

the exam. If you stick to the basics shown above, you will not have trouble answering exam questions.

Instantiating objects

As discussed in the previous chapter, a class defines a new data type. You define data types because they allow you to group raw data and give that group a special meaning in your application. For example, if you are dealing with names, dates of birth and addresses of people in your application, you may group the three raw data elements into a Person class and add behavior to Person class in terms of methods. You would then have several "persons" in your application. Each person will be modeled upon the same Person template. Every person would actually be just an instance of the Person class. So, two persons may have different names and other details but they will exhibit the same behavior. This commonality of behavior allows your application to treat all person instances in the same manner.

Once you have defined the template, it is easy to construct objects. Java has exactly three ways of instantiating objects - using the **new** keyword, by deserializing an object's data, and by cloning an existing object. Here, we will only focus on the **new** keyword because deserialization and cloning are not on the Part 1 exam.

The following is probably the simplest example of instantiating an object using the **new** keyword:

```
new java.lang.Object();
```

The above statement creates an instance of **java.lang.Object** class. Since a class is just a template...a cookie cutter, if you will, which is used to cut objects out of free memory, when you use the new keyword on a class, the JVM takes a chunk of free memory and formats that memory into the various fields defined in the class. This formatted chunk of memory becomes an object of the class. You can create as many instances as you want using this template.

Depending on how a class has been defined, you may be able to (or even required to) pass argument(s) while instantiating an object. For example, the

following statement creates an instance of String class with a string parameter:

```
new String("hello");
```

The **new** keyword actually causes a "constructor" of the class to execute. The constructor is responsible for initializing the data members of the instance. You will see the exact mechanism of object creation in the "Creating and using Methods" chapter.

As discussed earlier, a reference variable doesn't contain the actual object but just the address (or reference) to the object. This address is available only at the time of instantiation of an object and if you want to make use of that object later on, you must save that address somewhere. Usually, you store it in a variable, like this:

```
String str = new String("1234");
```

Since the address of the String object is saved in **str** variable, you can invoke methods on this object using the **str** variable. If you don't save the address of the object in a variable that you can access, that object will be lost after creation. In some cases, you may want exactly that. You may want to just create an object and forget about it after that. In such a case, there is no need to save its address in a variable.

You may see the term "instantiate a class" being used in articles and books instead of "instantiate an object". Both the terms mean the same thing and are equally acceptable. Since official exam objectives use the term "instantiate an object", I have used the same. Similarly, the words "create" and "instantiate" are also used interchangeably in the context of an object. So, "create an object" and "instantiate an object" mean the same thing.

Assigning objects to reference variables 

Since Java is a strongly typed language, you cannot assign just about any object to a reference variable of any type. The rule regarding object assignment is actually quite simple but requires an understanding of inheritance and polymorphism. Since I haven't discussed these concepts yet, assume that if the type of the variable is **A**, then only an object of type **A** or a sub-type of **A** can be assigned to that variable. For example, `Object obj = new String();` is valid because **String** is a subtype of **Object** but `String str = new Object();` is not valid. In later chapters, you will see how this rule applies to interfaces, abstract classes, and variables.

4.2 Read or write to object fields

4.2.1 Accessing object fields

By object fields, we mean instance variables of a class. Each instance of a class gets its own personal copy of these variables. Thus, each instance can potentially have different values for these variables. To access these variables, i.e., to read the values of these variables or to set these variables to a particular value, we must know the exact instance whose variables we want to manipulate. We must have a reference pointing to that exact object to be able to manipulate that object's fields.

Recall our previous discussion about how an object resides in memory and a reference variable is just a way to address that object. To access the contents of an object or to perform operations on that object, you need to first identify that object to the JVM. A reference variable does just that. It tells the JVM which object you want to deal with.

For example, consider the following code:

```
class Student{  
    String name;  
}  
public class TestClass{  
    public static void main(String[] args){  
        Student s1 = new Student();
```

```

Student s2 = new Student();

s1.name = "alice";
System.out.println(s1.name); //prints alice

System.out.println(s2.name); //prints null

s2.name = "bob";
System.out.println(s1.name); //prints alice

System.out.println(s2.name); //prints bob

}

}

```

In the above code, we created two Student objects. We then set the name variable of one Student object and print names of both the Student objects. As expected, the name of the second Student object is printed as null. This is because we never set the second Student object's name variable to anything. The JVM gave it a default value of `null`.

Next, we set the name variable of the second Student object and print both the values again. This time we are able to see the two values stored separately in two Student instances.

This simple exercise shows how to manipulate fields of an object. We take a reference variable and apply the dot operator and the name of the variable to reach that field of the object pointed to by that variable. It is not possible to access the fields of an object if you do not have a reference to that object. You may store the reference to an object in a variable (such as `s1` and `s2` in the code above) when that object is created and then pass that reference around as needed. Sometimes you may not want to keep a reference in a variable. This typically happens when you want to create an object to call a method on it just once. The following piece of code illustrates this:

```

class Calculator{
    public int calculate(int[] iArray){
        int sum = 0;
        for(int i : iArray){ //this is a for-each loop,
we'll cover it later
            sum = sum+i;
        }
        return sum;
    }
}
public class TestClass {
    public static void main(String[] args) {
        int result = new Calculator().calculate( new
int[]{1, 2, 3, 4, 5} );
        System.out.println(result);
    }
}

```

Observe that we created two objects in the main method but did not store their references anywhere - a **Calculator** object and an array object. Then we called an instance method on the **Calculator** object directly without having a reference variable. Since the method call is chained directly to the object creation, the compiler is able to create a temporary reference variable pointing to the newly created object and invoke the method using that variable. However, this variable is not visible to the programmer and therefore, after this line, we have lost the reference to the **Calculator** object and there is no way we can access the same **Calculator** object again.

Within the **calculate** method, the same **Calculator** object is available though, through a special variable called "**this**", which is the topic of the next section.

Similarly, the compiler created a temporary reference variable for the array object and passed it in the method call. However, we don't have any reference to this array object after this line and so, we cannot access it anymore. Within the calculate method, however, a reference to that array object is available through the method parameter **iArray**.

4.2.2 What is "this"?

Let us modify our `Student` class a bit:

```
class Student{  
    String name;  
  
    public static void main(String[] args) {  
        Student s1 = new Student(); //1  
  
        s1.name = "mitchell"; //2  
  
        s1.printName(); //3 prints mitchell  
  
    }  
  
    public void printName(){  
        System.out.println(name); //5  
  
    }  
}
```

I mentioned earlier that it is not possible to access the fields of an object without having a reference to that object. But at //5, we are not using any such reference. How is that possible? How is the JVM supposed to know which `Student` instance we mean here?

Observe that at //3, we are calling the `printName` method using the reference variable `s1`. Therefore, when the JVM invokes this method, it already knows

the instance on which it is invoking the method. It is the same instance that is being pointed to by s1. Now, in Java, if you don't specify any reference variable explicitly within any instance method, the JVM assumes that you mean to access the same object for which the method has been invoked. Thus, within the `printName` method, the JVM determines that it needs to access the name field of the same `Student` instance for which `printName` method has been invoked. You can also explicitly use this reference to the same object by using the keyword "`this`". For example, //5 can be written as:

```
System.out.println(this.name);
```

Thus, the rule about having a reference to access the instance fields of an object still applies. Java supplies the reference on its own if you don't specify it explicitly.

By automatically assuming the existence of the reference variable "this" while accessing a member of an object, Java saves you a few keystrokes. However, it is not considered a good practice to omit it. You should always type "this." even when you know that you are accessing the field of the same object because it improves code readability. The usage of "this." makes the intention of the code very clear and easy to understand.

When is "this" necessary?

When you have more than one variable with the same name accessible in code, you may have to remove the ambiguity by using the `this` reference explicitly. This typically happens in constructors and setter methods of a class. For example,

```
class Student{  
    String id;  
    String name;  
    public void setName(String code, String name){  
        id = code;  
        name = name;
```

```
    }  
}
```

In the `setName` method, four variables are accessible: two method parameters - `code` and `name`, and two instance variables - `id` and `name`. Now, within the method code, when you do `id = code;` the compiler knows that you are assigning the value of the method parameter `code` to the instance field `id` because these names refer to exactly one variable each. But when you do `name = name;`, the compiler cannot distinguish between the two `name` variables. It thinks that `name` refers to the method parameter and assigns the value of the method parameter to itself, which is basically redundant and is not what you want. There is nothing wrong with it from the compiler's perspective but from a logical perspective, the above code has a serious bug. In technical terms, this is called "**shadowing**". A variable defined in a method (i.e. either in parameter list or as a local variable) shadows an instance or a static field of that class. It is not possible to access the shadowed variable directly using a simple name. The compiler needs more information from the programmer to disambiguate the name.

To fix this, you must tell the compiler that the name on left-hand side of `=` should refer to the instance field of the `Student` instance. This is done using "`this`", i.e., `this.name = name;`.

While we are on the topic of shadowing, I may as well talk about shadowing of static variables of a class by local variables. Here is an example:

```
class Student{  
    static int count = 0;  
    public void doSomething(){  
        int count = 10;  
        count = count; //technically valid but logically incorrect  
    }  
}
```

`Student.count = count; //works fine in instance as well as static methods`

```
        this.count = count; //works fine in an instance  
method  
  
    }  
}
```

The above code has the same problem of redundant assignment. The local variable named `count` shadows the static variable by the same name. Thus, inside `doSomething()`, the simple name `count` will always refer to the local variable and not to the static variable. To disambiguate `count`, ideally, you should use `Student.count` if you want to refer to the static variable but if you are trying to use it from an instance method, you can also use `this.count`. Yes, using `this` is a horrible way to access a static variable but it is permissible. I will talk more about static fields and methods in the "Creating and Using Methods" chapter.

Redundant assignment is one of the traps that you will encounter in the exam. Most IDEs flash a warning when you try to assign the value of a variable to the same variable. But in the exam, you won't get an IDE and so, you must watch out for it by reading the code carefully.

Here are a few quick facts about `this`:

1. `this` is a keyword. That means you can't use it for naming anything such as a variable or a method.
2. The type of `this` is the class (or an enum) in which it is used. For example, the type of `this` in the `printName` method of `Student` class is `Student`.
3. `this` is just like any other local variable that is set to point to the instance on which a method is being invoked. You can copy it to another variable. For example, you can do `Student s3 = this;` in an instance method of `Student` class.

4. You can't modify `this`, i.e., you can't set it to null or make it point to some other instance. It is set by the JVM. In that sense, it is final.
5. `this` can only be used within the context of an instance of a class. This means, it is available in instance initializer blocks, constructors, instance methods, and also within a class. It is not available within a static method and a static block because static methods (and static initializer blocks) do not belong to an object.

4.3 Define the structure of a Java class

4.3.1 Class disambiguated [↳](#)

The word "class" may mean multiple things. It could refer to the OOP meaning of class, i.e., an abstraction of an entity, it could refer to the code written in a Java source file, or it could refer to the output of the Java compiler, i.e., a file with .class extension.

For example, let us say you are developing an application for a school. You could model the **Student** entity as a class. In this case, **Student** is a class in the OOP sense. When you actually start coding your application, you would write the code for Student class in **Student.java** file. Finally, you would compile **Student.java** using **javac** and produce **Student.class** file which contains the bytecode for the Student class.

The exam focuses primarily on the source code aspect of a class, i.e., the contents of **Student.java** file of the above example. However, you do need to know the basics of OOP as well because, after all, a Java source file is meant to let you write code for your OOP class model. You don't have to worry about the bytecode version of a class.

4.3.2 Structure of a Java source file [↳](#)

If you have written some Java code before, you are already aware of the basic structure of a Java source file. I will do a quick recap and then move on to the interesting situations and gotchas that you need to know for the exam.

A Java source file has the following three parts:

Part 1: zero or one **package statement**

Part 2: zero or more **import statements**

Part 3: one or more reference type (i.e. class, interface, or enum) definitions.

The ordering mentioned above is important. For example, you cannot have the package statement after the import statements or the class declaration(s). Similarly, you cannot have import statements after the class declaration.

A Java source file must define at least one Java reference type definition in it. You can define multiple reference types within a single source file as well. I will talk about the rules of that later.

4.3.3 Members of a class [↑](#)

Within a class definition, you can have **field declarations, methods, constructors, and initializers**. You can also have classes, interfaces, and enums, but more on that later. All of these are called "**members**" of that class.

Members can be defined as **static** or **non-static** aka **instance** (a member that is not defined as static is automatically non-static).

For example, the following code shows various members of a class:

```
//package com.school; //optional
```

```
import java.util.Date;  
//required because we are using Date instead of java.util.Date in the code below
```

```
public class Student
{
    private static int count = 0; //static field

    private String studentId; //instance field

    static{ //static initializer

        System.out.println("Student class loaded");
    }

    { //instance initializer

        Student.count = Student.count +1;
        System.out.println("Student count incremented"
);
    }

    public Student(String id){ //constructor

        this.studentId = id;
        System.out.println(
            new Date() +
            " Student instance created. Total student
s created = "+count);
    }

    public String toString(){ //instance method
```

```
        return "Student[studentId = "+studentId+"]";
    }

    public static void main(String[] args) { //static
method

        Student s = new Student("A1234");
        System.out.println(s.toString());
    }

}
```

The package statement at the top makes the **Student** class a member of **com.school** package. The **import** statement lets you use **Date** class of **java.util** package in the code by typing just **Date** instead of **java.util.Date** .

The class uses a static field named **count** to track the number of **Student** objects that have been created. Instance field **studentId** stores an id for each Student instance.

The **static initializer** is executed only once when the class is first loaded by the JVM and the **instance initializer** is executed just before the constructor every time an instance is created. Don't worry if you don't understand the purpose of static and instance initializer blocks. We will go deep into this later.

Then there is a constructor that allows you to create Student objects with a given id and the static **main method** that allows you to execute this class from command line. (Notice that I have commented out the package statement so that it will be easier to execute the class from command line without worrying about the directory structure.)

The following output is produced upon executing this class:

```
Student class loaded
Student count incremented
```

```
Mon Jul 31 09:35:19 EST 2017 Student instance created.  
Total students created = 1  
Student[studentId = A1234]
```

Important - You cannot have any statement in a class that does not belong to any of the categories specified above. For example, the following will not compile:

```
public class Student{  
    String id = ""; //this is ok because this statement  
    is a declaration  
  
    id = "test"; //this is not ok because this is a simple  
    statement that is not a declaration, or an initializer  
    block, or a method, or a constructor.  
  
    { //this is ok because it is an initializer block  
  
        id = "test"; //this is ok because it is inside an  
        instance initializer block and not directly within the  
        class.  
  
    }  
}
```

Comments

Java source files can also contain comments. **You will not get questions on comments in the exam** but it good to know that there are two ways to write comments in a Java source file - a single line comment, which starts with a // and closes automatically at the end of the line (that means you don't close it explicitly) closing and a multi line comment, which opens with /* and closes with */. Multi line comments don't nest. Meaning, the comment will start with a /* and end as soon as the first */ is encountered.

Comments are completely ignored by the compiler and have no impact on the resulting class file.

The following are a few examples:

```
//this is a short comment

/*
This is a multi line
comment.
*/

/*
This is a multi line
comment.
//This is another line inside a comment
*/
```

JavaDoc Comments [↳](#)

Java promotes writing well documented code. It allows you to write descriptions for fields, methods, and constructors of a class through smart use of comments. If you write comments in a certain format, you can produce HTML documentation for your code using the JavaDoc tool. This format is called the JavaDoc comment format and it looks like this:

```
/**  
 * Observe the start of the comment. It has an extra  
 * Each line starts with a *  
 * There is a space after each *  
 * <h3>You can write HTML tags here.</h3>  
 * Description of each parameter starts with @param  
 * Description of the return value starts with @return  
 * @see tag is used to add a hyperlink to description  
 * of another class
```

```
* @param name the location of the image, relative to  
the url argument  
* @return the image at the specified URL  
* @see SomeOtherClassName  
*/
```

```
public String sayHello(String name) {  
    return "Hello, "+name;  
}
```

The JavaDoc tool comes bundled with the JDK. It can extract all the information contained in the comments written in the above format and generate nicely formatted HTML documentation. In fact, all of the standard Java library classes contain descriptions in the above format. It is these descriptions that are used to generate the HTML pages of the Java API documentation automatically using the javadoc tool.

4.3.4 Relationship between Java source file name and class name



Other than the fact that Java source files have an extension **.java** (or **.jav**), there is **only one rule** about the class name and the name of its source code file - the code for a **top level public** type (recall that "type" implies class, interface, or enum) must be written inside a Java file with the same name (with extension dot java, of course!).

For example, if you are writing code for a public class named **Student**, then the name of the source code file must be **Student.java**

In light of the above rule, let us take a look at a few questions that might pop into your head:

Q. Does that mean I cannot have multiple classes in a single file?

A. No, you certainly can have multiple classes in a single file. But only one of

them can be public and the name of that public class must be the same as the name of the file. It is okay even if there is no public class in a file.

Q. What if I don't have a public class? What should be the name of the file in that case?

A. You can code a non-public class in a file with any name. However, it is a good programming practice to keep even a non-public class in a file by the same name.

Q. What about interfaces? Enums?

A. The rule applies to all types, i.e., classes, interfaces, and enums. For example, you cannot have a public class and a public interface in the same file. There can be only one public type in one file.

Q. What about nested types? Can I have two public classes inside a class?

A. The rule applies only to top level types. So, yes, you can have more than one public types inside another type. For example, the following is valid:

```
public class TestClass
{
    public interface I1{ }
    public class C1{ }
    public static class C2{ }
    public enum E1{ }
}
```

I1, C1, C2, and E1 are called "nested types" aka "nested classes" because their declaration appears within the body of another class or an interface. Types that are not nested inside other types are called "top level" types. The topic of nested classes is not included in the Part 1 exam but is included in Part 2 exam, so, it is good to know at least the terminology at this stage.

Remember that this restriction is imposed by the Java compiler and not the JVM. Compiler converts the source code into class files and generates an independent

class file for each type (irrespective of whether that type is public or not) defined in that source file. Thus, if you define three classes in Java file (one public and two non-public), three separate class files will be generated. The JVM has no idea about the Java source file(s) from which the class files originated.

It is a common practice, however, to define each type, whether public or not, in its own file. Defining each type in its own independent file is a very practical approach if you think about it. While browsing the code folder of a Java project, you only see the file name. Since you cannot see inside the file, it will be very hard for you to find out which class is defined in which Java file if you have multiple definitions in a single Java file.

It is interesting to know (though not required for the exam) that imposition of this rule is actually **optional**. A compiler may choose to ignore this rule altogether. **Java language specification, Section 7.6** mentions that this rule may be imposed by a Java compiler only if the source code is stored in the file system and the type in that source file is being referred to by other types. Thus, it is possible for a compiler to ignore this rule if, for example, the code is stored in the database. Or if the type defined in a file is not referred to by other types.

For the purpose of the exam, all you need to know is that Oracle's Java compiler enforces this rule.

You may see multiple public classes in the code listing of a question. But don't immediately jump to the conclusion that the code will not compile. Unless the problem statement explicitly says that these classes are written in the same file, Oracle wants you to assume that they are written in separate files.

Directory in which source files should reside [🔗](#)

Although it is a common (and a good) practice to keep the source file in the directory that matches the package name in the file, there is no restriction on the directory in which the source file should reside. For example, if the package statement in your **Student.java** file is **com.university.admin**, then you should keep **Student.java** file under **com/university/admin** directory. IDEs usually

enforce this convention. So, if you are using an IDE, you may see errors if you keep **Student.java** file anywhere else but remember that this is not required by the Java language. You can still compile the file from the command line. Check out the **Compilation and Execution** section under **Kickstarter for Beginners** chapter to understand the manual compilation process. In the Modules chapter, you will see that keeping source files organized this way has an additional benefit.

4.3.5 Quiz

The following options show complete code listings of a Java file named Student.java. Which of these will compile without any error?

Select 1 correct option.

A.

//Start of file

```
public class Student{  
}  
public enum Grade{ A , B, C, D }  
//End of file
```

B.

//Start of file

```
class Student{  
}  
enum Grade{ A , B, C, D }  
enum Score{ A1 , A2, A3, A4 }  
//End of file
```

C.

//Start of file

```
public interface Gradable{  
}  
public interface Person{  
}  
//End of file
```

D.

//Start of file

```
class Student{  
}  
public class Professor{  
}  
//End of file
```

E.

//Start of file

```
package com.enthuware.ocajp;  
//End of file
```

Correct answer is B.

Option A and C are **incorrect** because you cannot define more than one public top level type in a source file. Option D is **incorrect** because the Professor class

is public. A public class must reside in a file by the same name but here, the name of the file is `Student.java`. Option **E** is incorrect because every source file must have at least one Java artifact defined in it. Option **B** is **correct** because Java allows a file to have any number of non-public types.

4.4 Identify the scope of variables

4.4.1 Scope of variables [↳](#)

Java has three **visibility scopes** for variables - class, method, and block.

Java has five **lifespan scopes** for variables - class, instance, method, for loop, and block.

4.4.2 Scope and Visibility [↳](#)

Scope means where all, within a program, a variable is visible or accessible directly without any using any referencing mechanism.

For example, the scope of a President of a country is that country. If you say "The President", it will be interpreted as the person who is the president of the country you are in. There cannot be two presidents **of** a country. If you really want to refer to the presidents of two countries, you must also specify the name of the country. For example, the President of US and the President of India.

At the same time, you can certainly have two presidents **in** a country - the President of the country, and the President of a basketball association within that country! If you are in your basketball association meeting, and if you talk about the president in that meeting, it will be interpreted as person who is the president of the association and not the person who is the president of your country. But if

you do want to mean the president of the country, you will have to clearly say something like the "President of our country". Here, "of our country" is the referencing mechanism that removes the ambiguity from the word "president".

In this manner, you may have several "presidents" in a country. All have their own "visibility". Depending on the context, one president may shadow or hide (yes, the two words have different meanings in Java) another president. But you cannot have two presidents in the same "visibility" level.

This is exactly how "scope" in Java (or any other programming language, for that matter) works. For example, if you declare a static variable in a class, the **visibility** of that variable is the class in which you have defined it. The visibility of an instance variable is also the class in which it is defined. Since both have same visibility, you cannot have a static variable as well as an instance variable with the same name in the same class. It would be like having two presidents of a country. It would be absurd and therefore, invalid.

If you declare a variable in a method (either as a method parameter or within a method), the visibility of that variable is within that method only. Since a method scope is different from a class scope, you can have a variable with the same name in a method. If you are in the method and if you try to refer to that variable directly, it will be interpreted as the variable defined in the method and not the class variable. Here, a method scoped variable **shadows** a class scoped variable. You can, of course, refer to a class scoped variable within a method in such a case but you would have to use the class name for a static variable or an object reference for an instance variable as a referencing mechanism to do that.

Similarly, if you declare a variable in a loop, the visibility of that variable is only within that loop. If you declare a variable in a block such as an if, do/while, or switch, the visibility of that variable is only within that block.

Here, visibility is not to be confused with **accessibility** (public/private/protected). Visibility refers to whether the compiler is able to see the variable at a given location directly without any help.

For example, consider the following code:

```
public class Area{  
    public static String UNIT="sq mt"; //UNIT is visible all over inside the class Area
```

```

        public void printUnit(){
            System.out.println(UNIT); //will print "sq mt
" because UNIT is visible here

    }

}

public class Volume{
    //Area's UNIT is accessible in this class but not
    visible to the compiler directly.

    public static String UNIT="cu mt";

    public void printUnit(){
        System.out.println(UNIT); //will print "cu mt

        System.out.println(Area.UNIT); //will print "
sq mt"

    }
}

```

In the above code, a public static variable named **UNIT** of a class **Area** is accessible to all other classes but that doesn't mean another class **Volume** cannot also have a static variable named **UNIT**. This is because within the **Volume** class, **Area**'s **UNIT** is not directly visible. You would need to help the compiler by specifying **Area.UNIT** if you want to refer to **Area**'s **UNIT** in class **Volume**. Without this help, the compiler will assume that you are talking about **Volume**'s **UNIT**.

Besides shadowing and hiding, there is a third category

of name conflicts called "obscuring". It happens when the compiler is not able to determine what a simple name refers to. For example, if a class has a field whose name is the same as the name of a package and if you try to use that simple name in a method, the compiler will not know whether you are trying to refer to the field or to a member of the package by the same name and will generate an error. It happens rarely and is not important for the exam.

4.4.3 Scope and Lifespan

Scope and Lifespan

Besides visibility, **scope** is also related to the **lifespan** or **life time** of a variable. Think of it this way - what happens to the post of the president of your local basketball association if the association itself is dissolved? The post of the president of the association will not exist anymore, right? In other words, the life of the post depends on the existence of the association.

Similarly, in Java, the existence of a variable depends on the existence of the scope to which it belongs. Once its life time ends, the variable is destroyed, i.e., the memory allocated for that variable is taken back by the JVM. From this perspective, Java has five scopes: **block**, **for loop**, **method**, **instance**, and **class**.

When a block ends, variables defined inside that block cease to exist. For example,

```
public class TestClass
{
    public static void main(String[] args){
        {
            int i = 0; //i exists in this block only
    }
}
```

```
        System.out.println(i); //OK

    }

    System.out.println(i); //NOT OK because i has already gone out of scope.

}

}
```

Variables defined in a for loop's initialization part exist as long as the for loop executes. Notice that this is different from variables defined inside a for block, which cease to exist after each iteration of the loop. For example,

```
public class TestClass
{
    public static void main(String[] args){
        for(int i = 0; i<10; i++)
        {
            int k = 0; //k is block scoped. It is reset to 0 in each iteration.

            System.out.println(i); // i retains its value from previous iteration.

        }
        //i and k are both out of scope here.

    }
}
```

```
}
```

When a method ends, the variables defined in that method cease to exist.

When an object ceases to exist, the instance variables of that object cease to exist.

When a class is unloaded by the JVM (not important for the exam), the static variables of that class cease to exist

It is important to note here that lifespan scope doesn't affect compilation. The compiler checks for the visibility scope only. In case of blocks, loops, and, methods, the lifespan scope of the variables coincides with the visibility scope. But it is not so for class and instance variables. Here is an example:

```
public class TestClass
{
    int data = 10;
    public static void main(String[] args){
        TestClass t = new TestClass();
        t = null;
        System.out.println(t.data); //t.data is accessible therefore, it will compile fine even though the object referred to by t has already ceased to exist.

    }
}
```

Lifespan scope affects the run time execution of the program. For example, the above program throws a **NullPointerException** at run time because **t** doesn't exist and neither does **t.data** at the time we are trying to access **t** and **t.data**.

4.4.4 Scopes Illustrated ↗

The following code shows various scopes in action.

```
class Scopes{
    int x;//visible throughout the class

    static int y;//visible throughout the class

    public static void method1(int param1){ //visible
        throughout the method

        int local1 = 0; //visible throughout the method

        {

            int anonymousBlock = 0;//visible in this b
lock only

        }

    anonymousBlock = 1;//compilation error

    for(int loop1=0; loop1<10; loop1++){
        int loop2 = 0;
        //loop1 and loop2 are visible only here.

    }
}
```

```
loop1 = 0;//compilation error

loop2 = 0;//compilation error

if(local1==0){
    int block1 = 0;//visible only in this if
block

}

block1 = 7; //compilation error

switch(param1){
    case 0:
        int block2 = 10;//visible all over cas
e block

        break;
    case 1:
        block2 = 5;//valid

        break;
    default:

        System.out.println(block2);//block2 is
visible here but compilation error because block2 may
be left uninitialized before access

}
```

```

        block2 = 9; //compilation error

        int loop1 = 0, loop2 = 0, block1 = 0, block2 =
8; //all valid

    }

}

```

4.4.5 Scope for the Exam ↗

The important thing about scopes that you must know for the exam is when you can and cannot let the variable with different scopes overlap. A simple rule is that you cannot define two variables with the same name and same visibility scope. For example, check out the following code:

```

class Person{
    private String name; //class scope

    static String name = "rob"; //class scope. NOT OK be
    cause name with class scope already exists.

    public static void main(String[] args){
        for(int i = 0; i<10; i++){
            String name = "john"; //OK. name is scoped o
            nly within this for loop block

        }
    String name = "bob"; //OK. name is method scoped.
}

```

```
        System.out.println(name); //will print bob  
    }  
}
```

In the above code, the static and instance name variables have the same visibility scope and therefore, they cannot coexist. But the name variables inside the method and inside the for loop have different visibility scopes and can therefore, coexist.

But there is an **exception** to this rule. Consider the following code:

```
class Person{  
    private String name; //name is class scoped  
  
    public static void main(String[] args){  
        String name = "bob"; //method scope. OK. Overlaps  
        with the instance field name defined in the class  
  
        int i = -1; //method scope  
  
        for(int i = 0; i<10; i++){ //i has for loop scope  
            . Not OK.  
  
            String name = "john"; //block scope. Not OK.  
  
        }  
        { //starting a new block here
```

```
    int i = 2; //block scope. Not OK.  
}  
}  
}
```

Observe that it is possible to overlap the instance field with a method local variable but it is not possible to overlap a method scoped variable with a loop or block scoped variable.

4.4.6 Quiz

Q1. What will the following code print when compiled and run?

```
public class TestClass {  
    public static void main(String[] args) {  
  
        {  
            int x = 10;  
        }  
        System.out.println(x);  
    }  
}
```

Select 1 correct option

- A. It will not compile.
- B. It will print 10
- C. It will print an unknown number

The correct answer is A.

Notice that x is defined inside a block. It is not visible outside that block. Therefore, the line `System.out.println(x);` will not compile.

Q.2 What will the following code print when compiled and run with the command line:

```
java ScopeTest hello world
```

```
public class ScopeTest {  
    private String[] args = new String[0];  
  
    public static void main(String[] args) {  
        args = new String[args.length];  
        for(String arg: args){  
            System.out.println(arg);  
        }  
        String arg = args[0];  
        System.out.println(arg);  
    }  
}
```

Select 1 correct option

A. It will not compile.

B. It will print:

```
null  
null  
null
```

C. It will print:

```
null  
null  
hello
```

Answer is B.

The line `args = new String[args.length];` creates a new string array with the same length as the length of the original string array passed to the program and assigns it back to the same variable `args`. All the elements of this new array are `null`.

The original string array passed to the program is lost.

The instance variable `args` is not touched here because it is shadowed in the method code by the method parameter named `args`. Also, you need to have a reference to an object of class `ScopeTest` to access an instance variable from a static method.

4.5 Use local variable type inference

4.5.1 Local variable type inference

One among a few criticisms of Java is that it is too verbose. Meaning, even writing simple code requires too much typing. Can't really argue with that when a simple program that prints `hello world` takes about a hundred characters! The extends keyword is another example. What takes merely one character (`:`) in C++, requires seven in Java. This is not an oversight by Java designers. Java is actually designed to make the programmer state their intention very clearly, in an unambiguous and easy to understand fashion, instead of making the reader infer the intention of the coder based on the context.

In Java's defense, verbosity makes the code more readable and thus, easily maintainable. It also makes the code less prone to bugs.

On the other hand, too much verbosity poses problems of its own. It would be pretty annoying if you had to type

```
com.enthware.ets.data.MultipleChoiceQuestion q = new
```

`com.enthuware.ets.data.MultipleChoiceQuestion();` (103 characters) every time you wanted declare and create a `MultipleChoiceQuestion` object. Of course, importing `com.enthuware.ets.data` package reduces it to just `MultipleChoiceQuestion q = new MultipleChoiceQuestion()`. This is possible because the compiler is able to infer from the context that by `MultipleChoiceQuestion`, the developer really means `com.enthuware.ets.data.MultipleChoiceQuestion`.

Well, Java 10 helps you make the above statement even shorter. Java 10 onward, you can just write `var q = new MultipleChoiceQuestion();` (38 characters) and it will mean the same thing. Since you are creating an object of type `MultipleChoiceQuestion`, the compiler has no problem in inferring that the type of the variable would also be the same, i.e., `MultipleChoiceQuestion`. This is what type inference essentially means. The logic behind this shortcut is that if the compiler can unambiguously infer the type of the variable from the context, why make the programmer type it explicitly?

Now, about the "local variable" part. It may be possible for a compiler to infer the type of a variable in several contexts. However, Java allows it to do so only for a local variable. As you know, local variables are variables that are defined inside a method. Therefore, you can use this feature only inside a method body. This feature is commonly referred to as **LVTI** in short and, since it uses the word `var` in the syntax, it is also known as **var declaration**.

Let us now see a few examples of valid and invalid usages of this feature. Valid usages first:

```
public class LVTITest1 {  
    static{  
        var str1 = "hello1"; //valid in static as well as  
        instance initializers  
    }  
    public LVTITest1(){  
    }
```

```
var str2 = "hello2"; //valid in constructors

}

public static void main(String[] args)
{
    var i = 10; //type of i is int because 10 is an int

    var f = 1.0f; //type of f is float because 1.0f is
a float

    var strA = new String[]{"a", "b" }; //type of strA
is String[]

    var d = Math.random(); //type of d is double because
the return type of Math.random is double

    Object obj = "hello"; //valid, assigning a String to
an Object variable

    var obj2 = obj; //type of obj2 is Object and not String

    for(var str : strA){ //type of str is String

        var p = str; //type of p is String because type
of str is String

    }
}
```

```

switch(strA[0]){
    case "a":
        var m = new Object();//type of m is Object
    }
}

```

Observe that all of the usages of **var** are scoped locally. Even the variable declared within the static block is local to that block. Another important point illustrated in the above code is that the type of **obj2** will be inferred as **Object** and not **String**. This is because the type of the source variable **obj** is **Object**. The compiler has no knowledge of the type of the actual object that this variable will point to at run time. Therefore, it only goes by the declared type of **obj** to infer the type of **obj2**.

Here are a few examples of invalid usages that you will be required to identify in the exam:

```

public class LVTITest2 {
    var static value1 = 10; //can't use LVTI for class members
}

```

var value2 = 10; //can't use LVTI for instance member.

public static void main(var args)//can't use LVTI for method parameters

```
{
    var p;//can't use LVTI for uninitialized variable
}

```

var n = null;//invalid because type of null can't be determined

```

    var doubleArray = {1, 2}; //type must be specified
in array initializer if using LVTI for variable

var[] ia = new int[]{1 , 2}; //can't apply [] to va
r because var is not a type

}

public static var getValue(){//can't use LVTI for dec
laring return type of a method

    return "hello";
}
}

```

Observe that in the above code there is no way for the compiler to infer the type of `p` because there is no source variable or source value that is being assigned to `p` in the same statement. In other words, the context does not have enough information for the compiler to determine the type of `p`. Similarly, the compiler cannot figure out the type of `doubleArray` because the type of the value given is ambiguous. `{1, 2}` could be interpreted as an int array or a byte array also. In fact, to avoid confusion, Java prohibits type inference if the type of the array is not specified explicitly in the array initializer.

The case of instance field `value` in the above code is interesting. It is possible to infer the type of the variable from the value. However, Java prohibits using type inference for class and instance members because such members are part of the API of a class and are therefore, used by other classes. Inferring the type instead of explicitly stating the type makes the type of the field dependent on the value that is being assigned to it. If you change this value later on, the type of the variable may change and that may adversely impact other classes.

LVTI in Lambda Expressions [↳](#)

Java 11 has allowed the usage of LVTI in lambda expressions. I will discuss it in the Lambda Expressions

Typically, the exam has two kinds of questions on this topic.

In the first kind, you need to determine whether a particular usage of var declaration is valid or not. This is easy because any use of var keyword outside of a local scope (i.e. methods, constructors, or initializer blocks), is invalid. Even within a local scope, it is valid only if there is a value with a known type that is being assigned to the target variable.

In the second kind, you need to determine the type of the variable when you declare it with the var keyword. This is also simple because the type of the variable will be exactly the same as the declared type of the source variable that is being assigned to the variable. In case you are directly assigning a value to the variable, the type of the variable will be the same as the type of the value.

When should var be used? [↑](#)

LVTI is a powerful shortcut. But you should remember that that is exactly what it is. A shortcut. It is not a keyword or even a reserved word (so, this is actually a valid line of code: `var var = 10;`). When the compiler sees `var`, it simply converts the statement to a full blown declaration. As with all shortcuts, care must be taken to ensure that its usage doesn't affect the readability of the code. Consider the following line of code appearing in a method:

```
var x = getValue();
```

Compiler can figure out the type of `x` and will accept the code happily, but can you tell the type of `x` just by looking at the above code? You can't do that unless you look at the method declaration. In fact, every time someone goes through the method, they will have to check the `getValue()` method declaration to be sure of the type of `x`. Obviously, saving a few keystrokes has reduced readability tremendously here. Now, consider the following lines of code:

```
var mapOfStateCapitals = new HashMap<String, String>()
;
```

```
var state_listOfTownsMap = new HashMap<String, List<String>>();
```

The usage of `var` in the above code actually improves readability by reducing clutter.

4.6 Explain objects' lifecycles

4.6.1 Life cycle of an Object [🔗](#)

You know that objects are always created in **heap space**. The JVM allocates space in the heap to store the values of the instance variables of an object. Since every class ultimately extends from `java.lang.Object`, even if a class does not define any instance variable of its own, it will inherit the ones defined in the Object class. Thus, every object will take some space in the heap. Since heap space is not unlimited, only a limited number of objects can be stored in the heap. Thus, it is possible to run out of heap space if a program keeps creating objects. We must, therefore, have some way of getting rid of objects that we don't need anymore, right? Well, that is exactly what some languages such as C++ provide. C++ lets you create as well as delete objects. It lets you **allocate** and **deallocate** memory as you please. In other words, it makes the programmer "manage" the heap space.

In Java, on the other hand, the heap space is managed entirely by the JVM. There is no way for a programmer to directly manipulate the contents of this space. The only thing a programmer can do to indirectly affect this space is to create an object. Java does not even let you "delete" an object. There is no way to "deallocate" the memory consumed by objects either. The question that one may ask here is how in the world then can a program function if the memory runs out because of objects that are not needed?

Recall our discussion about object references. An object can be accessed only through its reference. If you have a reference to an object, you can read from or write to its fields, or invoke methods. You can keep this reference in a variable. You can also keep multiple copies of the same reference. You can pass it on to

other methods as well. You can think of it like a fish hooked on at the end of a fishing line. As long as you hold on to the fishing rod, you can get hold of the fish. You lose the rod, you lose the fish. Similarly, if you lose all the references to an object, there is no way to get to that object any more. The JVM uses this fact to manage the heap space.

The JVM keeps track of all the references to an object and as soon as it realizes that there are no references to that object, it concludes that this object is not required anymore and is basically "**garbage**". It then makes arrangements to reclaim the space occupied by the object. This arrangement for reclaiming the space occupied by such objects is aptly called "**garbage collection**". This essentially is what automatic memory management is, because, as you can see, the programmer does not have to deal with managing the memory. The programmer focuses only on creating objects and using them as and when required. The JVM cleans up the memory held by objects automatically in the background.

With the above discussion in mind, it should be easy to visualize the **life cycle** of an object. An object comes **alive** when it is created.

*An object can be created in three ways - using the **new keyword**, through **deserialization**, and by **cloning**. Deserialization and cloning are not on the OCP Java 11 Part 1 exam.*

It remains alive as long as it is being referenced from an active part of a program. The object is dead or inaccessible once there are no references pointing to it. An object no longer exists, once the JVM destroys, i.e., reclaims the memory consumed by the object during garbage collection.

4.6.2 Garbage Collection

Garbage Collection

Now that we have established what "garbage" is, it is easy to understand what

garbage collection entails. Garbage collection is an activity performed periodically by the JVM to reclaim the memory occupied by objects that are no longer in use. Let us dig deeper into each part of the previous sentence:

1. Activity performed periodically - The JVM performs garbage collection at regular intervals so that memory is made available to the program before a request to create an object fails for want of memory. It is not possible for a program to control the periodicity of this activity. It is not possible for a programmer to even make the JVM perform this activity instantly. There is a method named `gc` in `java.lang.System` class that lets a programmer request the JVM to perform garbage collection. You can call `System.gc()` any time you believe it is appropriate to clean up the memory but this is just a request. There is no guarantee that JVM will actually perform garbage collection after invocation of this method. A JVM may provide options to customize the behavior of its garbage collection process through command line arguments. Although a discussion about these arguments will be beyond the scope of the exam, I suggest you check them out in your spare time because garbage collection is a favorite interview topic of technical managers.

2. Reclaim the memory - Reclaiming the memory means that the memory occupied by an object is now marked as free to store new objects. After reclaiming the memory from multiple objects, the JVM may even reorganize the heap space by moving the objects around and creating large chunks of free memory. This process is pretty much like the defragmentation of a hard-drive but within the program's RAM.

3. Objects that are no longer in use - I mentioned earlier that if an object is not referenced from any active part of a program, the JVM concludes that the object is no longer in use. While this statement is true, it is a bit more complicated than it looks. Let us start with the following code:

```
public class TestClass{
    public static void main(String[] args){
        Object o1 = new Object(); //1
        o1 = new Object(); //2
```

```
    }  
}
```

An object is created at line marked //1 and its reference is assigned to variable **o1**. At this point there is only one reference variable that is pointing to this object. On the next line, another object is created and its reference is assigned to the same variable **o1**. Observe that the value held by **o1** has been overwritten by the new value. Thus, the object that was created at //1 is not being referenced by any variable at all after the line marked //2. There is no way we can access this object now because we have lost the only reference that we had of this object. The JVM is aware of this fact and will mark this object for garbage collection.

The object created at line marked //2 is being referenced by a variable and can be accessed through this variable. It is, therefore, not eligible to be garbage collected. Well, at least until the main method ends, after which there will be no reference pointing to this object either, and it will also be eligible to be garbage collected.

The above code illustrates how an object may be left without any reference variable pointing to it. Let us take it up a notch:

```
public class TestClass{  
    Object instanceVar;  
    public TestClass(Object methodParam){  
        instanceVar = methodParam;  
    }  
    public static void main(String[] args){  
        Object tempVar = new Object(); //1  
  
        TestClass tc = new TestClass(tempVar); //2  
  
        tempVar = new Object(); //3
```

```
        tc.instanceVar = null; //4  
  
    }  
}
```

An object of class **Object** is created at //1 and its reference is assigned to local variable named **tempVar**. An object of class **TestClass** is created at //2 and value of **tempVar** is passed to **TestClass**'s constructor through method parameter named **methodParam**. **TestClass**'s constructor copies this reference to an instance variable named **instanceVar**. Thus, after execution of line //2, the object created at //1 is referred to by two reference variables - **tempVar** and **instanceVar**.

Now, at //3, a new object is created and its reference is assigned to the local variable **tempVar**. Thus, **tempVar** stops pointing to the object it was pointing to earlier and starts pointing to this new object. But observe that the instance variable **instanceVar** is still pointing to the object created at //1.

At //4, we make the **instanceVar** lose its value by assigning it **null**. Therefore, after this line, the object that was created at //1 has no reference pointing to it. There is no way to access this object after this line and thus, this object is eligible to be garbage collected.

In the above two examples, I showed you how an object can be deemed no longer in use when there are no reference variables pointing to it. Indeed, if there are no variables pointing to an object, it is not possible to access that object. But can there be a situation where there is a variable pointing to an object but that object is still eligible for garbage collection? Let us modify the above code a bit:

```
public class TestClass{  
    Object instanceVar;  
    public TestClass(Object methodParam){  
        instanceVar = methodParam;  
    }
```

```
public static void main(String[] args){  
    Object tempVar = new Object(); //1  
  
    TestClass tc = new TestClass(tempVar); //2  
  
    tempVar = new Object(); //3  
  
    tc = null; //4  
  
}  
}
```

The only change I have made in this code is in the line marked //4. Instead of setting `tc.instanceVar` to `null`, I have set `tc` to `null`. Thus, the variable `tc` does not point to the `TestClass` object after this line. In fact, there is no variable that is pointing to the `TestClass` object created at //2. Thus, even though the instance variable `instanceVar` of this `TestClass` object still points to the object created at //1, there is no way to access that object because the only way to access that object was through the `TestClass` instance, which itself is not accessible anymore. Thus, both - the `TestClass` instance created at //2 and the object created at //1 - are eligible to be garbage collected.

In other words, not only the objects that have no reference to them are eligible for garbage collection, but the objects that are referenced only by objects that are themselves eligible to be garbage collected, are also eligible to be garbage collected. In the above example, there was a chain of just two objects (the `TestClass` instance and the `Object` instance) that became eligible for garbage collection but there could be any number of such interconnected objects that become eligible for garbage collected if none of the objects of that chain can be referenced from any **active part of a program**. This graph of interconnected

objects is known as an "**island**" of isolation and is considered garbage because none of the objects of that graph are reachable from an active part of a program even though they are reachable through each other.

Let me now explain what is meant by the cryptic looking phrase "**active part of a program**". Note that it is only when the statements written in a program are executed that objects are created. A program code may contain several statements that create objects using the new keyword but if those statements are not executed, obviously, no object will be created. Indeed, the code that we write is merely a set of instructions to the JVM. Nothing will actually happen if those instructions are not executed. In Java, execution of the code is done through "threads". When you run a program through the command line, a thread called "main thread" is created automatically and this thread starts executing the statements written in the main method. The statements may be simple statements such as `i = 10;` or they could be method calls, in which case the thread will execute the statements written inside that method first before moving on to the next statement in the main method. Java also allows you create your own threads and give them separate sets of instructions to execute. All such code that falls under the scope of the main thread and the threads created by the programmer is nothing but the active part of the program. When a thread dies, that is, when it is done executing all the instructions that fall under its scope (for example, the main thread dies when it reaches the end of the main method), any object that was created by this thread will be eligible for garbage collection unless the reference of that object is still held on to by some other live thread.

For the purpose of the OCP Part 1 exam, you do not have to worry about threads or the impact of threads on garbage collection. The exam merely scratches the surface of this topic. The exam only expects that you understand the meaning of garbage collection and that an object can be garbage collected when there are no references pointing to that object. You should be able to trace the reference variables pointing to an object and identify the point at which that object loses all its references.

However, a deep understanding of Garbage Collection is very critical for a Java programmer and that is why it

is a favorite topic of discussion in technical interviews. I suggest you read about the following terms if you want to ace a Java technical interview - finalization, finalize method, object resurrection, types of references, algorithms used to identify garbage, customizing the behavior of garbage collector through command line options, and object generations.

4.6.3 Garbage Collection for the exam

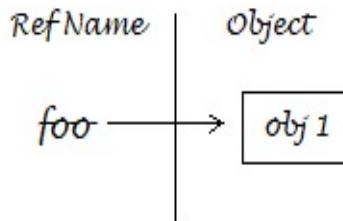
Typically, there are two types of questions that you will encounter in the exam: find out the line number after which an object becomes eligible for garbage collection; and find out how many objects will be eligible for garbage collection right after a particular line number. For both kinds you need to keep track of the references and the objects these references are pointing to at each line. It is possible to do all this in your head but the questions are designed to make you lose track of the objects and the references. Therefore, it is best to make use of a pen and paper and draw whatever is going on in your head. Let me show you how. Consider the following code where you are expected to find out the line after which the object created at line 3 is eligible for garbage collection.

```
1: public class TestClass {  
2:     public static void main(String[] args){  
3:         Object foo = new Object();  
4:         Object bar = foo;  
5:         foo = new Object();  
6:         Object baz = bar;  
7:         foo = null;  
8:         bar = null;  
9:         baz = new Object();  
10:    }  
11:}
```

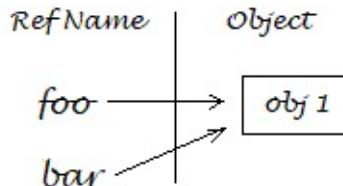
To get to the answer, I will draw a diagram to show the state of affairs after each line of code. On the left-hand side, I will write the reference variable name and on the right, I will draw a box to show the existence of an object on the heap. I

will also write a number in the box to distinguish one object from another.

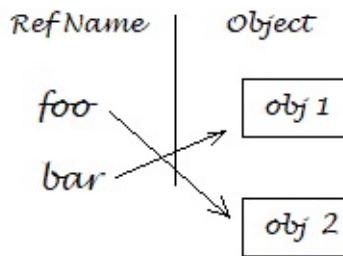
Step 1: At line 3, a new object is created and is assigned to the reference variable **foo**.



Step 2: At line 4, a new variable named **bar** is defined and is set to the same value as **foo**. Thus, **foo** and **bar** now point to the same object.

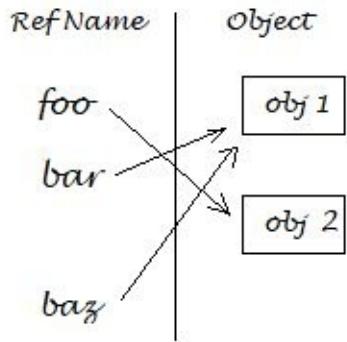


Step 3: At line 5, a new object is created and **foo** is made to point to this new object.

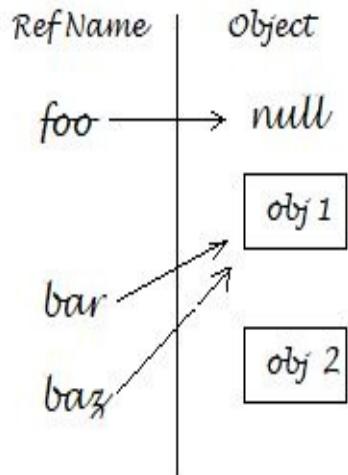


Step 4: At line 6, a new variable named **baz** is defined and is assigned the value held by **bar**. In other words, **baz** now points to whatever **bar** is pointing to, i.e., obj 1.

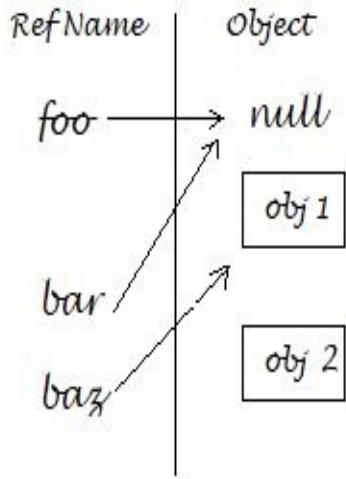
After line 6:



Step 5: At line 7, **foo** is set to **null**. In other words, **foo** is not pointing to the object it was pointing to earlier. Observe that **obj 2** is not being pointed to by any reference variable after the execution of line 7. Therefore, this object will be eligible for garbage collection after this line. However, this is not the object you are interested in. So, let's keep executing the statements.

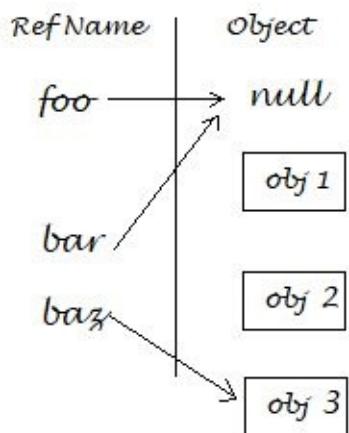


Step 6: At line 8, **bar** is also being set to **null**.



Step 7: At line 9, a new object is created and assigned to **baz**. Thus, **bar** stops pointing to obj 1 and starts pointing to obj 3 after the execution of this line.

After line:9



Observe that nobody is pointing to obj 1 now. This is the object that was created at line 3, and since it is not being referenced by any variable, it is eligible for garbage collection after line 9.

Now, let us look at the same code from another perspective. What if you are asked about the number of objects that are eligible for garbage collection after, say, line 8? It is easy to figure that out by looking at the above figures. Status of objects after line 8 shows that obj 2 is the only object that is not being referred to by any variable. Thus, it is the only object that is eligible for garbage collection after line 8. What about after line 9? Two (obj 1 and obj 2). At the end of the

method (i.e.line 10)? All three objects.

You should solve a few mock questions using this approach. With practice, you will be able to do it in your head and will not need to draw the diagrams on paper.

Garbage collection of Strings [🔗](#)

While going through mock exams or through other preparation material, you may encounter code snippets that show multiple String objects getting created. Something like this:

```
String str = "hello";
for(int i=0; i<5; i++){
    str = str + i;
}
```

The above creates one String object containing "**hello**" at the beginning and then two more in each iteration of the loop - a String containing the int value of **i** and the concatenated String. Thus, overall, the above code creates $1 + 2 \times 5 = 11$ Strings. However, if you are asked how many String will be eligible to be garbage collected, the answer is not that easy. The Java Language Specification mentions in Section 15.8.1 that the non-string operand of the + operator is converted to a String at runtime but it does not clearly tell whether this String goes to the String pool (in which case it will not be garbage collected) or not.

Let me show you another piece of code:

```
String s = "hello";
int k = 100;
s = s +"123"+k;
```

In this case, JLS section 15.8.1 clearly says that a compiler may avoid creating multiple strings altogether by making use of a StringBuilder. Thus, it is not possible to tell how many Strings will be created and how many will be eligible to be garbage collected.

Don't worry, you will not get questions in the exam about garbage collection of

Strings. I have talked about it here only to make you aware of the issue. Strings muddle the topic of garbage collection so much so that it is a bad idea to use them while explaining garbage collection. You need not spend anymore time on this topic.

Prior to Java 7, interned strings were never garbage collected even if there were no references to a particular string. Thus, it was possible for the string pool to run out of memory. Typically, programs create a lot of strings through concatenation while writing to log files. It was not too uncommon for such programs to start getting `OutOfMemoryError` after a while due to this issue. From Java 7 onwards, strings in the string pool can also be garbage collected. You will not be required to answer questions on this aspect of Strings in the exam.

4.7 Exercise

1. Define a reference type named `Bird`. Define an instance method named `fly` in `Bird`. Define a few instance as well as static variables of type `int`, `float`, `double`, `boolean`, and `String` in `Bird`.
2. Create a `TestClass` that has a static variable of type `Bird`. Initialize this variable with a valid `Bird` object. Print out the default values of static and instance variables of `Bird` from the `main` method of `TestClass`. Also print out the static variable of `TestClass` from `main`. Observe the output.
3. Create and initialize one more instance variable of type `Bird` in `TestClass`. Assign values to the members of the `Bird` instance pointed to by this instance variable in `TestClass`'s `main`. Assign values to the members of first `Bird` using the second `Bird`. Print the values of the members of both the `Bird` objects.
4. Write code in `fly` method to print out the values of all members of `Bird`. Alter `main` method of `TestClass` to invoke `fly` on both the instance of

`Bird` . Observe the values printed for static variables of `Bird` .

5. Add an instance variable of type `Bird` in `Bird` . Initialize this variable on the same line using "new `Bird` ()" syntax. Instantiate a `Bird` object in `TestClass` 's `main` and execute it. Observe the output.
6. Remove the initialization part of the variable that you added to `Bird` in previous exercise. Initialize it with a new `Bird` object separately from `TestClass` 's `main` . Identify how many `Bird` objects will be garbage collected when the `main` method ends.
7. Add a parameter of type `Float` to `Bird` 's `fly` method. Return an `int` value from `fly` by casting the method parameter to `int` . Invoke `fly` multiple times from `TestClass` 's `main` by passing a `float` literal, a `Float` object, a `double` literal, an `int` , an `Int` eger, and a `String` containing a `float` value. Observe which calls compile.
8. Assign the return value of `fly` to an `int` variable, a `float` variable, a `String` variable, and `boolean` variable. Observe which assignments compile. Try the same assignments with an explicit cast. Print these variables out and observe the output.

Chapter 5 Working with String APIs

- Create and manipulate Strings
- Manipulate data using the `StringBuilder` class and its methods

5.1 Create and manipulate Strings

5.1.1 What is a "string"? [↳](#)

In Java, a "string" is an object of class `java.lang.String`. It represents a series of characters. Strings such as "`1234`" or "`hello`" are really just objects of this class.

`String` is a **final** class, which means it cannot be extended. It extends `Object` and implements `java.lang.CharSequence`. If you are not sure of what **final** and **implements** mean, don't worry, I will cover these concepts while talking about inheritance and interfaces respectively.

In the Java world, `String` objects are usually just called "**strings**". Although a string is just like any other regular object, it is such a fundamental object that Java provides special treatment to strings in terms of how they are created, how they are managed, and how they are used. Let's go over these aspects now.

5.1.2 Creating Strings [↳](#)

Creating strings through constructors [↳](#)

The `String` class has several constructors but for the purpose of the exam, you only need to be aware of the following:

1. **String()** - The no-args constructor creates an empty String.
2. **String(String str), String(StringBuilder sb)** - Create a new String by copying the sequence of characters currently contained in the passed String or StringBuilder objects.
3. **String(byte[] bytes)** - Creates a new String by decoding the specified array of bytes using the platform's default charset.
4. **String(char[] value)** - Creates a new String so that it represents the sequence of characters currently contained in the character array argument.

Note that a string is composed of an array of **char** s. But that does not mean a string is the same as a **char** array. Therefore, you cannot apply the array indexing operator on a string. Thus, something like **char c = str[0];**, where **str** is a **String**, will not compile.

Creating strings through concatenation [↳](#)

The second common way of creating strings is by using the concatenation (, i.e., +) operator:

```
String s1 = "hello ";
String s12 = s1 + " world"; //produces "hello world"
```

The + operator is overloaded in such a way that if either one of its two operands is a string, it converts the other operand to a string and produces a new string by joining the two. There is no restriction on the type of operands as long as one of them is a string.

The way + operator converts the non-string operand to a string is important:

1. If the non-string operand is a reference variable, the **toString()** method is invoked on that reference to get a string representation of that object.
2. If the non-string operand is a primitive variable or a primitive literal value, a wrapper object of the same type is created using the primitive value and

then a string representation is obtained by invoking `toString()` on the wrapper object.

3. If the one of the operands is a `null` literal or a null reference variable, the string "`null`" is used instead of invoking any method on it.

The following examples should make this clear:

```
String s1 = "hello ";
String s11 = s1 + 1; //produces "hello 1"
```

```
String s12 = 1 + " hello"; //produces "1 hello"
```

```
String s2 = "" + true; //produces "true";
```

```
double d = 0.0;
String s3 = "-" +d +"-"; //produces "-0.0-"
```

```
Object o = null;
String s4 = "hello "+o; //produces "hello null". No NullPointerException here.
```

Just like a mathematical expression involving the `+` operator, string concatenation is also evaluated from left to right. Therefore, while evaluating the expression `"1"+2+3`, `"1"+2` is evaluated first to produce `"12"` and then `"12"+3` is evaluated to produce `"123"`. On the other hand, the expression `1 + 2 +"3"` produces `"33"`. Since neither of the operands to `+` in the expression `1 + 2` is a `String`, it will be evaluated as a mathematical expression and will therefore, produce integer `3 . 3 + "3"` will then be

evaluated as "33".

Remember that to elicit the overloaded behavior of the + operator, at least one of its operands must be a **String**. That is why, the following statements will not compile:

String x = true + 1; //Will not compile. First operand is boolean and second is int.

Object obj = "string"; //OK

String y = obj + obj; //Will not compile. Even though obj points to a String at runtime, as far as the compiler is concerned, obj is an Object and not a String.

Since the **toString** method is defined in the **Object** class, every class in Java inherits it. Ideally, you should override this method in your class but if you do not, the implementation provided by the **Object** class is used. Here is an example that shows the benefit of overriding **toString** in a class:

```
class Account{
    String acctNo;
    Account(String acctNo){
        this.acctNo = acctNo;
    }
    //overriding toString.
    //must be public because it is public in Object
```

```

public String toString(){
    return "Account["+acctNo+"]";
}
}

public class TestClass{
    public static void main(String[] args){
        Account a = new Account("A1234");
        String s = "Printing account - "+a;
        System.out.println(s);
    }
}

```

The above code produces the following output with and without overriding **toString** :

Printing account - Account[A1234]

and

Printing account - Account@72bfaced

Observe that when compared to **Object**'s **toString**, **Account**'s **toString** generates a meaningful string. Since the **Object** class has no idea about what a class represents, it just returns a generic string consisting of the name of the class of the object, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. Don't worry, you will not be asked to predict this value in the exam. Just don't get scared if you see such a value in the exam.

On a side note, the **print/println** methods that we have often used also behave just like the + operator with respect to generating a string representation of the object that is passed to them. For example, when you call **System.out.print(acct);** where acct refers to an **Account** object, the print method invokes **toString** on that **Account** object and prints the returned string.

The `+=` operator

In the chapter on Operators, we will see that `+=` is a compound operator. It applies the `+` operator on the two operands and then assigns the result back to the variable on the left side. If the type of the variable on the left is `String`, it performs a string concatenation. Here is an example:

```
String s = "1";
s += 2; //expanded to s = s + 2;
```

```
System.out.println(s); //prints "12"
```

The type of the variable on the left-hand side must be `String` or something that can refer to a `String` (there are only two such types really - `CharSequence` and `Object`), otherwise the expression will not compile:

```
int x = 1;
x += "2";
```

The above code will not compile because you can't assign an object of type `String` to a variable of type `int`. The type of the right operand can be anything because if the operand is not a string, it will be converted to a string as per the rules discussed above.

Now, can you tell what the following code will print?

```
Object m = 1;
m += "2";
System.out.println(m);
```

It will compile fine and print `"12"`. First, `1` will be boxed into an `Integer` object, which will be assigned to `m`. This assignment is valid because an `Integer` `is-a` `Object`. Next, the expression `m += "2"` will be expanded

to `m = m + "2"`. Since one of the operands of `+` in this expression is a string, a string concatenation will be performed, which will produce the string `"12"`. This string will be assigned to `m`. The assignment is also valid because a `String` is an `Object`

Okay, how about this?

```
Object m = "Hello ";
m += 1;
System.out.println(m);
```

It will fail to compile because as far as the compiler is concerned, type of `m` is `Object` and type of `1` is `int`. Therefore, when `m+=1` is expanded to `m = m + 1`, neither of the operands of `+` is a String!

5.1.3 String interning [↳](#)

Since strings are objects and since all objects in Java are always stored only in the **heap space**, all strings are stored in the heap space. However, Java keeps strings created without using the new keyword in a special area of the heap space, which is called "**string pool**". Java keeps the strings created using the new keyword in the regular heap space.

String interning [↳](#)

The purpose of the string pool is to maintain a set of unique strings. Any time you create a new string without using the new keyword, Java checks whether the same string already exists in the string pool. If it does, Java returns a reference to the same String object and if it does not, Java creates a new String object in the string pool and returns its reference. So, for example, if you use the string `"hello"` twice in your code as shown below, you will get a reference to the same string. We can actually test this theory out by comparing two different reference variables using the `==` operator as shown in the following code:

```
String str1 = "hello";
```

```
String str2 = "hello";
System.out.println(str1 == str2); //prints true
```

```
String str3 = new String("hello");
String str4 = new String("hello");
System.out.println(str1 == str3); //prints false
```

```
System.out.println(str3 == str4); //prints false
```

We will look at the `==` operator in detail later but for now, it simply checks whether two references point to the same object or not and returns `true` if they do. In the above code, `str2` gets the reference to the same String object which was created earlier. However, `str3` and `str4` get references to two entirely different String objects. That is why `str1 == str2` returns `true` but `str1 == str3` and `str3 == str4` return `false`.

In fact, when you do `new String("hello")`, two String objects are created instead of just one if this is the first time the string "`hello`" is used anywhere in program - one in the string pool because of the use of a quoted string, and one in the regular heap space because of the use of new keyword.

String pooling is Java's way of saving program memory by avoiding creation of multiple String objects containing the same value. It is possible to get a string from the string pool for a string created using the new keyword by using String's `intern` method. It is called "**interning**" of string objects. For example,

```
String str1 = "hello";
String str2 = new String("hello");
String str3 = str2.intern(); //get an interned string
object for str2
```

```
System.out.println(str1 == str2); //prints false
```

```
System.out.println(str1 == str3); //prints true
```

5.1.4 String immutability

Strings are immutable. It is impossible to change the contents of a string once you have created it. Let me show you some code that looks like it is altering a string:

```
String s1 = "12";
s1 = s1+"34";
System.out.println(s1); //prints 1234
```

The output of the above code indicates that the string pointed to by `s1` has been changed from `"12"` to `"1234"`. However, in reality, the original string that `s1` was pointing to remains as it is. A new string containing `"1234"` is created instead and its address is assigned to `s1`. So, after the last line of the above code, the JVM would have created three different strings - `"12"`, `"34"`, and `"1234"`.

There are several methods in the `String` class that may make you believe that they change a string but just remember that a string cannot be mutated. Ever. Here are some examples:

```
String s1 = "ab";
s1.concat("cd");
System.out.println(s1); //prints "ab"
```

```
s1.toUpperCase();
System.out.println(s1); //prints "ab"
```

In the above code, `s1.concat("cd")` does create a new string containing `"abcd"` but this new string is not assigned to `s1`. Therefore, the first `println` statement prints `"ab"` instead of `"abcd"`. The same thing happens with `toUpperCase()`. It does produce a new string containing `"AB"` but since this string is not assigned to `s1`, the second `println` statement prints `"ab"` instead of `"AB"`. Note that the newly created strings `"abcd"` and `"AB"` will remain in the string pool. The JVM will use them whenever it needs to create a string containing the same characters. But as of now, we don't have any reference that points to these strings.

Mutability [↳](#)

Generally, **mutability** refers to the properties of an object of a class. Thus, an immutable class implies that the instance variables of an object cannot be changed once the instance is created. This is achieved by making instance variables private and having only getter methods for reading their values. Mutability is an important concept that has many aspects. However, a detailed discussion on it is not in the scope for OCP Java 11 Part 1 exam. You should, however, be aware that it is included in the Part 2 exam and is also a favorite topic of discussion in technical interviews.

5.1.5 Manipulating Strings [↳](#)

The **String** class contains several methods that help you manipulate strings. To understand how these methods work, you may think of a string as an object containing an array of chars internally. These methods simply work upon that array. Since array indexing in Java starts with 0, any method that deals with the locations or positions of characters in a string, also uses the same indexing logic. Furthermore, any method that attempts to access an index that is beyond the range of this array throws **IndexOutOfBoundsException**.

Here are the methods that belong to this category with their brief JavaDoc

descriptions:

1. `int length()` - Returns the length of this string.

For example, `System.out.println("0123".length());` prints **4**. Observe that the index of the last character is always one less than the length.

2. `char charAt(int index)` - Returns the char value at the specified index. Throws `IndexOutOfBoundsException` if the index argument is negative or is not less than the length of this string.

For example, `System.out.println("0123".charAt(3));` prints **3**.

3. `int indexOf(int ch)` - Returns the index within this string of the first occurrence of the specified character. Returns -1 if the character is not found.

Examples:

```
System.out.println("0123".indexOf('2')); //prints 2  
System.out.println("0123".indexOf('5')); //prints  
-1
```

A design philosophy followed by the Java standard library regarding methods that deal with the starting index and the ending index is that the starting index is always inclusive while the ending index is always exclusive. `String`'s `substring` methods works accordingly:

1. `String substring(int beginIndex, int endIndex)` - Returns a new string that is a substring of this string.

Examples:

```
System.out.println("123456".substring(2, 4));  
//prints 34.
```

Observe that the character at index 4 is not included in the resulting

substring.

```
System.out.println("123456".substring(2, 6));  
//prints 3456  
System.out.println("123456".substring(2, 7));  
//throws StringIndexOutOfBoundsException
```

2. **String substring(int beginIndex)** - This method works just like the other substring method except that it returns all the characters from beginIndex (i.e. including the character at the beginindex) to the end of the string (including the last character).

Examples:

```
System.out.println("123456".substring(2));  
//prints 3456.  
System.out.println("123456".substring(7)); //throws  
StringIndexOutOfBoundsException .
```

The rule about not including the element at the ending index is followed not just by the methods of the String class but also by methods of other classes that have a concept of element positions such as java.util.ArrayList.

The following methods return a new **String** object with the required changes:

1. **String concat(String str)** - Concatenates the specified string to the end of this string.

Example - `System.out.println("1234".concat("abcd"));
//prints 1234abcd`

2. **String toLowerCase()/toUpperCase()** - Converts all of the characters in this String to lower/upper case.

Example - `System.out.println("ab".toUpperCase());
//prints AB`

3. **String replace(char oldChar, char newChar)** - Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

Example: `System.out.println("ababa".replace('a', 'c'));` //prints *cbc*

4. **String strip(), stripLeading(), stripTrailing(), trim()** - Returns a copy of the string, with leading and/or trailing white space omitted.

Example:

`System.out.println(" 123 ".strip());` //prints "123" (without the quotes)

`System.out.println(" 123 ".stripLeading());` //prints "123" (without the quotes)

`System.out.println(" 123 ".stripTrailing());` //prints "123" (without the quotes)

`System.out.println(" 123 ".trim());` //prints "123" (without the quotes)

One interesting thing about the methods detailed above is that they return the same string if there is no change in the string as a result of the operation. Thus, the following code prints true:

```
String s1 = "aaa"; //there is no b in this string
```

```
String s2 = s1.replace('b', 'c'); //nothing is replaced
```

```
System.out.println(s1 == s2); //prints true because the same string was returned by replace
```

```
System.out.println("1234".strip() == "1234"); //prints true for the same reason
```

It is very common to invoke these methods in the same line of code by chaining them together:

```
String str = " hello ";
str = str.concat("world ").trim().concat("!").toUpperCase();
System.out.println(str);
```

The above code prints **HELLO WORLD!**. Note that such chaining is possible only because these methods return a string. You will see a similar chaining of methods in **StringBuilder /StringBuffer** classes as well.

Difference between `trim()` and `strip()` ↴

String class has had the `trim` method since the beginning while the `strip` method was added in Java 11. Both the methods do the same thing, i.e., remove "white spaces" from the beginning and end of the String. The difference is in the way these methods determine what constitutes a white space. Without getting into too many details, you should be aware that `strip` uses a better definition of a white space than

trim and identifies more white spaces than trim . You should use strip instead of trim in new code. You may see the complete detail of difference in JavaDoc API but it is not required for the exam.

Finally, here are a few methods that let you inspect the contents of a string:

1. **boolean isBlank()** : Returns true if the string is empty or contains only white spaces, otherwise false.
2. **boolean isEmpty()** : Returns true if, and only if, length() is 0.
3. **boolean startsWith(String prefix)** : Returns true if this string starts with the specified prefix.
4. **boolean endsWith(String suffix)** : Returns true if this string ends with the specified suffix.
5. **boolean contains(CharSequence s)** : Returns true if and only if this string contains the specified sequence of char values.
6. **boolean equals(Object anObject)** : Returns true if the contents of this string and the passed string are exactly same. Observe that the type of the parameter is **Object** . That's because this method is actually defined in the Object class and the String class overrides this method. So, you can pass any object to this method, but if that object is not a string, it will return false.
7. **boolean equalsIgnoreCase(String anotherString)** : Compares this String to another String, ignoring case considerations.

The above methods are fairly self-explanatory and work as one would expect after looking at their names, so, I am not going to talk about them in detail here but I suggest you take a look at their JavaDoc descriptions and write a few test

programs to try them out. You will not get any trick questions on these methods in the exam.

5.1.6 Comparing strings [↳](#)

There are two ways you can compare strings - using the `==` operator and using the `equals` method of `String` class.

Comparison using `==` operator [↳](#)

As I discussed earlier, the `==` operator, when applied to references, checks whether two references point to the same object or not. You can, therefore, use this operator on string references to check whether they point to the same `String` object or not. If the two references point to the same string, they are obviously "equal". The following code, for example, prints `true` for this reason:

```
String str = "hello";
System.out.println(str == "hello");//prints true
```

Since creating strings in Java is quite easy, it is very tempting to use the `==` operator for testing their equality. However, it is very dangerous to do so as illustrated by the following code. All the `checkCode` method below wants to do is to check whether the string passed to it matches with the string referenced by the static variable named `code`.

```
public class TestClass{
    static String code = "1234";
    public static void checkCode(String str){
        System.out.println(code == str);
    }

    public static void main(String[] args){
        checkCode("1234");
        checkCode(new String("1234"));
    }
}
```

```
    }  
}
```

This code prints `true` for the first comparison and `false` for the second. Ideally, it should have printed `true` for both. The problem is that the `checkCode` method has no knowledge of how the string that was passed to it was created. As discussed earlier, when you create a string using the new keyword, an entirely new String object is created. That is why the second check returns `false`. It shows that `==` operator cannot guarantee you the right result if you want to compare the character data of two strings.

Comparison using equals method [↳](#)

`String` class has an `equals` method that compares the actual character data of two strings to determine whether they are equal or not. This method returns `true` if the data matches and `false` otherwise. Since it compares the actual characters contained in two strings, it doesn't matter whether the two string references are references to the same String object or are references to two different String objects. Thus, `new String("1234").equals("1234")` will always return true even though both are two different String objects.

The `equals` method, therefore, is a better way to compare two strings.

To fix the code shown above, just replace `System.out.println(code == str);` with `System.out.println(code.equals(str));` It is common to invoke `equals` on a String literal and pass a String variable instead of the other way round., i.e., `"1234".equals(str);` is preferred to `str.equals("1234")` because if `str` is `null`, the first style returns `false` but the second style throws a `NullPointerException`.

5.2 Manipulate data using the `StringBuilder` class and its methods

5.2.1 Why `StringBuilder` [↳](#)

`java.lang.StringBuilder` is the mutable sibling of `java.lang.String`. Both the classes directly extend `Object` and implement `CharSequence`. Both are `final` as well.

You may be wondering why we need another class to deal with strings if `String` allows us to do everything that we could possibly want to with strings! Well, besides being mutable, `StringBuilder` is quite different from `String` due to the fact that `StringBuilder` objects are treated just like objects of other regular classes by the JVM. There is no "string pool" or "interning" associated with `StringBuilder` objects. `StringBuilder` objects are garbage collected just like other objects once they go out of scope, which means they don't keep occupying memory forever. This makes `StringBuilder` objects more suitable for creating temporary strings that have no use once a method ends, such as, creating lengthy debug messages or building long xml documents. It may be hard to believe but a program can create a large number of `String` objects pretty quickly. For example, the following trivial code generates an HTML view for displaying a list of names in a browser:

```
public String showPerson(List persons){
    String html = "<h3>Persons</h3>";
    for(Object o : persons){
        Person p = (Person) o;
        html = html +p.getName()+"<br class=\"mynewline\""
    >";
    }
    return html;
}
```

The above code has the potential to wreak havoc on a program's memory. Depending on the size of the list, it will create a large number of `String` objects and all of them will sit in the memory for a long time, possibly for the entire life-time of the program. The same method, written using `StringBuilder`, is a lot more efficient:

```
public StringBuilder showPerson(List persons){
    StringBuilder html = new StringBuilder("<h3>Persons
</h3>");
    for(Object o : persons){
```

```

        Person p = (Person) o;
        html.append(p.getName()).append("<br class=\"mynewline\" >");
    }
    return html;
}

```

It creates exactly two **String** objects and exactly one **StringBuilder** object irrespective of the number of elements in the List. Furthermore, the **StringBuilder** object will be garbage collected as soon as it goes out of scope.

On the other hand, since **String** objects are interned, they are more suitable for creating short strings that are used repeatedly in a program (For example, "<br class=\"mynewline\" >" in the above code). Also, if you want to use strings in a **switch** statement, then **String** is the only option.

5.2.2 StringBuilder API [↳](#)

StringBuilder provides several constructors and methods and the exam expects you to know most, if not all, of them. Let's go over the constructors first:

1. **StringBuilder()** : Constructs a StringBuilder with no characters in it and an initial capacity of 16 characters. Here, "capacity" refers to the size of an internal array that is used to store the characters. Initially, this array is empty and is filled up as you start adding characters to the StringBuffer. The StringBuilder object automatically allocates a new array with larger size once this array is full.

The capacity of a StringBuilder is analogous to a bucket of water. It is empty at the beginning and fills up as you add water to it. Once it is full, you need to get a bigger bucket and transfer the water from the smaller bucket to the bigger one.

2. **StringBuilder(CharSequence seq)** : Constructs a StringBuilder that contains the same characters as the specified CharSequence. Recall

that `String` implements `CharSequence` as well. Thus, this constructor can be used to create a new `StringBuilder` with the same data as an existing `String` or `StringBuilder`.

3. `StringBuilder(int capacity)` : Constructs a `StringBuilder` with no characters in it and an initial capacity specified by the `capacity` argument. If you expect to build a large string, you can specify a big capacity at the beginning to avoid reallocation of the internal storage array later. For example, if you are building an HTML page in a method, you might want to create a `StringBuilder` with a large initial capacity. It is important to understand that specifying a capacity does not mean you can store only that many characters in the `StringBuilder`. Once you fill up the existing capacity, the `StringBuilder` will automatically allocate a new and larger array to store more characters.
4. `StringBuilder(String str)` : Constructs a `StringBuilder` initialized to the contents of the specified string. This constructor is actually redundant because of the `StringBuilder(CharSequence seq)` constructor. It exists only for backward compatibility with code written before JDK 1.4, which is when `CharSequence` was first introduced.

Since the whole purpose of having a `StringBuilder` is to have mutable strings, it is no wonder that it has a ton of overloaded append and insert methods. But don't be overwhelmed because all of them follow the same pattern. The append method only takes one argument. This argument can be of any type. The insert method takes two arguments - an `int` to specify the position at which you want to insert the second argument. Both the methods work as follows:

1. If you pass a `CharSequence` (which, again, implies `String` and `StringBuilder`) or a `char[]`, each character of the `CharSequence` or the char array is appended to or inserted in the existing `StringBuilder`.
2. For everything else, `String.valueOf(...)` is invoked to generate a string representation of the argument that is passed. For example, `String.valueOf(123)` returns the String "123", which is then appended to or inserted in the existing `StringBuilder`. In case of objects, `valueOf` invokes `toString()` on that object to get its string

representation.

3. If you pass a `null`, the string "`null`" is appended to or inserted in the existing `StringBuilder`. No `NullPointerException` is thrown.
4. All of the `append` and `insert` methods return a reference to the same `StringBuilder` object. This makes it easy to chain multiple operations. For example, instead of writing `sb.append(1); sb.insert(0, 2);`, you can write `sb.append(1).insert(0, 2);`

Here are a few examples of how the append methods work:

```
StringBuilder sb = new StringBuilder(100); //creating an empty StringBuilder with an initial capacity of 100 characters
```

```
sb.append(true); //converts true to string "true" and appends it to the existing string
```

```
System.out.println(sb); //prints true
```

```
sb.append(12.0); //converts 12.0 to string "12.0" and appends it to the existing string
```

```
System.out.println(sb); //prints true12.0
```

```
sb.append(new Object()); //calls toString on the object and appends the result to the existing string
```

```
System.out.println(sb); //prints true12.0java.lang.  
Object@32943380
```

And here are a couple of examples to illustrate the insert methods:

```
StringBuilder sb = new StringBuilder("01234");  
  
sb.insert(2, 'A'); //converts 'A' to string "A" and  
inserts it at index 2
```

```
System.out.println(sb); //prints 01A234
```

```
sb.insert(6, "hello"); //inserts "hello" at index 6
```

```
System.out.println(sb); //prints 01A234hello
```

In the above code, observe the location at which the string is being inserted. As always, since indexing starts with 0, the first position at which you can insert a string is 0 and the last position is the same as the length of the existing string. If your position argument is negative or greater than the length of the existing string, the insert method will throw an [StringIndexOutOfBoundsException](#).

The rest of the methods are quite straightforward and work as indicated by their names. To make them easy to remember, I have categorized them into two groups - the ones that return a self-reference (i.e. a reference to the same `StringBuilder` object on which the method is invoked), which implies they can be chained, and the ones that do not.

Methods that return a self-reference are - `reverse()`, `delete(int start, int end)`, `deleteCharAt(int index)`, and `replace(int start, int end, String replacement)`. Remember that **start index** is always **inclusive** and **end index** is always **exclusive**, so, the following code will print `0abcd34` and `0cd34`.

```
StringBuilder sb = new StringBuilder("01234");  
  
    sb.replace(1, 3, "abcd"); //replaces only the chars  
at index 1 and 2 with "abcd"
```

```
System.out.println(sb); //prints 0abcd34
```

```
sb.delete(1, 3); //deletes only the chars at index  
1 and 2
```

```
System.out.println(sb); //print 0cd34
```

Methods that cannot be chained are `int capacity()`, `char charAt(int index)`, `int length()`, `int indexOf(String str)`, `int indexOf(String str, int startIndex)`, `void setLength(int len)`, `String substring(int start)`, `String substring(int start, int end)`, and `String toString()`.

The `setLength` method is interesting. It truncates the existing string contained in the `StringBuilder` to the length passed in the argument. Thus, `StringBuilder sb = new StringBuilder("01234"); sb.setLength(2);` will truncate the contents of `sb` to `01`.

delete vs substring ↗

It is important to understand the difference between the delete and the substring methods of StringBuilder. The delete methods affect the contents of the StringBuilder while the substring methods do not. This is illustrated by the following code:

```
StringBuilder sb = new StringBuilder("01234");  
  
String str = sb.substring(0, 2);  
System.out.println(str+" "+sb);  
  
StringBuider sb2 = sb.delete(0, 2);  
System.out.println(sb2+" "+sb);
```

The above code prints 01 01234 and 234 234 .

Not important for the exam but you should be aware that prior to Java 1.5, the Java standard library only had the `java.lang.StringBuffer` class `StringBuffer` class to deal with mutable strings. This class is thread safe, which means it has a built-in protection mechanism that prevents data corruption if multiple threads try to modify its contents simultaneously. However, the Java standard library designers realized that `StringBuffer` is often used in situations where this protection is not needed. Since this protection incurs a substantial performance penalty, they added `java.lang.StringBuilder` in JDK 1.5, which provides exactly the same API as `StringBuffer` but without the thread safety features.

You may see old code that uses `StringBuffer` but unless you want to modify a string from multiple threads, you don't need to use `StringBuffer`. Code with `StringBuffer` will run a little slower than the

code that uses `StringBuilder` .

5.3 Exercise

1. Write code to determine whether the `toString` and `substring` methods of `StringBuilder` and `String` classes return an interned string or not. Confirm your results by checking the JavaDoc API descriptions of these methods.
2. Write a method that takes a `String` and returns a `String` of the same length containing the '`X`' character in all positions except the last 4 positions. The characters in the last 4 positions must be the same as in the original string. For example, if the argument is "`12345678`" , the return value should be "`XXXX5678`" .
3. Implement the same method as above but with a `StringBuilder` as the input parameter.
4. Write a method that takes a `String[]` as an argument and returns a `String` containing the concatenation of all the strings in the input array. Invoke your method with different arguments. Make sure that your code handles the cases where the argument is null, contains a few nulls, or contains only nulls. Is this a good place to make use of a `StringBuilder` ?

Chapter 6 Using Operators

- Use Java operators including the use of parenthesis to override operator precedence

6.1 Java Operators [🔗](#)

A program is nothing but an exercise in manipulating the data represented by variables and objects. You manipulate this data by writing statements and expressions with the help of operators. In that respect, operators are kind of a glue that keeps your code together. You can hardly write a statement without using any operator. Something as simple as creating an object or calling a method on an object requires the use of an operator (the new operator and the dot operator!). It is therefore, important to know what all operators does Java have and to understand how they work.

6.1.1 Overview of operators available in Java [🔗](#)

Java has a large number of operators. They can be classified based on the type of operations they perform (arithmetic, relational, logical, bitwise, assignment, miscellaneous) or based on the number of operands they require (unary, binary, and ternary). They may also be classified on the basis of the type of operands on which they operate, i.e., primitives (including primitive wrappers) and objects.

While, as a Java programmer, you should be aware of all of them, for the purpose of the exam, you can ignore a few of them. The following sections provide a brief description of all the operators. The ones that are not required for the exam are noted as such.

Arithmetic Operators

Arithmetic operators are used to perform standard mathematical operations on all primitive variables except boolean. They can also be applied to **wrapper objects for numeric types** (i.e. `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`) due to auto-unboxing.

Operator(s)	Brief description and Examples
<code>+, -, *, /</code> (Binary)	Addition, subtraction, multiplication, and division . Example: <pre>int a = 10; Integer b = 100; //using primitive wrapper here</pre> <pre>int c = a + b;</pre>
<code>%</code> (Binary)	Modulus operator - returns the remainder of the division of first operand by the second one. Example: <pre>int a = 10; int b = 3; int c = a % b;</pre> <p><code>c</code> is assigned a value of <code>1</code> because when <code>10</code> is divided by <code>3</code> , the remainder is <code>1</code> .</p> <p>Here is another example:</p> <pre>Integer i = 10; Character c = 'a';</pre>

```
System.out.println((i%c)); //prints 10
```

The above example illustrates that these operators work on wrapper objects including **Character**. Don't worry, you will not be required to perform mathematical calculations involving the modulus operator in the exam. But as an exercise, you should try to find out why the above code prints **10**.

- (Unary) **Unary minus** - returns a negated value of a literal value or a variable without changing the value of the variable itself.

A unary plus may also be used on a literal or a variable but it is not really an operator because it doesn't do anything.

Examples:

Using - on a literal :

```
int a = -10; //assigns -10 to a
```

Using - on a variable:

```
int b = -a;
```

Here, **b** is assigned the negated value of **a**, i.e., $-(-10)$, i.e., **10**. **a** remains **-10**.

Using + on a variable:

```
int c = +a;
```

This is valid but will not assign **10** to **c**. It will assign **-10** to **c** because **a** is **-10**. **a** remains **-10** as well.

- ++, --**
(Unary) **Unary increment and decrement operators** - Unlike the unary minus operator, these operators can only be used on a variable and they actually change the value of the variable on which they are applied.

Also unlike the unary minus, they can be applied before (pre) as well as after (post) the variable. I will explain the difference

between pre and post later.

Examples:

```
int a = 10;  
int b = -10;
```

Post increment:

```
a++; //a is incremented from 10 to 11  
b++; //b is incremented from -10 to -9
```

Pre increment:

```
++a; //a incremented from 11 to 12  
++b; //b is incremented from -9 to -8
```

(-- works the same way)

Relational Operators

Relational operators are used to compare integral and floating point values. They are used for unboxing.

Operator(s)	Brief description
<, >, <=, >= (Binary)	Less than, greater than, less than or equal to, and greater than or equal to operators for numeric types and return a boolean value.

Example:

```
int a = 10;  
Integer b = 100; //using primitive wrapper here
```

```
boolean flag = a < b; //flag is assigned a value  
//less than the value of b.
```

`==, !=`
(Binary)

Equal to and **Not equal to** - These operators are a bit special because they work on primitive types (int, float, double, boolean as well) and reference types. When used on two primitive values, it checks whether the two values are same or not.

Example:

```
int a = 10; Integer b = 20; char ch = 'a';  
System.out.println(a == b); //comparing an int equal to 20
```

```
System.out.println(a == 10.0); //comparing an int  
// considers 10 and 10.0 as equal
```

```
System.out.println(a == ch); //comparing an int equal to 'a'
```

```
System.out.println(97 == ch); //comparing an int  
// if 'a' is indeed 97
```

```
System.out.println(a != d); //comparing an int  
// to see if they have the same value
```

```
System.out.println(a != 10); //comparing two integers
```

```
System.out.println(false != flag); //comparing a boolean  
// to see if the flag is false
```

You cannot compare a numeric value and a non-numeric value such as strings.

the reference is to a primitive wrapper, of course) or even two references to the same numeric value can never be the same as a boolean value or a reference to an object. In such nonsensical comparison, the compiler deems it to be a coding error.

```
System.out.println(10 == false); //can't compare
```

```
Object obj = new Object();
System.out.println(obj != 10); //can't compare
```

```
System.out.println(obj == true); //can't compare
```

```
Integer INT = 10;
Double D   = 10.0;
System.out.println(INT == D); //can't compare
```

When used on references, `==` and `!=` check whether the two references point to the same object in memory.

```
Object o1 = new Object();
Object o2 = o1;
boolean e = (o1 == o2); //e is assigned a value indicating whether o1 and o2 refer to the same object in memory.
```

```
o2 = new Object();
System.out.println(o1 == o2); //prints false because o1 and o2 refer to different objects
```

```
String s1 = "hello";
String s2 = "hello";
```

```
System.out.println(s1 == s2); //prints true because they are the same reference
```

Comparison of references using `==` looks straight forward but it is a bit more complex than that. There are two parts. One that deals with their usage on String references and another part that deals with other types. We will cover the first part in this chapter and the second part in the "Working with Instances" chapter.

Logical Operators

Logical operators are used to form boolean expressions using boolean variables and constants.

Operator(s)	Brief description and Example
-------------	-------------------------------

<code>&&, </code> (Binary)	Short circuiting "and" and "or". They return a boolean value.
---	--

Example:

```
boolean iAmHungry = false;  
boolean fridgeHasFood = false;  
boolean eatUp = iAmHungry && fridgeHasFood; //  
// and if there is food in the fridge
```

```
boolean tooMuchExcitement = true;  
boolean eatAnyway = eatUp || tooMuchExcitement;  
// or if there is too much excitement in the area
```

They are called **short circuiting** operators because they avoid evaluating the second operand if the value of the first part does not make any difference to the final value of the expression. For example, if the value of the first operand is "conditional" like `iAmHungry`, the value of the second operand is "conditional" like `fridgeHasFood`. It is evaluated only once. Let's see how it works with the example I gave above. You are hungry and there is food in the fridge. Now, if you are not hungry, do you need to check if there is food in the fridge to see if there is food in it or not? Of course not. Since you are not hungry, the value of the first operand is false, so the value of the second operand does not matter.

decide that you won't eat food irrespective of whether there is food or not. In the second part of the expression, i.e., the check for `fridgeHasFood` is not evaluated) if the first part, i.e., check for `iAmHungry` is false.

Similarly, if you are hungry, do you still need excitement in the air to eat? If you are hungry, you can decide right there to eat food irrespective of whether there is food in the air or not. Therefore, even here, the second part of the expression, i.e., `tooMuchExcitement` can be short circuited if the first part, i.e.,

Short circuiting behavior is helpful in cases where parts of an expression evaluate. Think of the above example again. Would you get up and look for food in the fridge if it is empty or not when you are not hungry? Nah, it is too much of a waste of time. Consider an expression such as `iAmHungry && checkFridge()`, where `checkFridge()` returns `true` or `false` depending on whether there is food in the fridge. The expression is not evaluated if `iAmHungry` is `false`. Similarly, evaluating some complex database queries, which involve looking up the database, may be too time consuming and it may be better to skip them if their value doesn't make a difference to the final value of the expression.

You need to understand this behavior very clearly because it gets expressed in professionally written code.

&, |
(Binary)

Non-Short circuiting "and" and "or" (`|` is also known as inclusive OR)

Example:

```
boolean iAmHungry = false;  
boolean fridgeHasFood = false;  
boolean eatUp = iAmHungry & fridgeHasFood; //  
// if there is food in the fridge
```

```
boolean tooMuchExcitement = true;  
boolean eatAnyway = eatUp | tooMuchExcitement;  
// or if there is too much excitement in the air
```

They are actually bitwise operators and are mostly used to operate on boolean values just like `&&` and `||`. The only difference is that they short circuit any part of an expression.

This behavior is useful in cases where parts of an expression has side effects even if their value is irrelevant to the final value of the expression. Consider the following expression - `boolean accessGranted = auther != null && logToAudit(userid);`. Now, it is possible to decide that access is not authenticated irrespective of what `logToAudit` method returns. This is to make sure every request for access is logged. Thus, you may want to invoke `logToAudit` method irrespective of whether `authenticateUser` method returns `true`. In this case is appropriate because if you use `&&` instead of `&`, `logToAudit` will not be invoked if `authenticateUser` method returns `false`.

^ **Xor aka Exclusive Or** - Just like `&` and `||`, this is also a bitwise operator. On two boolean operands, it returns `true` if and only if exactly one of them is `true`.

Example:

```
boolean a = false;
boolean b = true;
boolean c = a^b; //c is assigned the value true
```

The question of short-circuiting does not arise here because both the operators determine the result. In other words, it can never short-circuit an expression.

! **Negation** - This operator returns the compliment of given a boolean value.

Example:

```
boolean hungry = false;
```

```
boolean stuffed = !hungry; //assigns true to
```

:?

(Ternary)

Ternary - To be precise, "ternary" is not really the name of this operator since this is the only operator in Java that requires three operands, it is a "ternary" operator. This operator is kind of a short form for the if-else construct and does not have a meaningful name. It evaluates either the second or the third operand based on the value of the first operand. For example:

```
int a = 5;
String str = a == 5 ? "five" : "not five";
System.out.println( str ); //prints five
```

There are quite a few rules that govern the type of a ternary expression. We will discuss them in the next chapter along with **if/if-else**.

↑

Assignment operators are used to assign the value of an expression given on the right side of the operator to the variable on the left. The assignment operators are `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, and, `|=`. The first one, i.e., `=` is the simple assignment operator.

Operator(s)

=

(Binary)

Simple assignment - It simply copies the value on the left to the variable on the right. The value assigned is the value of the reference, it is the value of the reference (not the actual object present).

Example:

```
byte b1 = 1; //assign 1 to variable b1
```

```
Object o1 = "1234"; //assign the address of the string to o1
```

```
Object o2 = o1; //assigns the value contained in o1 to o2
```

//Thus, o2 starts pointing to the same memory

//Note that there is only one instance of the

(If you are not clear about the difference between an object and a reference, see the "Java for beginners" chapter.)

***=, /=, %=, Compound assignment** - These operators are called compound assignment operators. They take a binary operator like `+=`, `-=`, `<-=`, and then assign the result of the operation to the variable on the left. For example, `>>=`, `>>>=`, is a right shift assignment operator, these operators do not apply to boolean and reference types.

&=, ^=, |=

(Binary) Example:

```
int i1 = 2; //bit pattern of 1 is 00000001
```

```
int i2 = 3;
i2 *= i1; //assigns the value of i2*i1, i.e.,
```

```
byte b1 = 8;
b1 /= 2; //assigns 4 to b1
```

The easiest way to understand how these operators work is to expand them into their expanded forms. For example, `i2 *= i1` can be expanded to `i2 = (int) (i2 * i1)`; Similarly, `b1 /= 2` can be expanded to `b1 = (byte) (b1 / 2)`. Notice the explicit cast in the expanded form. I will explain its reasons in the next section.

The `+=` operator is overloaded to work with Strings as well. It concatenates the string on the right to the string on the left.

Example:

```
String s = "hello";
s += " world"; //creates a new String "hello world"
```

Just like the other compound assignment statements mentioned above:
`s = s + " world";`

While the primary function of an assignment operator is quite simple, there are a few additional rules:

1. They are all **right associative**, which means `a = b = c = 10;` will be evaluated separately in the "Operator precedence and evaluation of expressions" section.
2. The left-hand operand of these operators must be a variable. It can either be a primitive type or a reference to an object (for example, an array element). Thus, you cannot do something like `10 = b;` because it would mean that `b` is assigned the value `10`. This is different from `aMethodThatReturnsAnObject() = 20;` either because a method returns a reference to an object and not the value itself, or because of the implication of the fact that Java does not have "**pass-by-reference**".
3. The right-hand operand of these operators must be an expression whose type is compatible with the left-hand operand. For example, you cannot assign `boolean` expression to an `int` variable. In case of primitive type, the compatibility is checked at compile time. In case of references, it is checked at runtime. Compatibility in case of references is a bit complicated. I will talk about it in the next chapter. Compatibility in case of references is a bit complicated. I will talk about it in the next chapter.

Bitwise Operators (Not required for the exam)

Bitwise operators are used to apply logical operations on individual bits of given objects (for example, `byte` or `int`). I will not discuss them in detail because they are not required for the exam.

(If you want to understand these operators better, try applying them to various numbers. For example, you can use `Integer.toHexString()` method to print out the bit pattern of any given number.)

Operator(s)	Brief description and Examples
<code>&, , ^</code> (Binary)	Bitwise "and", "or", and "xor" Example: <code>byte b1 = 1; //bit pattern of 1 is 00000001</code>

`byte b2 = 2; //bit pattern of 2 is 00000010`

```
byte b3 = (byte) (b1 & b2); //b3 gets 0
```

```
byte b4 = (byte) (b1 | b2); //b4 gets 3
```

```
byte b5 = (byte) (b1 ^ b2); //b5 gets 3
```

(I will explain the reason for explicit casting of the results to byte later)

~
(Unary) **Bitwise complement** - It toggles, i.e., turns a 0 to 1 and a 1 to 0, including primitive numeric wrapper objects)

Example:

```
byte b1 = 1; //bit pattern of 1 is 00000001
```

```
byte b2 = (byte) ~b1; //b2 gets 11111110, which is -2
```

>>, <<
(Binary) **Bitwise signed right and left shift** - They shift the given bit pattern of places specified by the right operand while keeping the sign of the number.

Example:

```
byte b1 = -4; //bit pattern of 4 is 11111100
```

```
byte b2 = 1;
```

```
byte b3 = (byte) (b1 >> b2); //b3 gets 111111
```

```
byte b4 = (byte) (b1 << b2); //b4 gets 111110
```

Observe that when we shifted the bits of -2 to right by 1 place, the s

>>> **Bitwise unsigned right shift** - This operator works the same way as pushes in zeros from the left irrespective of the sign of the number.

Example:

```
int i = -4; //bit pattern of -4 is 11111111 1.
```

```
int i2 = 1;
int i3 = i1 >>> i2; //i3 gets 01111111 111111.
646
```

Observe that when we shifted the bits of -4 to right by 1 place, the s over. Also observe that I have used **int** instead of **byte** variables casting from affecting the result. Again, **this is not required for the byte** variables instead of **int** and compare the values of **i3** using

Operator(s)	Miscellaneous
+ (Binary)	String concatenation - + operator can also be used to concatenate two strings.
	Example:

```
String s1 = "hello";
String s2 = " world";
String s = s1 + s2; //creates a new String "hello world"
```

The above example illustrates the most straight forward use of the + versatile and can be used to join any kind of object or primitive val

. (dot) (Binary) **The dot operator** - You have definitely seen it but most likely have variable or access a member of a class or an object, you use the dot nonetheless. It is applied to a reference to access the members of the

It will always throw a **NullPointerException** if you apply it object)

instanceof (Binary) **instanceof** - Although **not required** for the exam, a thorough understanding of this operator is closely tied to polymorphism, a concept that I haven't yet talked about. **instanceof** is used to check whether an object pointed to by a reference of this operator. It returns **true** if the object pointed to by the reference given on the right and **false** otherwise.

Example:

```
Object obj = "hello"; //declared type of obj
```

```
boolean isString = (obj instanceof String); //  
//nt to a String object
```

```
boolean isNumber = (obj instanceof Number); //  
//t point to a Number object
```

() (Binary) **The cast operator** - This operator can be used on numeric values and while assigning values of a larger type to variables of a smaller type

When used on a reference, it casts the reference of one type to another

polymorphism and `instanceof` operator.

- > **The lambda operator** - This operator is used to write lambda expressions.
(Binary)
- [] **The array access operator** - This operator is used to access the elements of arrays.
(Binary) See "Java Data Types" chapter.

6.1.2 Expressions and Statements [↳](#)

Difference between an Expression and a Statement [↳](#)

Before I move on to the intricacies of various operators, let me talk a bit about **expressions** and **statements**. This will give you a good perspective on how various operators work and why they work so.

Wikipedia defines an **expression** as a combination of one or more explicit values, constants, variables, operators, and functions that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce ("to return", in a stateful environment) another value.

The point to note here is that an **expression has a value**. This value could be a primitive value or a reference. You can combine these values together using various operators to create even bigger expressions as per the rules of the language. You can say that if something has a value that can be assigned to a variable, then that something is an expression.

A **statement**, on the other hand, is a complete line of code that may or may not have any value of its own. You cannot combine statements to produce another statement. You can, of course, write one statement after another to create a **program**.

Thus, an expression may be a statement on its own. For example, consider the following code:

```
public class TestClass{
    public static void main(String[] args){
        int a = 10;
        int b = 20;
        a + b; //this line will not compile

        a = a + b;
    }
}
```

In the above code, `int a = 10;` is a statement. `int b = 20;` is another statement. But neither of them are expressions because they don't have a value of their own. You cannot assign `int a = 10` to a variable. On the other hand, `a + b` is an expression but is not a valid statement. If you try to compile the above code, you will get an error saying "**Not a statement**" at line `a + b;`

However, `a = a + b;` is a valid statement as well a valid expression. It is, in fact, an expression made by combining two expressions - `a` and `a + b` using the `=` operator. Furthermore, `a + b` is also an expression made by combining two expressions - `a` and `b` using the `+` operator.

The question that should pop into your head now is, if `a = a + b` is a valid expression, does it have a value? Yes, it does. As a matter of fact, the value of an expression built using the assignment operator is the same as the value that is being assigned to the variable on the left side of the `=` operator. In this case, for example, `a` is being assigned a value of `a + b`. Thus, the value of the expression `a = a + b` is the value produced by the expression on the right of `=` operator, i.e., `a + b`. You can actually test it out by assigning this whole expression to another variable like this - `int k = (a = a + b);`

What values can be combined using which operators is really the subject of this chapter.

6.1.3 Post and Pre Unary Increment/Decrement Operators [↳](#)

The `++` and `--` operators can be applied to a **variable** in the **postfix** form (i.e. appearing after the variable) or in the **prefix** form (i.e. appearing before the variable). In both the cases, the value of the variable will be incremented (or decremented) by `1`. The difference between the two is that the postfix operator returns the existing value of the variable while prefix operator returns the updated value of the variable. This is illustrated in the following code:

```
int i = 1, post = 0, pre = 0;  
post = i++;  
System.out.println(i+, " +post); //prints 2, 1
```

`i = 1; //resetting to i back to 1.`

```
pre = ++i;  
System.out.println(i+, " +pre); //prints 2, 2
```

To understand this, you need to look at process of evaluation of the expression `post = i++` step by step. The expression `post = i++`; is composed of a two parts - the variable `post` and the expression `i++` - joined using the assignment (`=`) operator. To evaluate this expression, you need to first evaluate the expression that is on the right side of `=`, i.e., `i++`.

Now, since a postfix operator increments the variable but returns the existing value of the variable, and since the existing value of the variable `i` is `1`, the expression `i++` evaluates to `1` even though the value of `i` has been incremented to `2`. Thus, the variable `post` is assigned a value of `1`.

Let us follow the same process for evaluating the expression `pre = ++i`. Here, since the prefix operator increments the variable and returns the updated

value of the variable, and since the updated value of the variable `i` is `2`, the expression `++i` evaluates to `2`. Thus, the variable `pre` is assigned a value of `2`.

You need to appreciate the fact that the value of the variable `i` and the value of the expression `i++` (or `++i`) are two different things and they may or may not be the same depending on whether you use pre or post form of the increment/decrement operator. When you do `int x = ++i;` you are not assigning the value of the variable `i` to `x`. You are assigning the value of the expression `++i` to `x` independently from the process of applying `++` to `i`. If you truly understand this concept then you should be able to determine the output of the following code:

```
int i = 1;  
i = i++;  
System.out.println(i); //what will this line print?
```

postfix and prefix for the exam ↗

In the exam, you may get questions where you are required to evaluate a compound expression containing multiple pre/post increment operators. They do get tricky sometimes but the key to solving such problems is to apply the concept explained above without losing focus. I will now show you a representative question and the steps to work out the answer.

Q. What will the following code print?

```
int a = 2;  
int b = 5;  
int c = a * (a++ - --b) * a * b;  
System.out.println(a+" "+b+" "+c);
```

1. We start with putting the values of the variables in the expression from left to right. Since the value of `a` at the beginning of the evaluation is `2`, the expression becomes:

`c = 2 * (a++ - -b) * a * b`

2. Next, `a` is incremented using the post increment operator. Therefore, the value of `a` used in the expression will be the existing value of `a`, i.e., `2` and then `a` is incremented to `3`. Therefore, the expression now becomes:

`c = 2 * (2 - -b) * a * b; (a is 3 now)`

3. `b` is decremented using the pre decrement operator. Therefore, `b` is decremented first to from `5` to `4` and then the new value of `b`, i.e., `4` is used in the expression. Therefore, the expression becomes:

`c = 2 * (2 - 4) * a * b; (a is now 3 and b is now 4)`

4. There are no further operations left to be applied on `a` and `b` anymore so, their values can be substituted in the expression. The expression now becomes:

`c = 2 * (2 - 4) * 3 * 4;`

5. Now you can apply the usual rules of operator precedence and brackets to evaluate the expression. You can easily see that the value of `2 * (2 - 4) * 3 * 4` is `-48`, which is the value that is assigned to the variable `c`.

6. Thus, the print statement will print `3 4 -48`

The key point that you should observe in the evaluation process described above is how the value of the variables change while the expression is being evaluated and their impact on the expression. For example, the value of variable `a` changed from `2` to `3` while the expression was being evaluated. This change causes part of the expression to use old value of `a` while the subsequent part to use the updated value of `a`. Although a similar change is not noticeable in case of `b` in this expression because `b` is decremented using pre decrement operator instead of post decrement, had there been a use of `b` in the expression before encountering `- -b`, the old value of `b` would have been used just like in the case of `a`.

To test this theory, try to compute the value of this slightly modified expression:

`c = b * a * (a++ - -b) * a * b`

When to use postfix and when to use prefix? [↳](#)

As shown in the code above, the difference between the two is very subtle and is material only when you use the unary increment and decrement operators inside of another expression. So, the obvious answer is to use postfix when you want to use the existing value of the variable and then update it, and use prefix when you want to update the value first and then use its value.

Ideally, however, you should avoid using these operators in a compound expression altogether because they cause confusion and are a common source of hard to find bugs in the code. My advice is to get into the habit of using just one form of increment or decrement operator in all of your code.

Using unary increment/decrement operators on wrappers [↳](#)

You can use `++` and `--` operators on any numeric primitive wrapper and they work the same way as explained above. You should, however, remember that primitive wrapper objects are immutable and therefore, incrementing or decrementing a wrapper object does not change that particular wrapper object. Instead, a new wrapper object is assigned to the reference on which increment or decrement operator is applied. The following code illustrate this point:

```
Integer i = 1;  
Integer j = i; //now, i and j point to the same Integer object
```

```
i++; //a different Integer object containing 2 is assigned to i
```

```
//j still points to the same Integer object containing 1
```

```
System.out.println(i+" "+j); //prints 2 1
```

6.1.4 String concatenation using + and += operators

The + operator is quite versatile. Besides performing the mathematical addition of numeric operands, it can also "add" two Strings together to create a new String. In that sense, you can say that the + operator is overloaded because its behavior changes based on the type of operands. When both of its operands are numeric values (or their primitive wrappers), it performs the mathematical addition but if either of its operands is of type String, it performs the String addition. By the way, the technical term for String addition is "**concatenation**" and so, I will use this term from now onward.

To trigger the String concatenating behavior of the + operator, the declared type of at least one of its operands must be a String. If one of the operands is a String and the other one is not, the other operand will be converted to a String first and then both the operands will be concatenated to produce the new String. Let us see a few examples to make it clear:

```
String s1 = "hello" + " world"; //both the operands are Strings
```

```
System.out.println(s1); //prints "hello world"
```

```
String s2 = "hello " + 1; //first operand is a String and second is an int
```

```
System.out.println(s2); //prints "hello 1"
```

```
Double d = 1.0;
String s = "2";
String s3 = d + s; //first operand is a Double and sec
```

ond is a String

```
System.out.println(s3); //prints "1.02"
```

There is no restriction on the type of the non-String operand. It can be of any type. The interesting part is how the non-String operand is converted to a String. The answer is simple. Recall that **java.lang.Object** is the root class of all objects in Java. This class contains a **toString** method that returns a String representation of that object. The **+** operator invokes this **toString** method on the non-String operand to get a String value. If the non-String operand is a primitive, then the primitive value is first converted to its corresponding wrapper object and **toString** is invoked on the resulting wrapper object. The following program illustrates this process.

```
public class TestClass{  
    public static void main(String[] args){  
        TestClass tc = new TestClass();  
        String str = tc.toString();  
        System.out.println( str );  
        System.out.println( "hello " + tc );  
    }  
}
```

This program produces the following output:

```
TestClass@15db9742  
hello TestClass@15db9742
```

As you can see, the **TestClass@15db9742** part of the concatenated String on the second line of the output is the same as the String returned by the **toString** method.

The **+** operator has one more trick up its sleeve. Check out this code:

```
public class TestClass{  
    public static void main(String[] args){  
        TestClass tc = null;  
  
        System.out.println( "hello " + tc );  
    }  
}
```

It prints `hello null`. Normally, when you invoke any method using a null reference, the JVM throws a **NullPointerException**. But in this case, no NullPointerException was thrown. How come? The reason is that the `+` operator checks whether the operand is `null` before invoking `toString` on it. If it is null, it uses the String `"null"` in place of that operand.

String concatenation using `+=` operator [↳](#)

The `+=` operator works in the same way as the `+` operator but with the additional responsibility of an assignment operator. The operand on the left of `+=` must be a String variable but the operand on right can be a value or variable of any type. For example,

```
String str = "2";  
str += 1; //this is the same as str = str + 1;
```

```
System.out.print(str); //prints "21"
```

Here is another example:

```
public class TestClass{  
    public static void main(String[] args){  
        TestClass tc = new TestClass();  
        String str = null;
```

```

        str += tc; //same as str = str + tc;

        System.out.print( str ); //prints nullTestClass@15
db9742

    }

}

```

Here, even though `str` is `null` and the second operand to `+=` is not a String, the String concatenation behavior of `+=` will be triggered because the declared type of `str` is `String`. Thus, `toString` will be called on the non-string operand `tc`. Furthermore, since `str` is `null`, the String "`null`" will be used while concatenating `str` and the string value returned by the call to `tc.toString()`.

You need to keep in mind that compound assignment operators do not work in declarations. For example, the statement `String str += "test";` will not compile because `str` is being declared in this statement.

6.1.5 Numeric promotion and casting [↳](#)

Take a look at the following code:

```

public class TestClass{
    public static void main(String[] args){
        byte b = 1;
        short s = -b;
        System.out.println(b);
    }
}

```

Looks quite simple, right? One may believe that it will print -1. But the fact is that it doesn't compile! It produces the following error message:

```
TestClass.java:4: error: incompatible types: possible  
lossy conversion from int to short  
    short s = -b;  
           ^  
1 error
```

There is no use of `int` anywhere in the code, yet, the error message is talking about converting something from `int` to `short`. The reason is that Java applies the rules of "**numeric promotion**" while working with operators that deal with numeric values, which are, in a nutshell, as follows:

1. Unary numeric promotion - If the type of an operand to a unary operator is smaller than `int`, the operand will automatically be promoted to `int` before applying the operator.

2. Binary numeric promotion - Both the operands of a binary operator are promoted to an `int` but if any of the operands is larger than an `int`, the other operand is promoted to the same type as the type of the larger one. Thus, for example, if both the operands are of type `byte`, then both are promoted to `int` but if any of the operands is a `long`, `float`, or `double`, the other one is also promoted to `long`, `float`, or `double`, respectively. Similarly, if the operands are of type `long` and `float`, `long` will be promoted to `float`.

A direct implication of the above two rules is that the result of applying an operator to numeric operands is of the same type as the type of the larger operand but it can never be smaller than an `int`.

If you look at the above code in light of these rules, the error message is quite obvious. Before applying the unary minus operator on `b`, the compiler promotes `b` to `int`. The result of applying the operator is, therefore, also an `int`. Now, `int` is a larger type than `short` and the compiler is concerned about possible loss of information while assigning a `int` value to a `short` variable and so, it refuses to compile the code. The language designers could have easily allowed such assignments without a fuss, but an inadvertent loss of information is often a cause of bugs, and so, they decided to make the programmer aware of this issue at the compile time itself.

You may wonder here that the value of `b` is `1` and the result of `-b` is just `-1`,

which is well within the range of `short`. Then what's the problem? What loss of information is the compiler talking about? You need to realize that compiler does not execute any code. It can't take into account the values that the variables may take at run time, while making decisions. So, even though you know that the result of the operation is small enough to fit into a `short`, the compiler cannot draw the same inference at compile time.

Let us now take look at few more examples where numeric promotion plays a role:

```
short s1 = 1;  
byte b1 = 1;  
byte b2 = 2;  
  
short s2 = +s1; //won't compile because the result will  
//be an int  
  
byte b = s1 + 2; //won't compile because the result will  
//be an int  
  
b = b1 & b2; //won't compile because the result will  
//be an int  
  
s2 = s1 << 1; //won't compile because the result will  
//be an int  
  
s2 = s1 * 1; //won't compile because the result will be an int  
  
float f = 1.0f; //recall that to write a float literal  
//you have to append it with an f or an F
```

```
double d = 1.0;  
  
int x = f - 1; //won't compile because the result will  
be a float  
  
float f2 = f + d; //won't compile because the result will  
be a double
```

All of the above operations are affected by the rule of numeric promotion and therefore, fail to compile. To make them compile, you need to assure the compiler that you know what you are doing, and that you are okay with the possible loss of information if there is any. You give this assurance to the compiler by explicitly casting the result back to the type of target variable. I have already discussed casting of primitives in the "Working with Java Data Types" chapter. The following code is the fixed version of the code shown above:

```
short s1 = 1;  
byte b1 = 1;  
byte b2 = 2;  
  
short s2 = (short) +s1;  
byte b = (byte) (s1 + 2);  
b = (byte) (b1 & b2); //numeric promotion happens for  
bit-wise operators as well  
  
s2 = (short) (s1 << 1);  
s2 = (short) (s1 * 1);  
  
float f = 1.0f; //recall that to write a float literal  
you have to append it with an f or an F  
  
double d = 1.0;
```

```
int x = (int) (f - 1);
float f2 = (float) (f + d);
```

An important point to note here is these rules come into picture only when the operands involve a variable and not when all the operands are constants and the result of the operation lies within the range of the target variable. The need for promotion does not arise while dealing with constants because the values are known at compile time and therefore, there is no possibility of loss of information at run time. If the value produced by the constants doesn't fit into the target variable, the compiler will notice that and refuse to compile it. Thus, the statement **byte b = 200 - 100;** will compile fine because **200** and **100** are compile time constants and the result of the operation fits into a **byte** even though one operand of the **-** operator falls outside the range of **byte**. But **byte b = 100 + 100;** will not compile because the result of **100 + 100** cannot fit into a **byte**. Similarly, the following code will also compile without any error:

```
final int I = 10;
byte b = I + 2;
```

No cast is needed because **I** is a compile time constant and the result of **I + 2** fits into a **byte**.

Curious case of unary increment/decrement and compound assignment operators [↳](#)

To put it simply, the rules of numeric promotion do not apply to **++**, **--** and the compound assignment operators such as **+=**, **-=**, and ***=**. They are the exceptions to the rules of numeric promotion. Thus, the following statements will compile fine without any explicit cast.

```
byte b1 = 1;
byte b2 = ++b1; //result of ++b1 will be a byte
```

```
b2 = b1--; //result of b1-- will be a byte
```

```
b1 *= b2; //result will be a byte  
  
double d = 1.0;  
float f = 2.0f;  
f += d; //result of will be a float
```

Observe the statements `b1 *= b2;` and `f += d;`. They behave as if they are the short hand for `b1 = (byte)(b1*b2);` and `f = (float)(f+d);` In other words, the result of a compound assignment operation is implicitly cast back to the target type irrespective of the type of the second operand. Similarly, `++b1;` behaves like `b1 = (byte) (b1 + 1);`

One could certainly make the "source of bugs" argument in the case of compound assignments as well. I guess, convenience superseded that argument in this case :)

Numeric Promotion and Primitive Wrapper Objects [↳](#)

Remember that to apply an operator to wrapper objects, they have to be unboxed first into their respective primitive values. Thus, the rules of numeric promotion and their exceptions will come into play here as well. Therefore, for example, `Byte bw = 1; bw = -bw;` will not compile but `--bw;` will compile fine due to the presence of an implicit cast as explained before.

The rules differ a bit in case of final wrapper variables though. For example, the following code will not compile:

```
final Byte b1 = 10;  
Byte b2 = -b1; //will not compile even though b1 is final
```

This doesn't work because `b1` is a reference to an object. It is this reference that is final, not the contents of the object to which it points. The compiler doesn't know that `Byte` is immutable. Thus, as far as the compiler is concerned, it is not sure that the result of `-b1` will fit into a `byte` and so, it refuses to compile the code.

6.1.6 Operator precedence and evaluation of expressions [🔗](#)

I am sure you have come across simple mathematical expressions such as $2 + 6 / 2$ at school. You know that this expression evaluates to 5 and not 4 because division has higher precedence than addition and so, $6 / 2$ will be grouped together instead of $2 + 6$. To change the default grouping, you use brackets (aka parentheses), i.e., $(2+6)/2$. You have most likely also come across the acronym **BODMAS** (or PEDMAS, in some countries), which stands for Brackets/Parentheses, Orders/Exponents, Division, Multiplication, Addition, and Subtraction). It helps memorize the conventional precedence order in which brackets are evaluated first, followed by powers, and then the rest in that order.

Java expressions are not much different from mathematical expressions. Their evaluation is determined by similar conventions and rules. The only problem is that there are a lot of operators to worry about and, as we saw earlier, the operators are not just mathematical. They are logical, relational, assignment, and so many other types as well. This makes Java expression evaluation a lot more complicated than a mathematical expression. But don't worry, you will not be required to evaluate complicated expressions in the exam. But you still need to know a few basic principles of expression evaluation to analyze code snippets presented in the questions correctly.

Precedence [🔗](#)

Precedence determines which operator out of two is evaluated "first", in a

conceptual sense. Another way to put it is, precedence determines how tightly an operator binds to its operands as compared to the other applicable operator in an expression. For example, in the case of $2 + 6 / 2$, the operand **6** can be bound to **+** or to **/**. But the division operator, having higher precedence than addition operator, binds to an operand more tightly than the addition operator. The addition operator, therefore, is not able to get hold of 6 as its second operand and has to wait until the division operator is done with it. In an expression involving multiple operators, the operator with highest precedence gets the operand, followed by the operator with second highest precedence, and so on.

The following table shows the precedence of all Java operators:

Operators	Precedence
member and array access operators . and []	
cast	()
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=
lambda	->

An important thing to observe from the above table is that the access operator and the cast operators have the highest precedence among all while the assignment operators and the lambda operator have the lowest precedence among all.

This explains why the following code doesn't compile:

```
int i = 0;  
byte b = (byte) i + 1;
```

Since the cast operator has higher precedence than the + operator, i is first cast to byte and then the addition is performed. The end result, therefore, is an **int** instead of a **byte**. You need to put **i + 1** in parentheses like this: **byte b = (byte)(i + 1);**

Associativity

Associativity of operators determines the grouping of operators when an expression has multiple operators of same precedence. For example, the value of the expression **2 - 3 + 4** depends on whether it is grouped as **(2 - 3) + 4** or as **2 - (3 + 4)**. The first grouping would be used if - operator is **left-associative** and the second grouping would be used if - operator is **right-associative**. It turns out that operators are usually grouped in the same fashion in which we read the expression, i.e., from **left to right**. In other words, almost all of the operators in Java are defined to be **left-associative**. The only exceptions are the assignment operators (simple as well as compound) and the ternary operator. Thus, the expression **2 - 3 + 4** will be grouped as **(2 - 3) + 4** and will evaluate to 3. But the expression **a = b = c = 5;** will be grouped as **a = (b = (c = 5)) ;** because the assignment operator is right associative. Here is another example that shows the impact of associativity:

```
String s1 = "hello";  
int i = 1;  
String s2 = s1 + 1 + i;  
System.out.println(s2); //prints hello11
```

The above code prints **hello11** instead of **hello2** because the + operator is left-associative. The expression **s2 = s1 + 1 + i;** is grouped as **s2 = (s1 + 1) + i;**. Thus, **s1+1** is computed first, resulting in the string

`hello1`, which is then concatenated with `1`, producing `hello11`.

A programming language could easily prohibit ambiguous expressions. There is no technical necessity for accepting the expression $2 + 6 / 2$ as valid when it can be interpreted in two different ways. The only reason ambiguous expressions are accepted is because it is considered too onerous for the programmer to resolve all ambiguity by using parenthesis when a convention already exists to evaluate mathematical expressions. Rules of Operator Precedence and Associativity are basically a programming language extension to the same convention that includes all sorts of operators. You can, therefore, imagine that operator precedence and evaluation order are used by the compiler to insert parenthesis in an expression. Thus, when a compiler sees $2 + 6 / 2$, it converts the expression to $2 + (6 / 2)$, which is what the programmer should have written in the first place.

You should always use parenthesis in expressions such as $2 - 3 + 4$ where the grouping of operands is not very intuitive.

Parenthesis

You can use parentheses to change how the terms of an expression are grouped if the default grouping based on precedence and associativity is not what you want. For example, if you don't want `2 - 3 + 4` to be grouped as `(2 - 3) + 4`, you could specify the parenthesis to change the grouping to `2 - (3 + 4)`.

Evaluation Order

Once an expression is grouped in accordance with the rules of precedence and

associativity, the process of evaluation of the expression starts. This is the step where computation of the terms of the expression happens. In Java, expressions are evaluated from left to right. Thus, if you have an expression `getA() - getB() + getC()`, the method `getA` will be invoked first, followed by `getB` and `getC`. This means, if the call to `getA` results in an exception, methods `getB` and `getC` will not be invoked.

Java also makes sure that operands of an operator are evaluated fully before the evaluation of the operator itself. Obviously, you can't compute `getA() + getB()` unless you get the values for `getA()` and `getB()` first.

The important point to understand here is that evaluation order of an expression doesn't change with grouping. Even if you use parentheses to change the grouping of `getA() - getB() + getC()` to `getA() - (getB() + getC())`, `getA()` will still be invoked before `getB()` and `getC()`.

Let me show you another example of how the above rules are applied while evaluating an expression. Consider the following code:

```
public class TestClass{  
    static boolean a ;  
    static boolean b ;  
    static boolean c ;  
    public static void main(String[] args) {  
        boolean bool = (a = true) || (b = true) && (c = true) ;  
        System.out.println(a + ", " + b + ", "+ c );  
    }  
}
```

Can you tell the output? It prints `true, false, false`. Surprised?

Many new programmers think that since `&&` has higher precedence, `(b = true) && (c = true)` would be evaluated first and so, it would print `true, true, true`. It would be logical to think so in a Mathematics class. However, evaluating a programming language expression is a two step process. In the first step, you have to use the rules of precedence and associativity to group the terms of the expression to remove ambiguity. Here, the operand `(b = true)` can be applied to `||` as well as to `&&`. However, since `&&` has higher

precedence than `||`, this operand will be applied to `&&`. Therefore, the expression will be grouped as `(a = true) || ((b = true) && (c = true))`. After this step, there is no ambiguity left in the expression. Now, in the second step, evaluation of the expression will start, which, in Java, happens from left to right. So, now, `a = true` will be evaluated first. The value of this expression is `true` and it assigns `true` to `a` as well. Next, since the first operand of `||` is true, and since `||` is a short circuiting operator, the second operand will not be evaluated and so, `(b = true) && (c = true)` will not be executed.

6.2 Exercise

1. Work out the values of the variables after each of the following statements on paper:

```
String str = "7" + 5 + 10;
str = 7 + 5 + "10";
str = "7" + (5 + 10);

int m = 50;
int n = ++m;
int o = m--;
int p = --o+m--;
int x = m<n?n<o?o<p?p:o:n:m;

int k = 4;
boolean flag = k++ == 5;
flag = !flag;
```

2. Which of the following lines will fail to compile and why? Write down the value of the variables after each line.

```
byte b = 1;
b = b<<1;
```

```
int c = b<<1;
byte d += b;
byte e = 0;
e += b;
```

3. What will the following code print and why?

```
String s = "a";
String[] sa = { "a", s, s.substring(0, 1), new String("a"), ""+'a' };
for(int i=0; i<sa.length; i++){
    System.out.println(i);
    System.out.println(s == sa[i]);
    System.out.println(s.equals(sa[i]));
}
```

Chapter 7 Using Decision Constructs

Use Java control statements including:

- if and if/else
- switch

7.1 Create if and if/else constructs

7.1.1 Basic syntax of if and if-else [🔗](#)

The **if** statement is probably the most used decision construct in Java. It allows you to execute a single statement or a block of statements if a particular condition is true. If that condition is false, the statement (or the block of statements) is not executed. The following are two ways you can write an **if** statement:

```
if( booleanExpression ) single_statement;
```

Notice that there are no curly braces for the statement. The **if** statement ends with the semi-colon. If you have multiple statements that you want to execute instead of just one, you can put all of them within a block like this:

```
if ( booleanExpression ) {  
    zero or more statements;  
}
```

Observe that there is no semi-colon after the closing curly brace. It is not an error if present though.

If-else statement [↳](#)

The **if-else** statement is similar to an **if** statement except that it also has an **else** part where you can write a statement that you want to execute if the **if** condition evaluates to false:

```
if( booleanExpression ) single statement;  
  //semi-colon required here  
  
else single statement;  
  
  //semi-colon required here, if-else statement ends with this semi-colon.
```

Again, if you have multiple statements to execute instead of just one, you can put them within a block:

```
if ( booleanExpression ) single statement;  
  //semi-colon is required here  
  
else {  
    zero or more statements;  
} //no semi-colon required here, but not an error if it exists.
```

or, if both the **if** and the **else** parts have multiple statements:

```
if ( booleanExpression ) {  
    zero or more statements;  
} //must NOT have a semi-colon here, error if it exists  
  
. . .  
else {  
    zero or more statements;  
}//no semi-colon required here, but not an error if it exists.
```

If **booleanExpression** evaluates to **true** , the statement (or the block of statements) associated with **if** will be executed and if the **booleanExpression** evaluates to **false** , the **else** part will be executed.

Remember that an empty statement (i.e. just a semicolon) is a valid statement and therefore, the following if and if-else constructs are valid:

```
boolean flag = true;  
  
if(flag); //does nothing, but valid  
  
if(flag); else; //does nothing, but valid  
  
if(flag);else System.out.println(true); //does nothing  
because flag is true  
  
if(flag) System.out.println(true); else; //prints true
```

By the way, you may see **if/if-else** statements being called **if-then/if-then-else** statement. This is not entirely correct. In some languages such Pascal, "then" is a part of the syntax but it is not in Java. However, "then" is not a keyword in Java and there is no "then" involved in the syntax of **if/if-else**.

7.1.2 Usage of if and if-else in the exam [↳](#)

Let us now look at a few interesting ways if/if-else is used in the exam that might trip you up.

Bad syntax [↳](#)

```
boolean flag = true;  
if( flag )  
else System.out.println("false"); //compilation error
```

In the above code, there is no statement or a block of statements for the if part. If you don't want to have any code to be executed if the condition is true but want to have code for the else part, you need to put an empty code block for the if part like this:

```
boolean flag = false;  
if( flag ) {  
}  
else System.out.println("false");
```

or

```
boolean flag = false;
```

```
if( flag ) ; else System.out.println("false");
}
```

Instead of having an empty if block, it is better to negate the if condition and put the code in the if block. Like this:

```
boolean flag = false;
if( !
    flag ) {
    System.out.println("false");
}
```

Bad Indentation

Unlike some languages such as Python, indentation (and extra white spaces, for that matter) holds no special meaning in Java. Indentation is used solely to improve readability of the code. Consider the following two code snippets:

```
boolean flag = false;
if(flag)
System.out.println("false");
else System.out.println("true");
{
System.out.println("false");
}
```

and

```
boolean flag = false;
if(flag)
    System.out.println("false");
else
    System.out.println("true");
```

```
{  
    System.out.println("false");  
}
```

The above two code snippets are equivalent. However, since the second snippet is properly indented, it is easy to understand that the last code block is not really associated with the if/else statement. It is an independent block of code and will be executed irrespective of the value of **flag**. This code will, therefore, print **true** and **false**.

Here is another example of bad indentation:

```
boolean flag = false;  
int i = 0;  
if(flag)  
    i = i +1;  
    System.out.println("true");  
else  
    i = i + 2;  
    System.out.println("false");
```

The above code is trying to confuse you into thinking that there are two statements in the if part and two statements in the else part. But, with proper indentation, it is clear what this code is really up to:

```
boolean flag = false;  
int i = 0;  
  
if(flag)  i = i +1;  
  
System.out.println("true");  
  
else i = i + 2;  
  
System.out.println("false");
```

It turns out that the last three lines of code are independent statements. The **else** statement is completely out of place because it is not associated with the **if** statement at all and will, therefore, cause compilation error.

Missing else

As we saw earlier, the **else** part is not mandatory in an **if** statement. You can have just the **if** statement. But when coupled with bad indentation, an **if** statement may become hard to understand as shown in the following code:

```
boolean flag = true;  
if(flag)  
System.out.println("true");  
{  
System.out.println("false");  
}
```

The above code prints **true** and **false** because there is no **else** part in this code. The second **println** statement is in an independent block and is not a part of the **if** statement.

Dangling else

"Dangling else" is a well known problem in programming languages that have **if** as well as **if-else** statements. This is illustrated in the following piece of code that has two **if** parts but only one **else** part:

```
int value = 3;  
if(value == 0)  
if(value == 1) System.out.println("b");  
else System.out.println("c");
```

The question here is with which **if** should the **else** be associated? There are two

equally reasonable answers to this question as shown below:

```
int value = 3;
if(value == 0) {
    if(value == 1) System.out.println("b");
}
else System.out.println("c");
```

and

```
int value = 3;
if(value == 0) {
    if(value == 1)
        System.out.println("b");
    else
        System.out.println("c");
}
```

In the first interpretation, **else** is associated with the first **if** , while in the second interpretation, **else** is associated with the second **if** . If we go by the first interpretation, the code will print **b** , and if we go by the second interpretation, the code will not print anything. For a compiler, both are legally valid ways to interpret the code, which makes the code ambiguous.

Since neither of the interpretations is more correct than the other, Java language designers broke the tie by deciding to go with the second interpretation, i.e., the **else** is to be associated with the nearest **if** . That is why the above code does not print anything as there is no **else** part associated with the first **if** . Based on the above discussion, you should now be able to tell the output of the following code:

```
int value = 3;
if(value == 0)
if(value == 1)
    System.out.println("b");
else
```

```
        System.out.println("c");
else
    System.out.println("d");
```

Just follow the rule that an else has to be associated with the nearest if. The following is how the above statement will be grouped:

```
int value = 3;

if(value == 0){
    if(value == 1)
        System.out.println("b");
    else
        System.out.println("c");
}
else System.out.println("d");
```

It will print **d**.

Using assignment statement in the if condition

Recall that every assignment statement itself is a valid expression with a value of its own. Its type and value are the same as the ones of the target variable. This fact can be used to write a very tricky if statement as shown below:

```
boolean flag = false;
if(flag = true

){

    System.out.println("true");
}
else {
    System.out.println("false");
}
```

Observe that **flag** is not being compared with **true** here. It is being assigned

the value `true`. Thus, the value of the expression `flag = true` is true and that is why the `if` part of the statement is executed instead of the `else` part. While this type of code is not appreciated in a professional environment, a similar construct is quite common:

```
String data = null;  
if( (data = readData()) != null ) //assuming that read  
Data() returns a String  
  
{  
    //do something  
  
}
```

The above code is fine because the assignment operation is clearly separated from the comparison operation. The value of the expression `data = readData()` is being compared with `null`. Remember that the value of this expression is the same as the value that is being assigned to `data`. Thus, the `if` body will be entered only if `data` is assigned a non-null value.

Using pre and post increment operations in the if condition [↳](#)

You will see conditions that use pre and post increment (and decrement) operators in the exam. Something like this:

```
int x = 0;  
if(x++ > 0){ //line 2  
  
    x--; //line 3  
  
}
```

```
if (++x == 2){ //line 6  
  
    x++; //line 7  
  
}  
System.out.println(x);
```

You can spot the trick easily if you have understood the difference between the value of an expression and the value of a variable used in that expression (I explained this in the "Using Operators" chapter). At line 2, `x` will be incremented to `1` but value of the expression `x++` is `0` and therefore, the condition will be evaluated to `false`. Thus, line 3 will not be executed. At line 6, `x` is incremented to `2` and the value of the expression `++x` is also `2`. Therefore, the condition will be evaluated to `true` and line 7 will be executed, thereby increasing the value of `x` to `3`. Thus, the above code will print `3`.

Remember that conditions are used in ternary expressions and loops as well, so, you need to watch out for this trick there also.

7.2 Create ternary constructs

7.2.1 The ternary conditional operator ?: [↳](#)

The ternary operator is not mentioned in the exam objectives explicitly (although it was mentioned explicitly in OCA 8 objectives). We haven't seen any one getting a question on it either. However, it is one of the commonly used operators in Java and from that perspective, it falls within the scope of the exam objectives. You should go through the following discussion only if you have time.

The syntax of the ternary operator is as follows:

```
operand 1 ? operand 2 : operand 3;
```

Operand 1 must be an expression that returns a **boolean**. The **boolean** value of this expression is used to decide which one of the rest of the two other operands should be evaluated. In other words, which of the operands 2 and 3 will be evaluated is conditioned upon the return value of operand 1. If the boolean expression given in operand 1 returns true, the ternary operator evaluates and returns the value of operand 2 and if it is false, it evaluates and returns the value of operand 3. From this perspective, it is also a "**conditional operator**" (as opposed to **&**, **|**, **!**, and **^**, which are really just "**logical**" operators).

Examples:

```
boolean sweet = false;  
int calories = sweet ? 200 : 100; //assigns 100 to calories
```

```
boolean sweetflag = (calories == 100 ? false : true);/  
/assigns false to sweetflag
```

```
boolean hardcoded = false;  
//assuming getRateFromDB() method returns a double.
```

```
double rate = hardcoded ? 10.0 : getRateFromDB(); //invokes method getRateFromDB
```

```
String value = sweetflag ? "Sweetened" : "Unsweetened";
```

```
Object obj = sweetflag ? "Sweetened" : new Object();
```

The ternary conditional operator is often thought of a short form for the **if/else statement** but it is similar to the if/else statement only up to conditional evaluation of its other two operands part. Their fundamental difference lies in the fact that the ternary expression is just an **expression** while an if/else statement is a **statement**. As discussed earlier, every expression must have a value and so, must the ternary expression. Since the value of a ternary expression is the value returned by the second or the third operand, the second and third operands of the ternary operator can comprise any expression. As the example given above shows, they can also include **non-void method invocations**. There is no such restriction with the if/else statement. The following example highlights this difference:

```
boolean flag = true;
if(flag) System.out.println("true");
else System.out.println("false");
```

The above if/else statement compiles fine and prints **true** but a similar code with ternary expression does not compile.

```
flag ? System.out.println("true") : System.out.println("false");
```

The reason for non-compilation of the above code is two fold. The first is that a ternary expression is not a valid statement on its own, which means, you cannot just have a free standing ternary expression. It can only be a part of a valid statement such as an assignment. For example,

```
int x = flag ? System.out.println("true")
: System.out.println("false");
```

This brings us to the second reason. The second and third operands in this example are invocations methods that return **void**. Obviously, you cannot assign void to an int variable. In fact, you cannot assign void to any kind of variable. Therefore, it fails to compile.

Type of a ternary conditional expression [↳](#)

Now that we have established that a ternary conditional expression must return a value, all that is left to discuss is the type of the value that it can return. It can return values of three types: **boolean**, **numeric**, and **reference**. (There are no other types left for that matter!)

If the second and third operands are expressions of type boolean (or Boolean), then the return type of the ternary expression is boolean. For example:

```
int a = 1, b = 2;  
boolean flag = a == b? true : false; //ternary expression that returns a boolean
```

If the second and third operands are expressions of a numeric types (or their wrapper classes), then the return type of the ternary expression is the wider of the two numeric types. For example, `double d = a == b? 5 : 10.0;`. Observe here that the second operand is of type `int` while the third is of type `double`. Since `double` is wider than `int`, the type of this ternary expression is `double`. You cannot, therefore, do `int d = a == b? 5 : 10.0;` because you cannot assign a double value to an int variable without a cast.

If the second and third operands are neither of the above, then the return type of the ternary expression is reference. For example,

`Object str = a == b? "good bye" : "hello";` Here operands 2 and 3 are neither numeric nor boolean. Therefore, the return type of this ternary expression is a reference type.

The first two types are straight forward but the third type begs a little more discussion. Consider the following line of code:

```
Object obj = a == b? 5 : "hello"; .
```

Here, the second operand is of a numeric type while the third is of type `String`. Since this falls in the third category, the return type of the expression `a == b? 5 : "hello";` must be a reference. The question before the compiler now is what should be the type of the reference that is returned by the

expression. One operand is an `int`, which can be boxed into an `Integer` object and another one is a `String` object. If the `boolean` condition evaluates to `true`, the expression will return an `Integer` object and if the condition evaluates to `false`, the expression will return a `String` object. Remember that the compiler cannot execute any code and so, it cannot determine what the expression will return at run time. Thus, it needs to pick a type that is compatible with both kind of values. The compiler solves this problem by deciding to pick the most specific common superclass of the two types as the type of the expression. In this case, that class is `java.lang.Object`. By selecting the most specific common super class, the compiler ensures that irrespective of the result of the condition, the value returned by this expression will always be of the same type, i.e., `java.lang.Object`, in this case. Here is an example, where the most specific common super class is not `Object`:

```
Double d = 10.0;  
Byte by = 1;  
Number n = a == b? d : by;
```

Here, the most specific common superclass of `Double` and `Byte` is `Number`. (Recall that all wrapper classes for integral types extend from `java.lang.Number`, which in turn extends from `java.lang.Object`). You can therefore, assign the value of the expression to a variable of type `Number`.

You should now be able to tell the result of the following two lines of code:

```
int value = a == b? 10 : "hello"; //1  
  
System.out.println(a == b? 10 : "hello"); //2
```

As discussed above, the type of the expression `a == b? 10 : "hello"`; is `Object`. Can you assign an `Object` to an `int` variable? No, right? Therefore, the first line will not compile. Can you pass an `Object` to the `println`

method? Of course, you can. Therefore, the second line will compile and run fine.

The short circuiting property of ?: ↴

Depending of whether the value of operand one is true or false, either operand 2 or operand 3 is evaluated. In other words, if the condition is `true`, operand 3 is not evaluated and if the condition is `false`, operand 2 is not evaluated. In this respect, the ternary conditional operator is similar to the other two short circuiting operators, i.e., `&&` and `||`. However, there is an important difference between the two. While with `&&` and `||`, evaluation of both the operands is possible in certain situations, it is never the case with `?:` operator. `?:` evaluates exactly one of the two operands in all situations.

The short circuiting nature of `?:` provides a good opportunity for trick questions in the exam. For example, what will the following code print?

```
int x = 0;  
int y = 1;  
System.out.println(x>y? ++x : y++);  
System.out.println(x+" "+y);
```

This code prints:

1
0 2

Since the value of `x` is not greater than `y`, `x>y` evaluates to `false` and therefore, the ternary expression returns the value of the third operand, which is `y++`. Since `y++` uses post-increment operator, the return value of `y++` will be the current value of `y`, which is `1`. `y` will then be incremented to `2`. Observe that evaluation of the second operand is short circuited because the first operand evaluates to false. Therefore, `++x` is never executed. Thus, the second print statement prints `0 2`.

7.3 Use a switch statement

7.3.1 Creating a switch statement [🔗](#)

A switch statement allows you to use the value of a variable to select which code block (or blocks) out of multiple code blocks should be executed. Here is an example:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = args.length;  
  
        switch(i) { //switch block starts here  
  
            case 0 : System.out.println("No argument");  
                      break;  
            case 1 : System.out.println("Only one argument");  
                      break;  
            case 2 : System.out.println("Two arguments");  
                      break;  
            default : System.out.println("Too many arguments!");  
                      break;  
  
        }//switch block ends here  
  
        System.out.println("All done.");  
    }  
}
```

There are four blocks of code in the above switch statement. Each block of code is associated with a **case label**. Depending of the value of the variable **i**, the

control will enter the code block associated with that particular case label and keep on executing statements until it finds a **break** statement. For example, if the value of **i** is **0**, the control will enter the first code block. It will print **No argument** and then encounter the **break** statement. The break statement causes the control to exit the switch statement and move on to the next statement after the switch block, which prints "**All done**" .

If the value of **i** doesn't match with any of the case labels, the control looks for a block labelled **default** and enters that block. If there is no default block either, the control does not enter the switch block at all. Since this "switching" is done based on the expression specified in the switch statement (which is just **i** in this example), this expression is aptly called the "**switch expression**" .

Operationally, this seems quite similar to a cascaded **if/else** statement. Indeed, the above code can very well be written using an if/else statement as follows:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = args.length;  
  
        if(i == 0) {  
            System.out.println("No argument");  
        } else if (i == 1) {  
            System.out.println("Only one argument");  
        } else if (i == 2) {  
            System.out.println("Two arguments");  
        } else {  
            System.out.println("Too many arguments!");  
        }  
        System.out.println("All done.");  
    }  
}
```

Well, why do you need a switch statement then, you may ask. To begin with, as you can see, an if/else statement is a lot more verbose than a switch statement. The switch version is also a little easier to comprehend than the if/else version. But underneath this syntactic ease lies a complicated beast. This is evident when we look at the moving parts involved in a switch statement more closely.

The switch expression [↑](#)

A switch expression must evaluate to one of the following three kinds:

1. a limited set of integral types (`byte`, `char`, `short`, `int`), and their wrapper classes. Observe that even though `long` is an integral type, it cannot be the type of a switch variable. `boolean`, `float`, and `double` are not integral types anyway and therefore, cannot be the type of a switch variable either.
2. `enum` type
3. `java.lang.String` - Generally, reference types are not allowed as switch expressions but an exception for `java.lang.String` was made in Java 7. So now, you can use a String expression as a switch expression.

Compare this to an `if/else` statement where branching is done based on the value of a **boolean expression**. This limits an if/else statement to at most two branches. Of course, as we saw earlier, you can cascade multiple if/else statements to achieve multiple branches.

The case labels [↑](#)

Case labels must consist of **compile time constants** that are assignable to the type of the switch expression. For example, if your switch expression is of type `byte`, you cannot have a case label with a value that is larger than a byte. Similarly, if your switch expression is of type `String`, the case labels must be constant string values as illustrated in the following code:

```
public static void main(String[] args){  
    switch(args[0]) {  
        case "1" : System.out.println("one"); //valid because "1" is a compile time constant  
  
        case "1"+"2" : System.out.println("one"); //valid because "1"+"2" is a compile time constant  
  
        case args[1] : System.out.println("same args");//wi
```

11 not compile because args[1] is not a compile time constant

```
    case "abc".toUpperCase() : System.out.println("ABC")
);//will not compile because "abc".toUpperCase() is not a compile time constant
```

```
}
```

Observe that "**1**" and "**1**"+"**2**" are compile time constants because the value of these expressions is known at compile time, while **args[1]** and **"abc".toUpperCase()** are not compile time constants because their values can only be determined at run time when the code is executed.

The interesting thing about case labels is that they are **optional**. In other words, a switch statement doesn't necessarily have to have a case label. The following is, therefore, a superfluous yet valid switch statement.

```
switch(i){
    default : System.out.println("This will always be printed");
}
```

Another point worth repeating here is that although it is very common to use a single variable as the switch expression but you can use any expression inside the switch. And when you talk of an expression, all the baggage of numeric promotion, casting, and operator precedence that we saw previously, comes along with it. You need to consider all that while checking the validity of case labels. For example, while the following code fails to compile:

```
byte b = 10;
switch(b){ //type of the switch expression here is byte
```

```
case 1000 : //1000 is too large to fit into a byte  
    System.out.println("hello!");  
}
```

the following code compiles fine:

```
byte b = 10;  
switch(b+1){ //type of the switch expression here is now int due to numeric promotion  
  
case 1000 : //1000 can fit into an int  
  
    System.out.println("hello!");  
}
```

The default block [↳](#)

There can be at most one default block in a switch statement. The purpose of the default block is to specify a block of code that needs to be executed if the value of the switch expression does not match with any of the case labels. Just like the case labels, this block is also **optional**.

The order of case and default blocks [↳](#)

Java does not impose any particular order for the case statements and the default block. Thus, although it is customary to have the default block at the end of a switch block, you can have it even at the beginning. Similarly, Java does not care about the ordering of the case labels.

However, "does not care" does not mean "not important"! Ordering of case and

default blocks becomes very important in combination with the use of the **break** statement as we will see next.

The break statement ↗

I mentioned in the beginning that the case labels determine the entry point into a switch statement and the break statement determines the exit. That is true but the interesting thing is that even the break statement is **optional**. A case block does not necessarily have to end with a break. Let me modify the program that I showed you in the beginning:

```
public class TestClass {  
    public static void main(String[] args){  
  
        switch(args.length) { //switch block starts here  
  
            case 0 : System.out.println("No argument");  
                      //break;  
  
            case 1 : System.out.println("Only one argument");  
                      //break;  
  
            case 2 : System.out.println("Two arguments");  
                      //break;  
  
            default : System.out.println("Too many arguments!");  
                      //break;
```

```
//switch block ends here  
        System.out.println("All done.");  
    }  
}
```

I have commented out all the break statements. Now, if you run this program **without any argument**, you will see the following output:

```
No argument  
Only one argument  
Two arguments  
Too many arguments!  
All done.
```

The control entered at the block labelled **case 0** (because `args.length` is **0**), and executed all the statements of the switch block...even the statements associated with other case blocks that did not match the value of `args.length`. This is called "**fall through**" behavior of a switch statement. In absence of a break statement, the control falls through to the next case block and the next case block, and so on until it reaches the end of the switch statement. This feature is used when you want to have one code block to execute for multiple values of the switch expression. Here is an example:

```
char ch = 0;  
int noOfVowels = 0;  
while( (ch = readCharFromStream()) > 0) {  
    switch(ch) {  
  
        case 'a' :  
        case 'e' :  
        case 'i' :  
        case 'o' :  
        case 'u' :  
            noOfVowels++;
```

```
        default : logCharToWhatever(ch);
    }
}
System.out.println("Number of vowels received "+noOfVowels);
```

The above code logs each character it receives but increments `noOfVowels` only if the character received is a lower case vowel.

You will see questions in the exam on this behavior of the switch statement. So pay close attention to where in the switch block does the control enter and where it exits. Always check for missing break statements that cause the control to fall through to the next case block.

The following is a typical code snippet you may get in an exam. Try running it with different arguments and observe the output in each case:

```
public class TestClass {
    public static void main(String[] args){
        int i = 0;
        switch(args[0]) {

            default : i = i + 3;
            case "2" : i = i + 2;
            case "0" : break;
            case "1" : i = i + 1;

        }

        System.out.println("i is "+i);
    }
}
```

The fall through behavior of a switch statement does not really get used a lot in Java but it has been used in interesting ways to optimize code in other languages. If you have time, you might want to google "duff's

device" to see one such usage. This is, of course, not required for the exam :)

7.4 Exercise

1. Write a method that accepts a number as input and prints whether the number is odd or even using an if/else statement as well as a ternary expression.
2. Accept a number between 0 to 5 as input and print the sum of numbers from 1 to the input number using code that exploits the "fall through" behavior of a switch statement.
3. Accept a number as input and generate output as follows using a cascaded and/or nested if/else statement - if the number is even print "even", if it is divisible by 3, print "three", if it is divisible by 5, print "five" and if it is not divisible by 2, 3, or 5, print "unknown". If the number is divisible by 2 as well as by 3, print "23", and if the number is divisible by 2, 3, and 5, print "235".
4. Indent the following code manually such that it reflects correct if - else associations. Use a plain text editor such as Notepad. Copy the code into a Java editor such Netbeans or Eclipse and format it using the editor's auto code formatting function. Compare your formatting with the editor's.

```
int a = 0, b = 0, c = 0, d = 0;
boolean flag = false;
if (a == b)
if (c == 10)
{
if (d > a)
{
} else {
}
if (flag)
System.out.println("");
else
```

```
System.out.println("");
}
else if (flag == false)
System.out.println("");
else if (a + b < d) {
System.out.println("");
}
else
System.out.println("");
else d = b;
```

Chapter 8 Using Loop Constructs

- Create and use while loops
- Create and use do/while loops
- Create and use for and for each loops including nested loops
- Use break and continue statements

8.1 What is a loop

8.1.1 What is a loop

A loop causes a set of instructions to execute repeatedly until a certain condition is reached. It's like when the kids keep asking, "are we there yet?", when you are on a long drive in a car. They ask this question in a "loop", until you are really there :) Or until you "break" their loop by putting on a DVD.

Well, in Java, loops work similarly. They let you execute a group of statements multiple times or even forever depending on the **loop condition** or until you **break** out of them. Every repetition of execution of the statements is called an **iteration**. So, for example, if a group of statements is executed 10 times, we can say that the loop ran for 10 iterations.

Loops are a fundamental building block of **structured programming** and are used extensively for tasks ranging from the simple such as counting the sum of a given set of numbers to the complicated such as showing a dialog box to the user until they select a valid file.

Java has three different ways in which you can create a loop - using a while statement, using a do/while statement, and using a for statement. Let us take a look at each of them one by one.

8.2 Create and use while loops

8.2.1 The while loop [🔗](#)

As the name of this loop suggests, a while loop executes a group of statements **while** a condition is true. In other words, it checks a condition and if that condition is true, it executes the given group of statements. After execution of the statements, i.e., after finishing that iteration, it loops back to check the condition again. If the condition is false, the group of statements is not executed and the control goes to the next statement after the while block. The syntax of a while loop is as follows:

```
while (boolean_expression) {  
    statement(s);  
}
```

and here is an example of its usage:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = 4;  
        while(i>0){  
            i--;  
            System.out.println("i is "+i);  
        }  
        System.out.println("Value of i after the loop is  
"+i);  
    }  
}
```

It produces the following output:

```
i is 3  
i is 2  
i is 1  
i is 0  
Value of i after the loop is 0
```

Observe that the condition is checked **before** the group of statements is executed

and that once the condition `i>0` returns `false`, the control goes to the next statement after the while block.

As you have seen in the past with if/else blocks, if you have only a single statement that you need to execute in a loop, you may get rid of the curly brackets if you want. In this case, the syntax becomes:

```
while(boolean_expression) statement;
```

The example program given above can also be written to use a single statement like this:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = 4;  
        while(i-->0) System.out.println("i is "+i); //no  
        curly braces  
  
        System.out.println("Value of i after the loop is  
        "+i);  
    }  
}
```

The above code looks the same as the previous one but produces a slightly different output:

```
i is 3  
i is 2  
i is 1  
i is 0  
Value of i after the loop is -1
```

The difference is that I have used the post-decrement operator to decrement the value of `i` within the condition expression itself. Remember that when you use the **post**-decrement (or the **post**-increment) operator, the variable is decremented (or incremented) **after** its existing value is used to evaluate the expression. In this case, when `i` is `0`, the condition expression evaluates to false

and the while block is not executed, but the variable `i` is nevertheless decremented to `-1`. Therefore, the value of `i` after the loop is `-1`.

8.2.2 Using a while loop [↳](#)

Control condition and control variable [↳](#)

When using a while loop you should be careful about the expression that you use as the while condition. Most of the time, a while condition comprises a single variable (which is also called the "**control variable**", because it controls whether the loop will be entered or not) that you compare against a value. For example, `i>4` or `name != null` or `bytesRead != -1` and so on.

However, it can get arbitrarily large and complex. For example, `account == null || (account != null && account.accountId == null) || (account != null && account.balance == 0)`. The expression must, however, return a `boolean`. Unlike some languages such as C, Java does not allow you to use an integer value as the while condition. Thus, `while(1) System.out.println("loop forever");` will not compile.

while body [↳](#)

Ideally, the code in the while body should modify the control variable in such a way that the control condition will evaluate to false when it is time to end the loop. For example, if you are processing an array of integers, your control variable could be set to the index of the element that you are processing, and each iteration should increment that variable. For example,

```
int[] myArrayOfInts = //code to fetch the data
```

```
int i = 0; //control variable
```

```
while(i<myArrayOfInts.length){
```

```
//do something with myArrayOfInts[i]

i++;//increment i so that the control condition will evaluate to false after the last element is processed.

}
```

There are situations where you do not want a loop to end at all. For example, a program that listens on a socket for connections from clients may use a while loop as follows:

```
Socket clientSocket = null;
while( (clientSocket = serverSocket.accept() ) != null
){
    //code to hand over the clientSocket to another thread and go back to the while condition to keep listening for connection requests from clients.

}
```

The above is a commonly used while loop idiom.

A never ending while loop can also be as simple as this:

```
while(true){
    System.out.println("keep printing this line forever!");
}
```

On the other extreme, keep an eye for a condition that never lets the control enter the while body:

```
int i = 0;
while(i>0){ // the condition is false to begin with
```

```
System.out.println("hello"); //this will never be printed
```

```
i++;  
}
```

It is possible to exit out of a while loop without checking the while condition. This is done using the **break** statement. I will discuss it later.

8.3 Create and use do/while loops

8.3.1 The do-while loop [🔗](#)

A do-while loop is similar to the while loop. The only difference between the two is that in a do-while loop the loop condition is checked after executing the loop body. Its syntax is as follows:

```
do {  
    statement(s);  
}while(boolean_expression);
```

Observe that a do-while statement starts with a "do" but there is no "do" anywhere in a while statement. Another important point to understand here is that since the loop condition is checked after the loop body is executed, the loop body will always be executed at least once.

As with a while statement, it is ok to remove the curly brackets for the loop body if there is only one statement in the body. Thus, the following two code snippets are actually the same:

```
int i = 4;  
do {
```

```
    System.out.println("i is "+i);
} while(i-->0);
System.out.println("Loop finished. i is "+i);
```

```
int i = 4;
do
    System.out.println("i is "+i);
while(i-->0);
System.out.println("Loop finished. i is "+i);
```

Deciding whether to use a while loop or a do while loop is easy. If you know that the loop condition may evaluate to false at the beginning itself, i.e., if the loop body may not execute even once, you must use a while loop because it lets you check the condition first and then execute the body. For example, consider the following code:

```
Iterator<Account> acctIterator = accounts.iterator();
while(acctIterator.hasNext()){ //no need to enter the
    loop body if accounts collection is empty.
```

```
    Account acct = acctIterator.next();
    //do something with acct
```

```
}
```

Don't worry about the usage of `Iterator` or `<Account>` in the above code. The point to understand here is that you want to process each account object in the accounts collection and if there is no account object, you don't want to enter the loop body at all. If you use a do-while loop, the code will look like this:

```
Iterator<Account> acctIterator = accounts.iterator();
do {
    Account acct = acctIterator.next(); //will throw a
    n exception
```

```
//do something with acct  
}  
while(acctIterator.hasNext());
```

The above code will work fine in most cases but will throw an exception if the account collection is empty. To achieve the same result with a do-while loop, you would have to write an additional check for an empty collection at the beginning. Something like this:

```
Iterator<Account> acctIterator = accounts.iterator();  
if(acctIterator.hasNext()) {//no need to enter the if  
body if accounts collection is empty.
```

```
do{  
    Account acct = acctIterator.next();  
    //do something with acct  
  
}while(acctIterator.hasNext())  
}
```

I think the choice is quite clear. A while loop is a natural fit in this case.

Generally, a while loop is considered more readable than a do-while loop and is also used a lot more in practice. I have not needed to use a do-while loop in a long while myself :)

8.4 Create and use for loops

8.4.1 Going from a while loop to a for loop [↳](#)

The **for loop** is the big daddy of loops. It is the most flexible, the most complicated, and the most used of all loop statements. It has so many different

flavors that many programmers do not get to use some of its variations despite years of programming in Java. But don't be scared. It still follows the basic idea of a loop, which is to let you execute a group of statements multiple times.

To ease you into it, I will morph the code for a while loop into a for loop. Here is the code that uses a while loop:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = 4;  
        while(i>0){  
            System.out.println("i is "+i);  
            i--;  
        }  
        System.out.println("Value of i after the loop is "  
+i);  
    }  
}
```

The output of the above code is:

```
i is 4  
i is 3  
i is 2  
i is 1  
Value of i after the loop is 0
```

Here is the same code with a for loop:

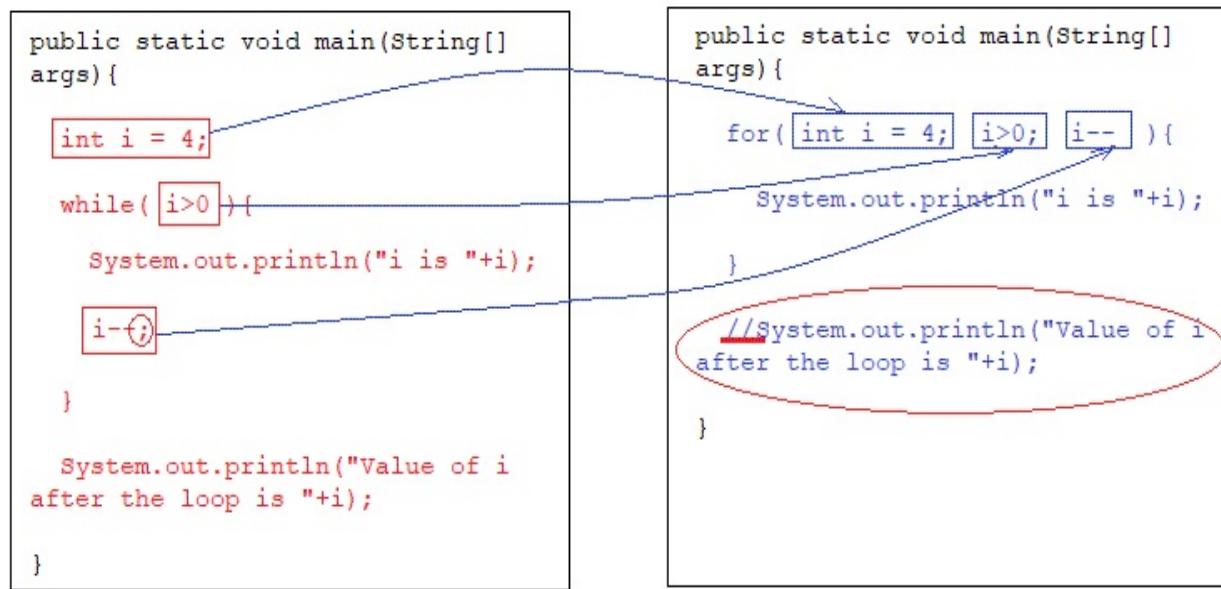
```
public class TestClass {  
    public static void main(String[] args){  
        for(int i = 4; i>0; i--){  
            System.out.println("i is "+i);  
        }  
        //System.out.println("Value of i after the loop is "  
+"i);
```

```
    }  
}
```

Its output is:

```
i is 4  
i is 3  
i is 2  
i is 1
```

As you can see, it produces the same output except the last line. The following image shows how the elements of a while loop of the first code were mapped to the for loop of the second code.



Here are a few things that you should observe in the transformation shown above.

1. The while statement contains just the comparison but the for statement contains initialization, comparison, and updation. Technically, the declaration of the loop variable `i` and its updation through `i--` is not really a part of the while statement. But the for loop has a provision for both.
2. The implication of declaring the loop variable `i` in a for statement is that it

is scoped only to the for block and is not visible outside the for block, which is why I have commented out the last print statement in the second code.

3. There is no semi-colon after `i--` in the for statement.

This example illustrates that fundamentally there is not much difference between the two loop constructs. Both the loops have the same three basic components - declaration and/or initialization of a loop variable, a boolean expression that determines whether to continue next iteration of the loop, and a statement that updates the loop variable. But you can see that the for loop has a compact syntax and an in-built mechanism to control the initialization and updation of the loop variable besides the comparison as compared to the while loop.

8.4.2 Syntax of a for loop

A for loop has the following syntax:

```
for( optional initialization section ;  
     optional condition section ;  
     optional updation section ) {  
     statement(s);  
}
```

Predictably, if you have only zero or one statement in the for block, you can get rid of the curly braces and end the statement with a semicolon like this:

```
for( optional initialization section ;  
     optional condition section ;  
     optional updation section ) ;  
//no body at all
```

or

```
for( optional initialization section ;  
     optional condition section ;  
     optional updation section )  
     single_statement;
```

All three sections of a for statement are optional but the **two semi-colons** that separate the three sections are not. You are allowed to omit one, two, or all of the three sections. Thus, `for(; ;);` is actually a valid for statement but `for();` and `for();;` are not.

Order of execution ↴

Various pieces of a for loop are executed in a specific order, which is as follows:

1. The **initialization section** is the first that is executed. It is executed **exactly once** irrespective of how many times the for loop iterates. If this section is **empty**, it is **ignored**.
2. Next, the **condition section** is executed. If the result of the expression in this section is true, the next iteration, i.e., execution of the statements in the for block is kicked off. If the result is false, the loop is terminated immediately, i.e., neither the for block and nor the updation section is executed. If this section is **empty**, it is assumed to be **true**.
3. The statements in the **for block** are executed.
4. The **updation section** is executed. If this section is **empty**, it is **ignored**.
5. The control goes back to the **condition section**.

Let us see how the above steps are performed in the context of the following for loop.

```
for(int i = 3; i>0; i--){  
    System.out.println("i is "+i);
```

}

1. `int i = 3;` is executed. So, `i` is now `3`.
2. Expression `i>0` is evaluated and since `3` indeed greater than `0`, it evaluates to `true`. Therefore, the next iteration will now commence.
3. The print statement is executed, which prints `3`.
4. `i--` is executed thereby reducing `i` to `2`.
5. Control goes back to the condition section. `i>0` evaluates to `true` because `2` is greater than `0`. Therefore, the next iteration will now commence.
6. The print statement is executed, which prints `2`.
7. `i--` is executed thereby reducing `i` to `1`.
8. Control goes back to the condition section. `i>0` evaluates to `true` because `1` is greater than `0`. Therefore, the next iteration will now commence.
9. The print statement is executed, which prints `1`.
10. `i--` is executed thereby reducing `i` to `0`.
11. Control goes back to the condition section. `i>0` evaluates to `false` because `0` is not greater than `0`. Therefore, the loop will be terminated. The statements in the for block and the updation section will not be executed and the control goes to the next statement after the for block (which is the end of the method in this case).

So far, the for statement looks quite straight forward. You will see the complications when we turn our attention to the intricacies of the three sections of the for statement next.

8.4.3 Parts of a for loop

The initialization section

The initialization section allows you to specify code that will be executed at the beginning of the for loop. This code must belong to a category of statements that are called "**expression statements**". Expression statements are expressions that can be used as statements on their own. These are: **assignment**, **pre/post increment/decrement expression**, **a method call**, and **a class instance creation expression** (e.g. using the new operator). Besides expression statements, this section also allows you to declare local variables.

Here are a few examples of valid expression statements in a for loop:

```
int i = 0;  
for(i = 5;  
  
     i<10; i++); //assignment
```

```
Object obj;  
for(obj = "hello";  
  
     i<10; i++); //assignment
```

```
int i = 0;  
int k = 0;  
Object obj = "";  
for(i = 0, k = 7, obj = "hello";  
  
     i<10; i++); //multiple assignments
```

```
int k = 0;  
for(++k;  
  
     i<10; i++); //pre-increment
```

```
for(new ArrayList(); i<10; i++); //instance creation
```

```
int i = 0;  
for(System.out.println("starting the loop now");
```

```
i<10; i++); //method call  
  
for(k++, i--, new String()  
;i<10; i++); //multiple expressions
```

Observe that there doesn't need to be any relationship between the variable used in the initialization section and the variable used in other sections of a for loop.

The following are a few examples of valid local variable declarations:

```
for(int i = 5;  
    i<10; i++); //single variable declaration  
  
for(int i = 5, k = 7;  
    i<10; i++); //multiple variable declaration
```

You can only declare variables of one type. So the following is **invalid** :

```
for(int i = 5, String str = "";  
    i<10; i++); //invalid
```

Redeclaring a variable is also invalid:

```
int i = 0;  
for(int i = 5;  
    i<10; i++){ //invalid because i is already declared}
```

Another important point to note here is that a variable declared in the initialization section is visible only in the for loop. Thus, the following **will not compile** because i is not visible outside the loop.

```
for(int i = 5;  
    i<10; i++){  
    System.out.println("i is "+i);  
}  
  
System.out.println("Final value of i is "+i); //this line will not compile
```

The condition section [↳](#)

This one is simple. No, really :) You can only have an expression that returns a **boolean** or **Boolean** in this section. If there is no expression in this section, the condition is assumed to be **true**.

The updation section [↳](#)

The rules for the updation section are the same as the initialization section except that you cannot have any declarations here. This section allows only "expression statements", which I have already discussed above. Generally, this section is used to update the loop variable (**i++** or **k = k + 2** and so on) but

as you saw in examples above, this is not the only way to use it. There doesn't need to be any relationship between the code specified here and in other sections. The following are a few valid examples:

```
int i = 0;
for(;i<10; i++
);
//post-increment

for(;i<10; i = i + 2
);
//increment by two

for(;i<10; i = someRef.getValue()
)
//assignment

for(Object obj = new Object(); obj != null; ) { //empty
    updatation section

    System.out.println(obj);
    obj = null;
}

for(int i = 0; i<10; callSomeMethod()
);
//method call
```

Observe that there is no semi-colon after the expression statement. It is

terminated by the closing parenthesis of the for statement.

An infinite for loop [↳](#)

Based on the above information, it should now be very easy to analyze the following loop:

```
for( ; ; ) ;
```

The initialization section is empty. The condition section is empty, which means it will be assumed to be **true**. The updation section is empty. The loop code is also empty. There is nothing that makes the condition section to evaluate to **false** and therefore, there is nothing to stop this loop from iterating forever.

Now, as an exercise, try analyzing the following loop and determine its output:

```
boolean b = false;
for(int i=0 ; b = !b ; ) {
    System.out.println(i++);
}
```

8.5 Create and use for each loops

8.5.1 The enhanced for loop [↳](#)

Motive behind the enhanced for loop [↳](#)

In professional Java programming, looping through a collection of objects is a very common requirement. Here is how one might do it:

```
String[] values = { "a", "bb", "ccc" };
for(int i = 0; i<values.length; i++){
```

```
    System.out.println(values[i]); //do something with each value  
}  
  
The above code iterates through an array but the same pattern can be used to iterate through any other collection such as a List. You can code this loop using a while or do-while construct as well.
```

Java designers thought that this is too much code to write for performing such a simple task. The creation of an iteration variable **i** and the code in the condition section to check for collection boundary is necessary only because of the way the for (or other) loops work. They have no purpose from a business logic point of view. All you want is a way to do something with each object of a collection. You don't really care about the index at which that element is inside that collection. Furthermore, some collections such as a Set have no notion of index. Here is the code that iterates through the elements of a Set just to give you an idea of how you may iterate though a collection of objects that does not support index based retrieval:

```
Set s = new HashSet();  
s.add("a");  
s.add("bb");  
s.add("ccc");  
  
Iterator it = s.iterator();  
  
while(it.hasNext() ){  
    Object value = it.next();  
    System.out.println(value); //do something with each value  
}  
  
The above code has to create an Iterator object, which has no relation to the
```

business logic, to iterate through the elements. Such code that is not required by the business logic but is required only because of the way a programming language works is called "**boilerplate**" code. Such code can be easily eliminated if a programming language provides higher level constructs that internalize the logic of this code.

In Java 5, Java designers simplified the iteration process by doing exactly that. They introduced a new construct that internalizes the creation of iteration variable and the boundary check and called it the "**enhanced for loop**" or the "**for-each loop**".for each

The enhanced for loop [↳](#)

Let me first show how the two code snippets given above can be simplified and then I will get into the details:

```
String[] values = { "a", "bb", "ccc" };
for(String value : values){
    System.out.println(value); //do something with each
value
}
```

and

```
Set s = new HashSet();
s.add("a");
s.add("bb");
s.add("ccc");

for(Object value : s){
    System.out.println(value); //do something with each
value
```

```
}
```

Isn't that simple? It reads easy too - "for **each** **value** in **values** do ..." and "for **each** **value** in **s** do ...".

8.5.2 Syntax of the enhanced for loop [↳](#)

The syntax of the enhanced for loop is as follows:

```
for( Type variableName : array_or_Iterable ){
    statement(s);
}
```

or if there is only one statement in the for block:

```
for( Type variableName : array_or_Iterable ) statement
;
```

Type is the type of the elements that the array or the collection contains, **variableName** is the local variable that you can use inside the block to work with an element of the array or the collection, and **array_or_iterable** is the array or an object that implements `java.util.Iterable` interface.

I know that I have been using the word "collection" up till now and suddenly I have switched to **Iterable**. The reason for the switch is that `java.util.Iterable` is a super interface of `java.util.Collection` and although for-each loop is used mostly to iterate over collections, technically, it can iterate through an object of any class that implements `java.util.Iterable`. In other words, any class that wants to allow a user to iterate through the elements that it contains must implement `java.util.Iterable` interface.

The **Iterable** interface was introduced in Java 1.5 specifically to denote that an object can be used as a target of the for-each loop. It has only one method

named **iterate**, which returns a `java.util.Iterator` object. Since `java.util.Collection` extends `java.util.Iterable`, all standard collection classes such as `HashSet`, and `ArrayList` can be iterated over using the enhanced for loop.

For the purpose of the exam, you don't need to worry about the **Iterable** or the **Iterator** interface but it is a good idea to develop a mental picture of the relationship between the Iterable and the Iterator interfaces. Always remember that for-each can only be used to "**iterate**" over an object that is "**Iterable**".

Note that `java.util.Iterator` itself is not a collection of elements and does not implement `java.util.Iterable`. Therefore, an Iterator object cannot be a target of a for-each loop. Thus, the following code will not compile:

```
Iterator it = myList.iterator();
for(Object s : it){ //this line will not compile
}
```

For the purpose of the OCP Java 11 Part 1 exam, you only need to know that you can use the for-each loop to iterate over any collection such as a **List** and an **ArrayList**.

8.5.3 Enhanced for loop in practice [🔗](#)

Using enhanced for loop with typified collections [🔗](#)

In the code that I showed earlier, I used Object as the type of the variable inside the loop. Since I was only printing the object out I didn't need to cast it to any other type. But if you want to invoke any type specific method on the variable, you would have to cast it explicitly like this:

```
List names = //get names from somewhere

for(Object obj : names){
    String name = (String) obj;
    System.out.println(name.toUpperCase());
}
```

The true power of the enhanced for loop is realized when you use generics (which were also introduced in Java 5, by the way) to generify the collection that you are trying to iterate through. Here is the same code with generics:

List<String>

```
names = //get names from somewhere
```

```
for(String name : names){
    System.out.println(name.toUpperCase());
}
```

Although the topic of generics is not on the OCP Java 11 Part 1 exam, you will see questions on foreach that use generics. You don't need to know much about generics to answer the questions, but you should be aware of the basic syntax. I will discuss it more while talking about List and ArrayList later.

Counting the number of iterations ↗

In a regular for loop, the iteration variable (usually named i) tells you which iteration is currently going on. There is no such variable in a foreach loop. If you do want to know about the iteration number, you will need to create and manage another variable for it. For example:

```
List<String> names = //get names from somewhere
```

```
int i = 0;
```

```
for(String name : names){  
    i++;  
  
    System.out.println(i+" : "+name.toUpperCase());  
}  
System.out.println("Total number of names is "+i);
```

8.6 Use break and continue

8.6.1 Terminating a loop using break ↗

A loop doesn't always have to run its full course. For example, if you are iterating through a list of names to find a particular name, there is no need to go through the rest of the iterations after you have found that name. The break statement does exactly that. Here is the code:

```
String[] names = { "ally", "bob", "charlie", "david" }  
;  
for(int i=0; i<names.length; i++){  
    System.out.println(names[i]);  
    if("bob".equals(names[i])){  
        System.out.println("Found bob at "+i);  
        break;  
    }  
}
```

The output of the above code is:

```
ally  
bob  
Found bob at 1
```

Observe that **charlie** and **david** were not printed because the loop was broken after reaching **bob**.

It doesn't matter which kind of loop (i.e. while, do-while, for, or enhanced for) it is. If the code in the loop encounters a break, the loop will be broken immediately. The condition section and the updation section (in case of a for loop) will not be executed anymore and the control will move to the next statement after the loop block.

8.6.2 Terminating an iteration of a loop using **continue** [↑](#)

The **continue** statement is a little less draconian than the **break** statement. The purpose of a **continue** statement is to just skip the rest of the statements in the loop while executing that particular iteration. In other words, when a loop encounters the continue statement, the remaining statements within the loop block are skipped. The rest of the steps of executing a loop are performed as usual, i.e., the control moves on to updation section (if it is a for loop) and then it checks the loop condition to decide whether to execute the next iteration or not.

This is useful when you don't want to execute the rest of the statements of a loop after encountering a particular condition. Here is an example:

```
String[] names = { "ally", "bob", "charlie", "david" };  
  
for(String name : names){ //using a for-each loop this time  
  
    if("bob".equals(name)){  
        System.out.println("Ignoring bob!");  
        continue;  
    }  
    System.out.println("Hi "+name+"!");  
}
```

This code produces the following output:

```
Hi ally!  
Ignoring bob!  
Hi charlie!  
Hi david!
```

Observe that the print statement saying **Hi** was not executed only when the name was **bob**.

Just like the **break** statement, the **continue** statement is applicable to all kind of loops.

Both of these statements are always used in conjunction with a conditional statement such as an if/if-else. If a continue or a break executes unconditionally then there would be no point in writing the code below a continue or a break. For example, the following code will fail to compile:

```
String[] names = { "ally", "bob", "charlie", "david" }  
;  
  
for(int i=0; i<names.length; i++){  
    continue; //or break;  
  
    System.out.println("Hi "+names[i]+"!"); //This line  
    //will never get to execute  
  
}
```

The compiler will complain that the print statement is unreachable.

8.7 Nested loops

8.7.1 Nested loop

A nested loop is a loop that exists inside the body of another loop. This is not a big deal if you realize that the body of a loop is just another set of statements. Among these set of statements, there may also be a loop statement. For example, while looping through a list of Strings, you can also loop through the characters of each individual String as shown in the following code:

```
String[] names = { "ally", "bob", "charlie", "david" };

for(String name : names) {
    int sum = 0;
    for(int i=0; i<name.length(); i++) {
        char ch = name.charAt(i);
        int letterNumber = ch - 96; //converts an 'a' to 1, 'b' to 2, etc.
        sum = sum + letterNumber;
    }
    System.out.println("Lucky number for "+name+" is "+sum);
}
```

The above code nests a regular for loop inside an enhanced for loop. The following is the output produced by this code:

```
Lucky number for ally is 50
Lucky number for bob is 19
Lucky number for charlie is 56
Lucky number for david is 40
```

One thing that you need to be very careful about when using nested loops is managing the loop variables correctly. Consider the following program that is supposed to calculate the sum of all values in a given multidimensional array of ints:

```
int[][] values = { {1, 2, 3}, { 2, 3}, {2 }, { 4, 5,
6, 7 } };

int sum = 0;
for(int i = 0; i<values.length; i++) {
    for(int j=0; j<values[i].length; i++) {
```

```

        sum = sum + values[i][j];
    }
}
System.out.println("Sum is "+sum);

```

Read the above code carefully and try to determine the output.

It actually throws an exception: `Exception in thread "main"`
`java.lang.ArrayIndexOutOfBoundsException: 4`

Observe that the inner for loop is incrementing `i` instead of incrementing `j`. Now, let's execute the code step by step:

Step 0 : Before the outer loop starts, `values.length` is 4 and sum is 0.

Step 1 : Outer loop is encountered - `i` is initialized to 0 , `i<values.length` is true because 0 is less than 4 , so, the loop is entered.

Step 2 : Inner loop is encountered - `j` is initialized to 0 , `j<values[i].length` is true because `i` is 0, therefore, `values[i]` refers to { 1, 2, 3} and `values[i].length` is 3 . Therefore, the loop is entered.

Step 3 : Loop body is executed. `sum` is assigned `0 + values[0][0]` , i.e., 1 . `sum` is now 1 .

Step 4 : Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to 1 .

Step 5 : Inner loop's condition `j<values[i].length` is executed and it evaluates to true because `j` is still 0 and `i` is 1 , so, `values[i]` refers to {2, 3} and `values[i].length` is 2 . Thus, second iteration of the inner loop will start.

Step 6 : Loop body is executed. `sum` is assigned `1 + values[1][0]` , i.e., 1+2 . `sum` is now 3 .

Step 7 : Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to 2 .

Step 8 : Inner loop's condition `j<values[i].length` is executed and it evaluates to true because `j` is still 0 and `i` is 2 , so, `values[i]` refers to {2} and `values[i].length` is 1 . Thus, third iteration of the inner loop will start.

Step 9 : Loop body is executed. `sum` is assigned `3 + values[2][0]` , i.e., 3+2 . `sum` is now 5 .

Step 10 : Inner loop body is finished. Inner loop's updation section is executed, so `i` is incremented to `3` .

Step 11 : Inner loop's condition `j < values[i].length` is executed and evaluates to `true` because `j` is still `0` and `i` is `3` , so, `values[i]` refers to `{4, 5, 6, 7}` and `values[i].length` is `4` . Thus, fourth iteration of the inner loop will start.

Step 12 : Loop body is executed. `sum` is assigned `5 + values[3][0]` , i.e., `5 + 4` . `sum` is now `9` .

Step 13 : Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to `4` .

Step 14 : Inner loop's condition `j < values[i].length` is executed. Now, here we have a problem. `i` is `4` and because the length of the `values` array is only `4` , `values[4]` exceeds the bounds of the array and therefore `values[i]` throws an `ArrayIndexOutOfBoundsException` .

This is a very common mistake and while in this case the exception message made it easy to find, such mistakes can actually be very difficult to debug as they may produce plausible but incorrect results. That is why, although there is no technical limit to how many levels deep you can nest the loops, most professionals avoid nesting more than two levels.

In the exam you will be presented with questions containing two levels of loops. Many students ask if there is an easy way to figure out the output of such loops. Unfortunately, there is no short cut. It is best if you execute the code step by step while keeping track of the variables at each step as shown above on a piece of paper. The code in the exam will be tricky and it may trip you up even if you are an experienced programmer.

As an exercise, try executing the same code given above after changing `i++` to `j++` .

8.7.2 breaking out of and continuing with nested loops [🔗](#)

You can use break and continue statements in nested loops exactly like I showed you earlier with single loops. They will let you break off or continue with the loop in which they are present. For example, check out the following code that tries to find "bob" in multiple groups of names:

```
String[][] groups = { { "ally", "bob", "charlie" } , {  
    "bob", "alice", "boone"}, { "chad", "dave", "elliot"  
} };  
  
for(int i = 0; i<groups.length; i++){  
    for(String name : groups[i]){  
        System.out.println("Checking "+name);  
        if("bob".equals(name)) {  
            System.out.println("Found bob in Group "+i);  
            break;  
        }  
    }  
}
```

The following is the output of the above code:

```
Checking ally  
Checking bob  
Found bob in Group 0  
Checking bob  
Found bob in Group 1  
Checking chad  
Checking dave  
Checking elliot
```

When the inner loop encounters the String "bob" , it executes break, which causes the inner loop to end immediately. The control goes on to execute the next iteration of the outer loop, i.e., it starts looking for "bob" in the next group of names. This is the reason why the above output does not contain "Checking charlie", "Checking alice", and "Checking boone" .

Now what if you want to stop checking completely as soon as you find "bob" in any of the groups? In other words, what if you want to break out of not just the inner loop but also out of the outer loop as soon as you find "bob"? Java allows you to do that. You just have to specify which loop you want to break out of. Here is how:

```
String[][] groups = { { "ally", "bob", "charlie" } ,  
                      { "bob", "alice", "bo  
one" },  
                      { "chad", "dave", "el  
liot" } } ;  
  
MY_OUTER_LOOP:  
  
for(int i = 0; i<groups.length; i++){  
    for(String name : groups[i]){  
        System.out.println("Checking "+name);  
        if("bob".equals(name)) {  
            System.out.println("Found bob in Group "+i);  
            break MY_OUTER_LOOP;  
        }  
    }  
}
```

The following is the output of the above code:

```
Checking ally  
Checking bob  
Found bob in Group 0
```

I just "**labeled**" the outer for loop with **MY_OUTER_LOOP** and used the same label as the target of the break statement. This is known as a "**"labeled break"**". A "**"labeled continue"**" works similarly.

Let me give a quick overview of labels and then I will get back to the rules of using labeled break and continue.

What is a label?

A **label** is nothing but a name that you generally give to a **block of statements** or to statements that allow block of statements inside them (which means if, for, while, do-while, enhanced for, and switch statements). The exam does not test you on the precise rules for applying labels or naming of labels, but here are few examples:

```
SINGLE_STMT: System.out.println("hello");

HELLO: if(a==b) callM1(); else callM2();

COME_HERE : while(i=>0) {
    System.out.println("hello");
}

SOME_LABEL: { //ok, because this is a block of statements

    System.out.println("hello1");
    System.out.println("hello2");
}
```

Here are some examples of invalid usage of labels:

BAD1 : int x = 0; //can't apply a label to declaration.

BAD2 : x++; //can't apply a label to expressions

```
BAD3 : public void m1() { } //can't apply a label to methods
```

Although not necessary, the convention is to use capital letters for label names. Also, applying a label to a statement doesn't necessarily mean that you have to use that label as a target of any break or continue. You can label a statement just for the sake of it :)

Rules for labeled break and continue

Getting back to using labeled break and continue, you need to remember that if you use a labeled break or a labeled continue, then that label must be present on one of the nesting loop statements within which the labeled break or continue exists. Thus, the following code will fail to compile:

```
LABEL_1 : for(String s : array) System.out.println(s); //usage of LABEL_1 is valid here.
```

```
for(int i = 0; i<10; i++){  
    if(i ==2) continue LABEL_1; //usage of continue is invalid because LABEL_1 does not appear on a loop statement that contains this continue.  
}
```

The following is invalid as well:

```
for(int i = 0; i<10; i++){
```

```

LABEL_1 : if(i ==2) System.out.println(2); //usage o
f LABEL_1 is valid here.

for(int j = 0; j<10; j++){
    if(i ==2) continue LABEL_1; //usage of continue i
s invalid because LABEL_1 does not appear on a loop st
atement that contains this continue.

}

```

But the following is valid because the continue statement uses a label that nests the continue statement.

```

LABEL_1 : for(int i = 0; i<10; i++){
    if(i ==2) System.out.println(2);
    for(int j = 0; j<10; j++){
        if(i ==2) continue LABEL_1; //usage of continue i
s valid because it refers to an outer loop.

}

```

You can use labeled break and continue for non-nested loops as well but since unlabeled break and continue are sufficient for breaking out of and continuing with non-nested loops, labels are seldom used in such cases.

It is actually possible to use a labelled break (but not a labelled continue) inside any block statement. The block doesn't necessarily have to be a loop block. For example, the following code is valid and prints **1 3**.

```

public static void main (String[] args) {
    MYLABEL: {
        System.out.print("1 ");

```

```

        if( args != null) break MYLABEL;
        System.out.print("2 ");
    }
    System.out.print("3 ");
}

```

However, you need not spend time in learning about weird constructs because the exam doesn't test you on obscure cases. The **break** and **continue** statements are almost always used inside loop blocks and that is what the exam focuses on.

To answer the questions on break and continue in the exam correctly, you need to practice executing simple code examples on a piece of paper. The following is one such code snippet:

```

public static void doIt(int h){
    int x = 1;
    LOOP1 : do{
        LOOP2 : for(int y = 1; y < h; y++ ) {
            if( y == x ) continue;

            if( x*x + y*y == h*h){
                System.out.println("Found "+x+" "+y);
                break LOOP1;
            }
        }
        x++;
    }while(x<h);
}

```

Execute the above code mentally or on a piece of paper and find out what will it print when executed with 5 as an argument and then 6 as an argument.

8.8 Comparing loop constructs

8.8.1 Comparison of loop constructs

As you saw before, there are only a few technical differences between the four kind of loops. The **for**, **foreach**, and **while** loops are conceptually a little different than the **do-while** loop due to the fact that a **do-while** loop always executes at least one iteration. The **foreach** loop is a little different than other loops due to the fact that it can be used only for arrays and for classes that implement `java.util.Iterable` interface.

Other than these differences, they are all interchangeable. For example, you can always replace a **while** loop with a **for** loop as follows:

```
while( booleanExpression ){
}
for( ; booleanExpression ; ){
}
```

You can also replace a **foreach** loop with a **for** loop as shown below:

```
for( Object obj : someIterable){
}
for( Iterator it = someIterable.iterator() ; it.hasNext() ; ){
    Object obj = it.next();
}
```

Of course, the foreach version is a lot simpler than the for version and that is the point. While you can use any of the loops for a given problem, you should use the one that results in code that is most understandable or intuitive.

A general rule of thumb is to use a for loop when you know the number of iterations before hand and use a while loop when the decision to perform the

next iteration depends on the result of the operations performed in the prior iteration. For example, if you want to print "hello" ten times, use a for loop, but if you want to keep printing "hello" until the user enters a secret code as a response, use a while loop!

8.9 Exercise

1. Initialize a triangular matrix of ints using a for loop such the each element contains an value equal to the sum of its row and column index. Do the same using a while loop.
2. Write a method that determines whether a given number N is a prime number or not by dividing that number with all the numbers from 2 to N/2 and checking the remainder.
3. Use nested for loops to print a list of prime numbers from 2 to N.
4. Use nested for loops to print out each element of the array referred to by _3D in the format `[i][j][k] = N`:

```
int[] _1D1 = new int[]{1, 2, 3};  
int[][] _2D1 = new int[][]{ _1D1 };  
int[][] _2D2 = new int[][]{ _1D1, _1D1 };  
int[][][] _3D = new int[][][]{ _2D1, _2D2 };
```

5. Write a method that accepts a String as input and count the number of spaces in the string from start to until it finds an '`x`' , or if there is no '`x`' in the string, till end.
6. The following code contains a mistake that is quite common while using nested for loops. Identify the problem, fix it and print out all the elements of chars array.

```
String[][] chars = new String[2][];  
chars[0] = new String[2];  
chars[1] = new String[4];  
char cha = 97;  
for(char c=0;c<chars.length; c++){
```

```
    for(char ch=0;ch<chars.length; ch++){
        chars[c][ch] = ""+cha;
        cha++;
    }
}
```

What will happen if `char[0]` is initialized as `new String[1]` instead of `new String[2]` ?

7. To avoid the possibility of inadvertently introducing the mistake shown in the above code, a programmer decided to use for-each loops instead of the regular for loops:

```
for(String[] chara : chars){
    for(String s : chara){
        s = ""+cha;
        cha++;
    }
}
```

Is this a good idea?

8. Use an enhanced for loop to print alternate elements of an array. Can you use an enhanced for loop to print the elements in reverse order?
9. Given two arrays of same length and type, copy the elements of the first array into another in reverse order.

Chapter 9 Creating and Using Arrays

- Declare, instantiate, initialize and use a one-dimensional array
- Declare, instantiate, initialize and use multi-dimensional arrays

9.1 Declare, instantiate, initialize and use a one-dimensional array

9.1.1 Declaring array variables [↳](#)

An array is an object that holds a fixed number of values of a given type. You can think of an array as a carton of eggs. If you have a carton with 6 slots, then that carton can hold only six eggs. Each slot of the carton can either have an egg or can be empty. Observe that the carton itself is not an egg. Similarly, if you have an array of size six of `int` values (or ints, for short), then that array can hold six ints but the array itself is not an `int`.

An array of a given type cannot hold anything else except values of that type. For example, an array of ints cannot hold long or double values. Similarly, you can have an array of references of any given type. For example, if you have an array of Strings, this array can only hold references to String objects. An important point to note here is that even though we call it an "array of strings", it does not actually contain String objects. It contains only references to String objects. You cannot really have an array that contains actual objects.

Array declaration [↳](#)

When you declare an array variable, you are basically specifying the type of values that you want to deal with through that variable. The way to specify that in Java is to apply `[]`, i.e., square brackets to the type of the values. For example, if you want a variable through which you will deal with `int` values,

you will write `int[]`. Arrays can be multi-dimensional and there will be one set of opening and closing brackets for each dimension. For example,

```
int i; //i is an int
```

```
int[] ia1, ia2; //ia1 and ia2 are one dimensional arrays of ints
```

```
int[][] iaa; //iaa is a two dimensional array of ints and so on
```

An array declaration can never include the size of the array. Thus, the following are declarations will not compile:

```
int[2] invalid1;  
int[3][] invalid2;  
int[][][4] invalid3;
```

Unfortunately, the above method is not the only way to declare arrays. Java allows you to apply square brackets to the variable name instead of type name as well. For example,

```
int i, ia[]; //i is an int but ia is a one dimensional array of int values
```

```
int[] ia, iaa[]; //ia is a one dimensional array of ints but iaa is a two dimensional array of ints and so on
```

Observe that the rule of thumb of one set of square brackets per dimension still holds. In the case of `iaa`, you have one set applied on the type and one set

applied on the variable, therefore `iaa` is a two dimensional array.

Arrays of objects are declared the same way. For example,

```
Object[] obja, objaa[]; //obja is a one dimensional array of Objects but objaa is a two dimensional array of Objects
```

```
String[] strA; //strA is a one dimensional array of Strings
```

Note that the statements shown above only declare array variables. They don't actually create arrays. Array creation and initialization is a topic in its own right and that is what I will discuss next.

9.1.2 Creating and initializing array objects [↳](#)

Creating arrays using array creation expressions

You use the `new` keyword to create an array object. For example,

```
int[] ia = new int[10]; //an array of ints of size 10
```

```
boolean[] ba = new boolean[3]; //an array of booleans of size 3
```

```
String[] stra = new String[5]; //an array of Strings of size 5
```

```
MyClass[] myca = new MyClass[5]; //an array of MyClass of size 5
```

```
int[] invalid = new int[]; //missing size. will not compile.
```

The parts on the right-hand side of = sign in the above statements are called "**array creation expressions**" . These expressions merely allocate space to hold **10 ints , 3 booleans , 5 string references , and 5 MyClass references** respectively. Every element of the array is also initialized to its default value automatically by the JVM. The default values of array elements are very straightforward - references are initialized to **null** , numeric primitives to **0** , and booleans to **false** . In the above lines of code, **ia** is set to point to an array of ten int values and each element of the array is initialized to 0, **ba** is set to point to an array of three boolean values and each element of the array is initialized to **false** , **stra** is set to point to an array of five **String** references and each element of the array is initialized to **null** , and finally, **myca** is set to point to an array of five **MyClass** references and each element of the array is initialized to **null** . Observe that all the elements of an array are initialized to the same value. This shows us another important aspect of arrays - that arrays can have **duplicate values** .

It is important to understand that, in the above statements, we are not creating instances of the class of the array elements. In other words, we are not creating instances of **String** or instances of **MyClass** . (We are not creating instances of ints or booleans either, for that matter, but since they are primitives, and since primitives are not objects, they don't have instances anyway.)

In Java, arrays, whether of primitives or objects, are objects of specific classes. In other words, an array object is an instance of some class. It is not an instance of Object class but since Object is the root of every class in Java, an array object is an Object and all methods of the Object class can be invoked on an array. Let us now look at the following program and its output to know more about the class of the above defined array objects -

```
public class TestClass{
    public static void main(String[] name){
        int[] ia = new int[10];
```

```

boolean[] ba = new boolean[3];
String[] stra = new String[5];
TestClass[] ta = new TestClass[5];
System.out.println(ia.getClass().getName() + " , "+i
a.getClass().getSuperclass().getName());
System.out.println(ba.getClass().getName() + " , "+b
a.getClass().getSuperclass().getName());
System.out.println(stra.getClass().getName() + " , "
+stra.getClass().getSuperclass().getName());
System.out.println(ta.getClass().getName() + " , "+t
a.getClass().getSuperclass().getName());
}
}

```

Output:

```

[I , java.lang.Object
[Z , java.lang.Object
[Ljava.lang.String; , java.lang.Object
[LTestClass; , java.lang.Object

```

The output shows that `ia`, which is declared to be of type `int[]`, is not an instance of `int` but of a class named `[I`. `stra`, which is declared to be of type `String[]`, is not an instance of `String` but of a class named `[Ljava.lang.String` and so on. These names of the classes look weird. Actually, Java cooks up the name of the class of an array by looking at the number of dimensions and the type of the elements. For each dimension, there is one opening square bracket. This is followed by a letter for the class of the elements as per the following table and, if the array is not of a primitive, the name of the class followed by a semi-colon.

Type	Letter
boolean	Z
byte	B
char	C
short	S
int	I

long	J
float	F
double	D
any Object	L

Based on the above table, the name of the class for `long[][]` (i.e. a two dimensional array of longs) would be `[[J` and the name of the class for `mypackage.SomeClass[][][]` (i.e. a three dimensional array of `mypackage.SomeClass`) would be `[[[Lmypackage.SomeClass;` .

Although the above discussion about the class of arrays is not included in the exam objectives, a few test takers have reported getting a question on the exam that requires this information.

Creating arrays using array initializers

In the previous section, we created array objects using the new keyword. It is possible to create array objects without using the new keyword. For example, the arrays that we created above can also be created as follows:

```
int[] ia = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; //an array of ints of size 10
```

```
boolean[] ba = {true, false, false}; //an array of booleans of size 3
```

```
String[] str = {"a", "b", "c", "d", "e"}; //an array of Strings of size 5
```

```
MyClass[] myca = { new MyClass(), new MyClass(), new MyClass(), new MyClass(), new MyClass() } ; //an array of MyClass objects of size 5
```

The parts on the right-hand side of the = sign in the above statements are called "**array initializers**" . An array initializer is a shortcut that allows you to create as well as initialize each element of the array with the values that you want (instead of the default values that you get when you use array creation expressions).

Since the compiler can find the type of the elements of the array by looking at the declaration (i.e. the left-hand side of the statement), specifying the same on the right-hand side is not required. Similarly, the compiler finds out the length of the array as well by counting the number of values that are specified in the initializer.

Array initializers can also be mixed with array creation expressions. For example:

```
int[] ia = new int[]{ 1, 2, 3, 4, 5 };
```

Observe that size is missing from the expression on the right-hand side. Java figures out the size of the array by counting the number of elements that are specified in the initialization list. In fact, it is prohibited to specify the size if you are specifying individual elements. Therefore, the following is invalid:

```
int[] ia = new int[2]{ 1, 2 }; //will not compile.
```

9.2 Using arrays

9.2.1 Array indexing [🔗](#)

In Java, array indexing starts with 0. In other words, the index of the first element of an array is 0. For example, if you have an array variable named **accounts** that refers to an array of 100 **Account** objects, then you can access the first object through **accounts[0]** and the 100th object through **accounts[99]** . **accounts[0]** or **account[99]** are like any other variable of type **Account** .

Similarly, if you have an array variable **ia** pointing an array of 5 ints, the first

element can be accessed using `ia[0]` and the last element using `ia[4]`.

java.lang.ArrayIndexOutOfBoundsException ↗

If you try to access any array beyond its range, JVM will throw an instance of `ArrayIndexOutOfBoundsException`. For example, if you have `int[] ia = new int[3];` the statements `int i = ia[-1];` and `int i = ia[3];` will cause an `ArrayIndexOutOfBoundsException` to be thrown.

Do not worry if you don't know much about exceptions at this point. I will discuss exceptions in detail later.

Arrays of length zero ↗

As strange as it may sound, it is possible to have an array of length 0. For example, `int[] ia = new int[0];` Here, `ia` refers to an array of ints of length 0. There are no elements in this array. It is important to understand that an array of length 0 is not the same as `null`. `ia = null` implies that `ia` points to nothing. `ia = new int[0];` implies that `ia` points to an array of ints whose length is 0.

A good example of an array of length 0 is the `args` parameter of the main method. If you run a class with no argument, `args` will **not** be `null` but will refer to an array of Strings of length 0.

Changing the size of an array ↗

Java arrays are always of fixed size(or length). Once you create an array, you cannot change the number of elements that this array can have. So, for example, if you have created an array of 5 ints and if you have 6 int values to store, you will need to create a new int array. There is no way to increase or decrease the size of an existing array. You may change the array variable to point to a different array, and you can, of course, change the values that an array contains.

9.2.2 Members of an array object [↑](#)

We saw earlier that arrays are actually objects of specific classes. We also saw the names of some of these classes. But what do these classes contain? What are the members of these classes? What functionality do these classes provide? Let us take a look:

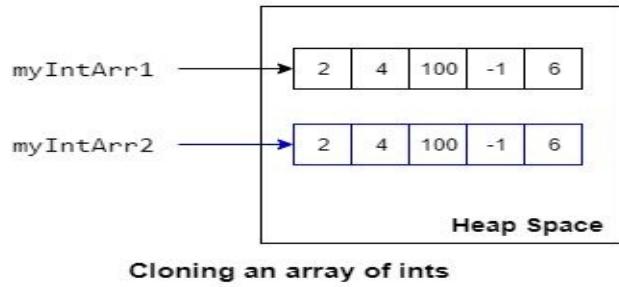
Array length

All array classes have one field named **length** , which is of type **int** . This field is **public** and it stores the length of the array. This field is also **final** , which reflects the fact that you cannot change the length of an array after its creation. Since the length of an array can never be less than 0, the value of this field can never be less than 0 either.

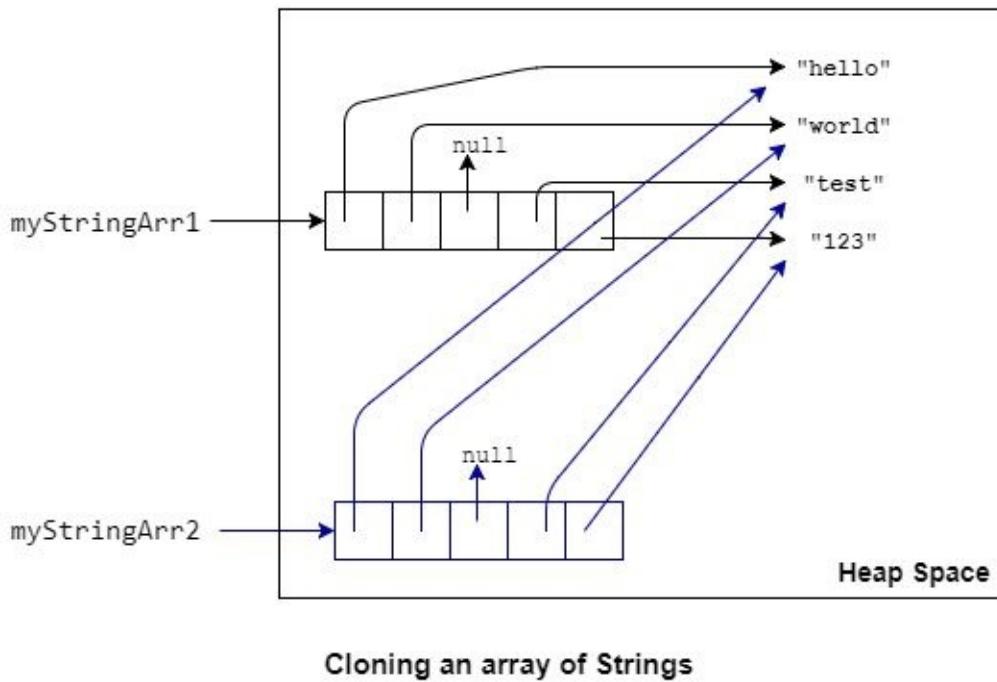
Array cloning

All array classes also have a public method named **clone** . This method creates a copy of the array object. Note that it doesn't create copies of the objects referred to by the array elements. It merely creates a new array object of the same length and copies the contents of existing array into the new array. Which means, if the existing array contained primitive values, those values will be copied to the elements of the new array. If the existing array contained references to objects, those references will be copied to the elements of the new array. Thus, the elements of the new array will also point to the same objects. This is also known as "**shallow copy**" .

For example, the following figure shows exactly what happens when you clone an array of five ints referred to by the variable `myIntArr1` using the statement `int[] myIntArr2 = (int[]) myIntArr1.clone();` ; Don't worry about the explicit cast for now, but observe that a new array object with the same number of slots is created and the contents of the slots of the existing array are copied to the slots of the new array.



The following figure shows what happens when you clone an array of Strings referred to by the variable `myStringArr1` using the statement `String[] myStringArr2 = (String[]) myStringArr1.clone();`; Observe that the references are copied into the new array and that no new String objects are created in the heap space.



Cloning is not required for the exam but it is an important aspect of arrays anyway.

Members inherited from Object class

Remember that since `java.lang.Object` is the root class of all classes, it is

the root class of all array classes as well and therefore, array classes inherit all the members of the **Object** class. This includes **toString**, **equals**, and **hashCode** methods.

9.2.3 Runtime behavior of arrays [🔗](#)

Although arrays are the simplest of data structures, they are not without their quirks. To use arrays correctly and effectively, you must be aware of the two most important aspects of arrays in Java.

The first is that they are "**reified**". Meaning, the type checking of arrays and its elements is done at runtime by the JVM instead of at compile time by the compiler. In other words, the type information of an array is preserved in the compiled bytecode for use by the JVM during run time. The JVM knows about the type of an array and enforces type checking on arrays.

For example, if you have an array of Strings, the JVM will not let you set any element of that array to point to any object other than a String, even though the compiler may not notice such a violation as illustrated by the following code:

```
String[] sa = { "1", "2", "3" };
Object[] oa = sa;
oa[0] = new Object();
```

The above code will compile fine. The compiler has no objection when you try to assign an array of Strings to a variable of type array of Objects. It has no objection when you try to set an element of this array of Strings to point to an object that is not a String. But when you run it, the JVM will throw a **java.lang.ArrayStoreException** on the third line. This is because the JVM knows that the array pointed to by **oa** is actually an array of Strings. It will not let you corrupt that array by storing random objects in it.

It is important to understand this concept because it is diametrically opposite to how generics work. This is the reason why arrays and generics operate well with each other. You will realize this when you learn about generics later.

The second thing about arrays is that they are "**covariant**" . Meaning, you can store a subclass object in an array that is declared to be of the type of its superclass. For example, if you have an array of type `java.lang.Number` , you can store `java.lang.Integer` or `java.lang.Float` objects into that array because both are subclasses of `java.lang.Number` . Thus, the following code will compile and run fine:

```
Number[] na = { 1, 2, 3 };
na[0] = new Float(1.2f);
```

We haven't seen anyone getting a question on array reification and covariance in the OCP Java 11 Part 1 exam. However, any discussion about arrays is incomplete without these two points and that is why I have talked about them here.

9.2.4 Uses of arrays [↳](#)

Arrays are quite powerful as a **data structure** but they are somewhat primitive as a **data type** . As we saw earlier, arrays have only one field and merely a couple of methods. But because of their simplicity, arrays are used as building blocks for other data types and data structures. For example, the `String` data type is built upon an array of `char` s. So are `StringBuffer` and `StringBuilder` . These higher level classes really only wrap an array of characters and provide methods for manipulating that array.

Arrays are also used extensively for building higher level data structures such as `List` , `Stack` , and `Queue` . You may have come across the `ArrayList` class (this class is also on the exam, by the way, and I will discuss it later). Guess what, it is a `List` that manages its collection of objects using an array inside. Since these classes provide a lot of additional features on top of arrays, more often than not, it is these classes that get used in application programs rather than raw arrays.

java.util.Arrays class [↳](#)

Java standard library does include a utility class named **Arrays** in package **java.util** that makes working with raw arrays a little easier.

java.util.Arrays class contains a large number of static utility methods for manipulating any given array object. Although not mentioned explicitly in the part 1 exam objectives, it is a good idea to browse through the JavaDoc API description of Arrays class to know about these methods. We have seen questions in the exam on the **compare** and **mismatch** methods that were added to this class in Java 9, so, I will discuss these two methods in detail.

Arrays.compare and Arrays.mismatch methods [🔗](#)

The **compare** and **mismatch** methods are static methods and are meant to compare two arrays of the same types. Although they can compare arrays of primitives as well as arrays of reference types, they are most useful for comparing primitive arrays. Efficient comparison of primitive arrays is a common requirement in data processing. Since these methods compare data in chunks, they are able to perform substantially better than custom code that compares elements individually. That is probably why the OCP Java 11 exam has questions on these methods.

If you browse through the API JavaDoc, you will notice a large number of variants for these methods. Although these methods are heavily overloaded, practically, there are only two flavors that you need to understand for the purpose of the exam. The rest are mere replicas; one for each data type.

Before getting into the details of the methods, you need to understand the meaning of the two words that are used often in the description for these methods.

1. Lexicographical - This simply means "dictionary order", i.e., the way in which words are ordered in a dictionary. If you were to determine which one of the two words would come before the other in a dictionary, you would compare the words letter by letter starting from the first position. For example, the word "**children**" will appear before (and can be considered smaller) than the word "**chill**" because the first four letters of the two words are the same but the letter at the fifth position in "**children**" (i.e. **d**) comes before (or is smaller than) the letter at the fifth position in "**chill**" (i.e. **l**). Observe that lengthwise, the word "**children**" is bigger than "**chill**" but length is not

the criteria here. For the same reason, an array containing **12345** will appear before an array containing **1235**.

2. Prefix - This refers to the common part of the two arrays. In the above example, "**chil**" can be called the prefix while comparing "**children**" and "**chill**". Similarly, **123** is the prefix while comparing **12345** and **1235**.

Let's check out the JavaDoc descriptions of compare and mismatch methods now.

public static int compare(int[] a, int[] b) : Compares two int arrays lexicographically. If the two arrays share a common prefix then the lexicographic comparison is the result of comparing two elements, as if by Integer.compare(int, int), at an index within the respective arrays that is the prefix length. Otherwise, one array is a proper prefix of the other and, lexicographic comparison is the result of comparing the two array lengths.

A null array reference is considered lexicographically less than a non-null array reference. Two null array references are considered equal.

It returns 0 if the first and second array are equal and contain the same elements in the same order; a value less than 0 if the first array is lexicographically less than the second array; and a value greater than 0 if the first array is lexicographically greater than the second array.

For example, the following code will print -1 because the first array has smaller integer at index 2.

```
int[] ia1 = { 0, 1, 2, 6};  
int[] ia2 = { 0, 1, 5};  
System.out.print(Arrays.compare(ia1, ia2)); //prints -1
```

Observe that the returned value (-1) does not depend on the amount of difference between the two elements (2 - 5 is -3).

public static int mismatch(int[] a, int[] b) : Finds and returns the index of the

first mismatch between two int arrays, otherwise return -1 if no mismatch is found. The index will be in the range of 0 (inclusive) up to the length (inclusive) of the smaller array.

If the two arrays share a common prefix then the returned index is the length of the common prefix and it follows that there is a mismatch between the two elements at that index within the respective arrays. If one array is a proper prefix of the other then the returned index is the length of the smaller array and it follows that the index is only valid for the larger array. Otherwise, there is no mismatch.

Two non-null arrays, **a** and **b** , share a common prefix of length **p1** if the following expression is **true** :

```
p1 >= 0 &&
p1 < Math.min(a.length, b.length) &&
Arrays.equals(a, 0, p1, b, 0, p1) &&
a[p1] != b[p1]
```

Note that a common prefix length of 0 indicates that the first elements from each array mismatch.

Two non-null arrays, **a** and **b** , share a proper prefix if the following expression is true:

```
a.length != b.length &&
Arrays.equals(a, 0, Math.min(a.length, b.length),
             b, 0, Math.min(a.length, b.length))
```

For example, the following code will print 2 because the arrays differ at index 2:

```
int[] ia1 = { 0, 1, 2, 6};
int[] ia2 = { 0, 1, 5};
System.out.println(Arrays.mismatch(ia1, ia2)); //prints 2
```

Observe that 2 is also the length of the prefix, which is {0, 1} .

You should go through the JavaDoc API description of the various overloaded compare and mismatch methods. You will notice that they all work in the same basic fashion.

9.3 Declare, instantiate, initialize and use multi-dimensional arrays

9.3.1 Multidimensional Arrays [↳](#)

The phrase multidimensional array brings a picture of a matrix to mind. But it is important to understand that Java doesn't have matrix kind of multidimensional arrays. What Java has is arrays whose elements themselves can be arrays. Recall that, in Java, every array object is an object of a specific class. For example, the class of an **array of ints** is [\[I](#). Now, what if you want to have an array of objects of this class. In other words, an **array of "array of ints"**. You can declare it like this:

```
int[][] iaa;
```

Visually, the declaration looks like `iaa` is a two-dimensional matrix of ints. But in reality, `iaa` points to a single dimensional array, where each element of the array is an array of ints. There is a fundamental difference between the two approaches. In a matrix, the number of elements in a given dimension are the same at each index of the higher dimension. For example, in a two dimensional array, the length of each row will always be the same, i.e., each row will have same number of columns. While in an array of arrays, there is no such restriction. Each row can refer to an array of any length. This is illustrated in the following figure:

iaa[2x3]

xxxx	1110
...	
1110	1
1111	2
1112	3
1113	4
1114	5
1115	6
1116	
...	

iaa[2][]

xxxx	1111
xxxx+1	1200
1111	1
1112	2
1200	3
1201	4
1202	5

In this figure, **iaa[2x3]** , which is a made up syntax (Java does not have this), is a matrix of size 2x3, i.e., 2 rows and 3 columns. Since the size of each dimension is known in advance, it can be easily stored in a contiguous chunk of $2 \times 3 = 6$ memory cells. You can also locate the address of any element using a simple formula.

But in case of an array of array of ints, you can only allocate 2 continuous memory addresses to store two references - one for each array of ints. These two references can, in turn, point to two arrays of different lengths. In the above figure, **iaa[0]** points to an array stored at location 1111, and **iaa[1]** points to another array stored at location 1200. **iaa[0]** points to an array of length 2, while **iaa[1]** points to an array of length 3. This is what Java has. Since the array of arrays are not required to be symmetric, such arrays are called "**jagged arrays**". FYI, C# supports both kinds of arrays, i.e., symmetric as well as jagged.

Keeping the above discussion in mind, let us now look at the rules of declaring and creating an array of arrays in Java:

1. The type of an array is determined by the number of pairs of square brackets applied to the variable. For example, in case of **int[] ia;** and **int ia[];** ia is an array of ints. In case of **int[] iaa[];** and **int iaa[][];** iaa is an array of arrays of ints. **int[][] iaaa[];** is an array of arrays of arrays of ints.

2. You never specify the length of the array in the type declaration. Thus, **int[3] ia;** and **int[2][] iaa;** are invalid declarations.

3. The rules of array creation expressions and array initializers that we talked about in the previous lesson, are applicable here as well.

The following are a few examples:

1. `int[][] iaa = new int[2][3];`

`iaa` is created using an array creation expression. `iaa` refers to an array of length 2. Each element of this array refers to an array of ints of length 3. Each element of both the arrays of ints is initialized to 0.

2. `int[][] iaa = new int[3][];`

`iaa` is created using an array creation expression. `iaa` refers to an array of length 3. Can you guess what each of the three elements of this array are initialized to? Observe that the type of each element is "array of ints", which means `iaa` is an array of objects (and not of primitives). Since every element of array of objects is automatically initialized to `null`, each element of the array pointed to by `iaa` is initialized to `null`. You can make them point to arrays of ints like this:

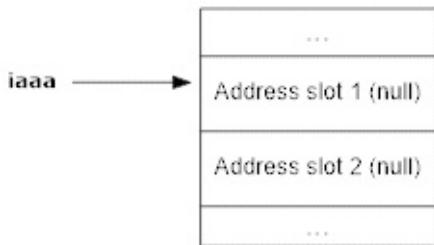
```
iaa[0] = new int[2]; //iaa[0] points to an array of  
ints of length 2  
iaa[1] = new int[3]; //iaa[1] points to an array of  
ints of length 3
```

Each element of these two arrays is now initialized to 0 but `iaa[2]` is still `null`.

This example illustrates another important aspect of creating arrays. Notice the difference between the specification of dimension sizes of example 1 and this one. Example 1 has `[2][3]`, while here, it is `[3][]`. We have omitted the size of the second dimension. The size of the first dimension tells the JVM how many references you want to store in your array. (Remember that the type of those references that you want to store is array of ints, i.e., `int[]`). `[3]` implies that you want to store three references. In other words, the length of your array (which is of type array of ints) is 3. However, the size of the second dimension is not needed because the arrays pointed to by those references can be created later and can be of

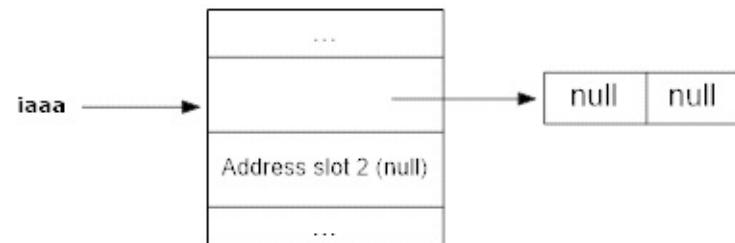
different lengths. You will need to specify their lengths only when you create them.

What happens when you create a three dimensional array of ints, i.e., an array of array of ints? The process is same. You only need to tell the JVM how many number of references (whose type will be array of array of ints, i.e., `int[][][]`) do you want to keep in your array. If you want only 2 such references, you can create it with `new int[2][][]`. The following three figures illustrates what happens in each step as you initialize an array of array of ints.



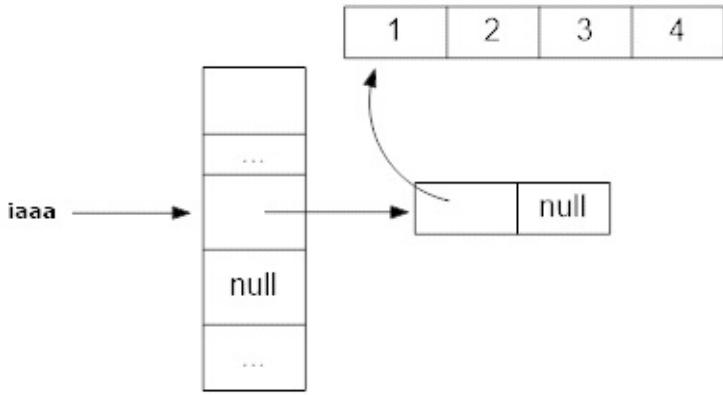
Step 1 - create space to store iaaa.

You can now set the first slot of this array to point to an array of length 2 using `iaaaa[0] = new int[2][];`



Step 2 - create space to store two references of type
array of array ints.

Note that `iaaaa[1]` is still null. Next, you can set `iaaaa[0][0]` to point to an array of 4 ints using `iaaaa[0][0] = new int[]{ 1, 2, 3, 4};` or `iaaaa[0][0] = { 1, 2, 3, 4};` (You can also do `iaaaa[0][0] = new int[4];` and in that case the value of each element will 0).



Step 3 - create space to store array 4 ints.

You cannot, however, leave out the size of a higher dimension if you want to specify the size of a lower dimension. For example, you cannot do **new int[][][2];** The reason is simple - **new int[][][2]** tries to create an array of **int[2]** objects. But it does not tell the JVM how many **int[2]** objects you want to store. Without this information, the JVM has no idea how much space it needs to allocate for this array. On the other hand, **new int[2][]** is fine because now, you are telling the JVM that you want to create an array of length 2. In this case, the JVM is clear that it needs to allocate space to store 2 references. Remember that the size of a reference doesn't depend on the length of the array to which it points. So, the JVM doesn't care about the length of the arrays to which these two references will refer. It simply allocates space to store 2 references.

3. **int[][] iaa = new int[][]{ new int[]{ 1, 2 } };** This statement uses an array creation expression coupled with array initializer.

int[][] iaa = { { 1, 2 } }; This is the same as above but with array initializer.

In both the cases, **iaa** refers to an array of length 1. The first and only element in this array refers to an array of ints of length 2.

4. **Object[] obj[] = { {"string is also an object"}, {null} , { new Object() , new Integer(10)} };**
obj refers to an array of array of objects. The length of the array is 3. The first array refers to an array of objects of length 1. The only element of this

array of objects points to a String that contains "string is also an object". The second array refers to an array of length 1. The only element of this array points to `null`. The third array refers to an array of length 2. The first element of this array points to an object of class Object and the second element points to an object of class Integer.

Here is a simple program that prints out useful information from an array of arrays. I suggest you play around with it by changing the arrays.

```
public class TestClass
{
    public static void main(String[] args) {
        Object[] iaa[] = { {"string is also an object"
}, {null} , { new Object() , new Integer(10)} };
        for(int i=0; i<iaa.length; i++){
            System.out.println("Element["+i+"] contains
an array of length "+iaa[i].length);
            for(int j = 0; j<iaa[i].length; j++){
                System.out.println("    Element["+i+"]["+j
+"] contains "+iaa[i][j]);
            }
        }
    }
}
```

The output is:

```
Element[0] contains an array of length 1
    Element[0][0] contains string is also an object
Element[1] contains an array of length 1
    Element[1][0] contains null
Element[2] contains an array of length 2
    Element[2][0] contains java.lang.Object@2a139a55
    Element[2][1] contains 10
```

As an exercise, modify this code to print out the contents of an array of array of array of `int`s.

You will see questions in the exam that require you to iterate through all the elements of multi-dimensional arrays using a for loop as well as while/do-while loop.

9.3.2 Assigning arrays of primitives to Object variables [↳](#)

I explained earlier that every array is an object. This means, you can assign any array object to variable of type **Object**. Like this:

```
int[] intArray = new int[]{ 0, 1, 2 };  
Object obj = intArray;
```

This is valid because an array of **int** s is an **Object**.

Then how about this - **Object[] oa = intArray;** ? This will not compile because elements of the array pointed to by **intArray** are not **Object** s. They are **int** s. Therefore, you cannot assign an array of **int** s to a variable of type array of **Object** s.

You need to be very clear about this concept because it gets confusing very quickly as the following example shows:

```
Object[] oa = new int[2][3]; //this is valid.
```

```
Object[][] oaa = new int[2][3]; //this will not compile.
```

Of course, each element of the array created using **new int[2][3]** is an array of **int** s. An array of **int** s is an **Object** and therefore, an array of array of **int** s is an array of **Object** s.

9.4 Exercise

1. Create a array of booleans of length 3 inside the main method. Print the elements of the array without initializing the array elements explicitly. Observe the output.
2. Given `int[] first = new int[3];`, `int[] second = {};`, and `int[] third = null;`, print out the length of the three arrays and print out every element of the three arrays.
3. Create an array of chars containing four values. Write assignment statements involving the array such that the first element of the array will contain the value of the second element, second element will have the value that was there in the third element. and third element will contain the value of the fourth element.
4. Declare and initialize an array of length 4 of type array of Strings without using the new keyword such that no two arrays of Strings have the same length. Print the length of all of the arrays one by one (including the length of the two dimensional array).
5. Given the statement `String[][] names = new String[2][3];` How many Strings will you need to fill up names completely? Initialize each element of names with a non-null String. Print each of those values one by one without using a loop. Do the same using nested for loops after going through the chapter on loops.
6. Define two variables of type array of Strings. Initialize them using the elements of the array created in the previous exercise.
7. Define a simple class named `Data` with a public instance field named `value` of type `int`. Create and initialize a `Data` variable named `d` in `TestClass`'s main. Create an array of `Data` of length 3 and initialize each of its elements with the same `Data` instance. Use any of the array elements to update the `value` field of the `Data` object. Print out the `value` field of the `Data` object using the three elements of the array. Finally, print the `value` field of the original `Data` using the variable `d`.
8. Define and initialize an array of array of ints that resembles a triangular matrix of integers.
9. Declare another array of array of ints and initialize it using the elements of the array created in the previous exercise in such a way that it resembles an inverted triangular matrix of integers.
10. Declare and initialize a variable of type array of Objects of length 3. Initialize the first element of this array with an array of ints, second with an

array of array of ints, and third with an array of Objects. See if any of the assignments fails compilation.

11. Given the statement `int[][][] nums = new int[1][3];`, how many `int` values can `nums` store? Write down how each element of `nums` can be addressed.
12. Given the following code:

```
int[][][][] nums = new int[1][4][2];
for(int i = 0; i<nums.length; i++){
    for(int j = 0; j<nums[i].length; j++){
        for(int k = 0; k<nums[i][j].length; k++){
            nums[i][j][k] = i + j + k;
            System.out.println("num["+i+"]["+j+"]["+k+
] = "+nums[i][j][k]);
        }
    }
}
```

Execute it mentally and write down its output on paper. Run the code and check your answer.

Chapter 10 Creating and Using Methods

- Create methods and constructors with arguments and return values
- Create and invoke overloaded methods
- Apply the static keyword to methods and fields

10.1 Create methods with arguments and return values

10.1.1 Creating a method

We have already seen methods in previous chapters. Indeed, we have been using the "main" method to run our test programs all along. But we haven't discussed them formally yet.

A **method** is what gives **behavior** to a type. Java allows classes, interfaces, and enums to have a behavior. A behavior is nothing but a high level action performed by a piece of code. If you expect this action to be performed as and when required, you put this piece of code in a method and give it a name. This lets you invoke that action whenever you need by using the name. It is like a black box to which you give some input and get back some output in return.

Thus, the basic structure of a method is as follows:

```
returnType methodName(parameters) {  
    methodBody  
}
```

A **ReturnType** specifies what the method returns as a result of its execution.

For example, a method that returns the sum of two integers may specify that it returns an `int` as a result. A method that returns an `Account` object may specify `Account` as the return type. If a method doesn't return anything, it must specify so, using the keyword `void`.

It is important to know that a method can return one thing at the most.

A **MethodName** is the name given to this method. It must be a valid Java identifier. Remember that an identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. It cannot be a reserved word or a keyword.

The **Parameters** of a method are specified by a list of comma-separated parameter declarations. Each parameter declaration consists of a type and an identifier that specifies the name of the parameter. If a method does not take any parameter, the parameters list will be empty. The following are a few examples of a valid parameter list:

```
void save() //no parameters
```

```
void saveAccount(Account acct) //takes an Account as a parameter
```

```
void add(int a, int b) //takes two ints as parameters
```

Note that, unlike regular declarations, each parameter in a parameter list must be specified along with its type individually. Thus, while `int a, b;` is a valid statement on its own that declares two `int` variables, `add(int a, b)` is an invalid declaration of method parameter `b`.

Each parameter in the parameter list may also be declared as final if the code in the method does not change its value. For example:

```
void add(int a, final  
        int b) //b is final.
```

If a parameter is declared final and if the method body tries to change its value, the code will fail to compile.

The **MethodBody** contains the code that is to be executed upon invocation of that method. It must be contained within curly brackets. Observe that this is unlike other block statements such as if/else, for, and while, where you can omit the curly brackets if there is just one statement in their body. In the case of methods, curly brackets are required even if the body consists of just one statement.

There are several other bells and whistles associated with methods such as varargs, accessibility, static, abstract, final, and exception handling. I will discuss all of these as we go along.

10.1.2 Returning a value from a method [🔗](#)

A method must always return a value of the type that it promises to return in its declaration after successful completion. This means that it is not possible to return a value conditionally as shown in the following code:

```
public int get2X(int x){  
    if(x>0) return 2*x;  
}
```

The compiler will generate an error message saying, "error: missing return statement" because it notices that the method will not return anything if the if condition is false. This makes sense if you consider what will happen to the caller of the get2X method if the method doesn't return anything. For example, what value should be assigned to **y** by this statement - **int y = get2X(-1);** ? There is no good answer. Thus, the method must return an **int**

in all situations (except when it throws an exception, but you can ignore that for now).

This rule applies even to methods that return a reference type (and not just to method that return a primitive). But if the return type of a method is a reference type, it is ok for the method to return `null` because `null` is a valid value for a reference. In other words, returning `null` is not the same as returning nothing. Thus, the following method is fine:

```
String getValue(int x){  
    if(x > 0) return "good day!";  
    else return null; //this is ok.  
  
}
```

The only situation where a method can avoid returning a value is if it ends up throwing an exception, which means that the method didn't really finish successfully and therefore, it cannot be expected to return a value!

Method returning void

If a method says that it doesn't return anything (i.e. its return type is specified as `void`), then it must not return anything in any situation. Obviously, it cannot even return `null` because as we saw above, `null` is not the same as nothing. But the interesting thing is that it cannot even return `void`. Thus, the following method will not compile.

```
void doSomething(){  
    System.out.println("hello");  
    return void; //invalid  
  
}
```

There are only two options in this case - do not have any return statement at all

or have an empty return statement, i.e., `return;` For example:

```
void doSomething(){
    System.out.println("hello");
    return; //empty return

}
```

or

```
void doSomething(){
    System.out.println("hello");
    //no return statement at all

}
```

Returning values of different types from a method [↳](#)

The general rule is that Java does not allow a method to return a value that is of a different type than the one specified in its declaration. This means that if a method says that it returns an `int`, it cannot then return a `boolean` value. However, there are three exceptions to this rule. Two are about primitives and one is about references.

1. **Numeric promotion** - If the return type of a method is a numeric type (i.e. `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), then the return value can be of any other numeric type as long as the type of the return value is smaller than the type of the declared return type. For example, if a method says it returns an `int`, it is ok for that method to return a `byte`, `short`, or `char` value. But it cannot return a `long`, `float`, or `double` value. Thus, the following code will compile fine:

```
public int getVal(int x){
```

```
char ch = 'a';
byte b = 0;
if(x>0) return ch;
else return b;
}
```

This is allowed because a smaller type can easily be promoted to a larger type without any loss of information.

2. **Autoboxing/Unboxing** - Java allows a return value to be a reference to a wrapper class if the return type is of a primitive type of the same or smaller type. Thus, the following code is ok:

```
public int getVal(){
    return new Integer(10); //wrapper object will
be unboxed into a primitive
}
```

The reverse is also allowed:

```
public Integer getVal(){
    return 10; //primitive will be boxed into a w
rapper object
}
```

3. The third exception is related to inheritance. Since I haven't discussed this topic yet, I will only mention it briefly here. Don't worry if you don't understand it completely at this point.

It is ok for a method to return a reference of a subtype of the type declared as its return type. This is called "covariant return types." For example, if a method declares that it returns an Object, it is ok for the method to return a String because a String is also an Object. The following code shows some valid possibilities:

```
Object getValue(){
    return "hello";
    //return 10; //This is ok, 10 will be boxed into
    //an Integer object, which is an Object.

    //return null;//This is ok too.

}
```

This is the same as promising someone that you will give them a fruit, and then give them a banana. This is ok because a banana is also a type of fruit. But the reverse is not true. If you promise someone that you will give them a banana, you cannot then give any other fruit. Thus, the following will fail to compile:

```
String getValue(){
    return new Object(); // will not compile because
    //an Object is not a String.

}
```

It will become clearer after you learn about inheritance.

These rules ensure that only those values that are "assignable" to the declared return type are returned by a method. In other words, if you can assign a particular value to a variable of the declared return type directly, then you can return that value from that method. For example, you can assign a `char` value to an `int` variable directly, therefore, you can return a `char` value from a method that declares that it returns an `int`.

Returning multiple values from a method ↗

Java does not allow a method to return more than one value. Period. This seems like difficult restriction to overcome if you want to return multiple values from a method. For example, what if your getName method wants to return first name and last name separately? Well, the way to do that in Java is to use a class to capture multiple values and then return a reference to an object of that class. Here is what your getName method may look like:

```
Name getName(){
    Name n = new Name(); //capture two values in a Name
object

    n.firstName = "Ann";
    n.lastName = "Rand";
    return n; //return a reference to the Name object

}
```

This code assumes the existence of a separate class called Name that can capture the two components of the name:

```
class Name{
    String firstName, lastName;
}
```

You may also use arrays to overcome this restriction. For example, the getName method can also be coded as follows:

```
String[] getName(){
    return new String[]{"ann", "rand"};
}
```

Generally, classes are designed to capture values that are related to each other. For example, you might have a Student class that captures a student's id, name, and

address. You may, however, encounter situations where you want to return multiple unrelated values from the same method. You can use the same approach to return these values. Classes that are used to capture unrelated values are called holder classes. If you encounter such situations too often in your code then it is a symptom of a bad design. A detailed discussion on this is beyond the scope of the exam.

10.1.3 Varargs

If you know the type of the arguments but don't know the exact number of arguments that the caller is going to pass to a method, you can put the arguments in an array and pass the array to the method. For example, if you want to write a method that computes the average of any number of integer values, you could do something like this:

```
public double average(int[] values){  
    /* by the way, can you tell what will happen if sum  
     * is declared as int?  
     * Expect questions in the exam that seem to be about one topic  
     * but are actually about something entirely different.  
    */  
  
    double sum = 0;  
    for(int i=0; i<values.length; i++) sum += values[i]  
;  
    return values.length==0?0 : sum/values.length;  
}
```

The caller of this method can put all the integer values in an array and call it as follows:

```
int[] values = { 1, 2, 3, 4 };  
double average = average(values);
```

This approach works fine but is a little tedious to write. Java 5 introduced a new syntax called "**varargs**" that makes passing a variable number of arguments to a method a little easier. Instead of using an array parameter in the method declaration, you use a varargs parameter by appending three dots to **int** like this:

```
public double average(int...  
    values){  
    //same code as shown earlier goes here  
}
```

There is absolutely no change in the method body. Within the method body, **values** remains an array of integers like before. The caller, however, does not need to create an array explicitly. It can simply pass any number of int arguments in the method invocation. Thus, the following are all valid method invocations of the new average method.

```
double average = average(); //no argument  
  
double average = average(1); //one argument  
  
double average = average(1, 2, 3, 4); //multiple arguments
```

An important point to understand here is that the varargs syntax is just a syntactic sugar for the developer. It saves a few keystrokes while typing the code but makes absolutely no difference to the resulting bytecode generated by the

compiler. When the compiler sees the invocation of a method with varargs parameter, it simply wraps the arguments into an array and passes the array to the method. Indeed, the old code that used an array to call the average method still works with the new average method. If you update the method code to print the number of elements in the values array (just add `System.out.println(values.length);`), you will see it print 0, 1, and 4 for the above three invocations respectively.

Observe that if you don't pass any argument, the compiler will create an array with a length of zero and pass that array. The method will not receive a `null` but an array of length zero in this case. This is unlike the other ways that you invoke the array version of the method. It is not possible to invoke the array version without any argument. If you don't want to pass any value, you will have to pass `null`. Try it out and see what happens.

You can apply the varargs approach to a parameter of any type and not just primitives.

Restrictions on varargs [↳](#)

Let us try to expand the usage of varargs from one parameter to two. What if we define a method as follows:

```
public double test(int... p1, int... p2){  
    System.out.println(p1.length+" "+p2.length);  
}
```

and call this method like as follows?

```
test(1, 2);
```

This poses a problem for the compiler. It has three equally valid possibilities for initializing `p1` and `p2`. It can create two int arrays containing `{1}` and `{2}`, or it can create `{1, 2}` and `{ }`, or it can create `{ }` and `{ 1, 2 }` and pass them as arguments for `p1` and `p2` respectively.

Resolving this ambiguity through complicated rules would make the varargs

feature too confusing to use and so, Java imposes the following two restrictions on varargs:

1. A method cannot have more than one varargs parameter.
2. The varargs parameter, if present, must be the last parameter in the parameter list of a method.

With the above two rules in place, it is easy to understand what will happen if you have a method as follows:

```
void test(int x, int... y){  
    //some code here  
}
```

and call it as follows:

```
test(1, 2); //x is assigned 1, y is assigned {2}  
  
test(1); //x is assigned 1, y is assigned {}  
  
test(1, 2, 3, 4); //x is assigned 1, y is assigned { 2,  
, 3, 4}
```

Note that since **x** is a non-varargs parameter, any invocation of the method **test** must include a value for **x**. Therefore, a call to **test()** with no argument will not compile.

10.2 Create overloaded methods

10.2.1 Method signature [🔗](#)

Before learning about method overloading, you need to know about something called "method signature".

A **method signature** is kind of an "id" of a method. It uniquely identifies a method in a class. A class may have several methods with the same name but it cannot have more than one method with the same **signature**. When you call a method of a class, you basically tell the compiler which method you mean to call by mentioning its signature. If a class has multiple methods with the same signature, the compiler will not be able to determine which method you mean and will raise an error.

The question then is: what exactly constitutes a method signature? Simple. Method signature includes just the **method name** and its **ordered list of parameter types**. Nothing else. For example, all of the following method declarations have the same signature:

1. `void process(int a, string str);`
2. `public void process(int value, String name);`
3. `void process(int a, String str) throws Exception;`
4. `String process(int a, String str);`
5. `private int process(int a, String data);`
6. `static void process(int a, String str);`

Observe that in all of the above cases, the method name (i.e. `process`) and the ordered list of parameter types (i.e. `int` , `String`) are exactly the same and their access types, static-instance types, return types, parameter names, and the `throws` clauses are all different. However, since these attributes are not part of the method signature, they do not make the methods different. The compiler will consider all of the above methods as having the same signature and will complain if you have any two of the above methods in the same class.

10.2.2 Method overloading

Now that you know about method signature, method overloading is a piece of cake. You know that a class cannot have more than one method with the same signature. But it can certainly have multiple methods with the same name and different parameter types because having different parameter types would make their signatures different. In such a situation where a class has multiple methods with same name, it is said that the class has "**overloaded**" the method name. The method name is overloaded in the sense that there are multiple possibilities associated with that name.

From the compiler's perspective, method overloading is nothing special. Since the compiler cares only about the method signatures, it makes no difference to the compiler whether two methods are different because of a difference in their method names, or their method parameters, or both. To the compiler, overloaded methods are just different methods.

However, method overloading does hold importance for the developer and the users of a class because it can either make the code more intuitive to use or make it totally confusing. For example, you have been using

`System.out.println` methods to print out all sorts of values to the console. You are able to use the same method named `println` for printing an `int` as well as a `String` only because there are multiple `println` methods that take different parameter types. Take a look at the JavaDoc API description of `java.io.PrintStream` class to see how many `println` methods it has. Wouldn't you be frustrated if you had to use `printlnByte` to print a `byte` , `printlnShort` to print a `short` , `printlnString` to print a `String` and so on? In this case, method overloading has certainly made your life easier.

Now, consider the following code:

```
public class TestClass{
    static void doSomething(Integer i, short s){
        System.out.println("1");
    }

    static void doSomething(int in, Short s){
        System.out.println("1");
    }
}
```

```
}

public static void main(String[] args){
    int b = 10;
    short x = 20;
    doSomething(b, x);
}
}
```

Any guess on what the above code prints? Don't worry, even experienced programmers will scratch their heads while figuring this one out. The answer is that it will not compile. But not because of the presence of two `doSomething` methods. The methods are fine. They have different signatures. The problem is with the call to `doSomething(b, x)`. The compiler is not able to determine which one of the two `doSomething` methods to use because both of them are equally applicable.

You will not get questions this hard in the exam. I showed you the above code only to illustrate how overloading can be misused to create horrible code.

10.2.3 Method selection [↳](#)

The code that I showed you in the previous section illustrates a problem with overloaded methods. When their parameters lists are not too different, it is very difficult to figure out which of the overloaded methods will be picked up for a method call. If the parameters are too similar, it may even become an impossible task.

Java specifies precise rules that are used by the compiler to disambiguate such method calls. While you will not get questions that are too complicated in the exam, you still need to know a few basic rules to figure out the simple cases. So, here they are -

1. The first rule is that having overloaded methods does not cause a compilation error by itself. In other words, as long as their method signatures are different, the compiler doesn't care whether they are too

similar or not. Compilation error occurs only if the compiler is not able to successfully disambiguate a particular method call.

2. **Exact match** - If the compiler finds a method whose parameter list is an exact match to the argument list of the method call, then it selects that method. For example, consider the following two methods:

```
void processData(Object obj){ }
void processData(String str){ }
```

and the method call `processData("hello")`. Since `String` is an `Object`, the `String` argument matches the parameter list of both the methods and so both the methods are capable of accepting the method call. However, `String` is an exact match to the argument and so the compiler will select the second method.

This rule applies to primitives as well. Thus, out of the following two methods, the first method is selected when you call `processData(10)`; because 10 is an int, which matches exactly to the parameter type of the first method even though the long version of the method is also perfectly capable of accepting the value.

```
void processData(int value){ }
void processData(long value){ }
```

3. **Most specific method** - If more than one method is capable of accepting a method call and none of them is an exact match, the one that is "**most specific**" is chosen by the compiler. For example, consider the following two methods:

```
void processData(Object obj){ }
void processData(CharSequence str){ }
```

and the method call `processData("hello")`. Remember that `String` extends `CharSequence` and `CharSequence` extends `Object`. Thus, a `String` is a `CharSequence` and a `String` is also an

`Object` . So here, neither of the methods has a parameter list that is an exact match to the type of the argument but both the methods are capable of accepting a `String` . However, between `Object` and `CharSequence` , `CharSequence` is more specific and so the compiler will select the second method.

It is important to understand what "**more specific**" means. It really just means closer or more similar to the type of the parameter being passed. A `String` is closer to a `CharSequence` than it is to an `Object` . Technically, a subclass (or a subtype) is always more specific than a super class (or supertype).

Since primitives are not classes, there is no subclass/superclass kind of relation between them as such but Java does define the subtype relation for them explicitly, which is as follows:

`double > float > long > int > char`

and

`int > short > byte`

What it means is, `float` is a subtype of `double` , `long` is a subtype of `float` , `int` is a subtype of `long` , and `char` is subtype of `int` . And also, `short` is a subtype of `int` and `byte` is a subtype of `short` .

Based on the above, you can easily determine which of the following two methods will be picked if you call `processData((byte) 10)` ;

```
void processData(int value){ }
void processData(short value){ }
```

The `short` version will be picked because `short` is a subtype of `int` and is therefore, more specific than an `int` .

Here is another interesting situation that can be explained easily based on the above subtype hierarchy of primitives. What will the following code print when compiled and run?

```

public class TestClass{
    public static void m(char ch){
        System.out.println("in char");
    }
    public static void main(String[] args) {
        byte b = 10;
        m(b);
    }
}

```

The above code will cause a compilation error saying, "error: incompatible types: possible lossy conversion from byte to char". Observe that `char` is not a supertype of `byte`. Therefore, the method `m(char ch)` is not applicable for the method call `m(b)`. This makes sense because even though `char` is a larger data type than `byte`, `char` cannot store negative values, while `byte` can.

4. **Consider widening before autoboxing** - Since autoboxing only came into existence when Java 5 was released, it is necessary to give higher priority to the primitive versions if the argument can be widened to the method parameter type so that existing code will keep working as before. Therefore, out of the following two methods, the `short` version will be picked instead of the `Byte` version if you call `processData((byte) b)`; even though the `Byte` version is an exact match with `byte` after autoboxing.

```
void processData(short value){ }
```

```
void processData(Byte value){ }
```

5. **Consider autoboxing before varargs** - This rule mandates that if an argument can be autoboxed into a method parameter type then that method be considered even if a method with varargs of the same type is available. This explains why if you call `processData(10)`; then the `Integer` version (and not the `int...` version) is picked out of the following two.

```
void processData(int... values){ }
```

```
void processData(Integer value){ }
```

Let us now try to figure out what happens if you call `processData((byte) 10);` with the same two methods available.

```
void processData(int... values){ }
```

```
void processData(Integer value){ }
```

Just apply the rules one by one:

- **Applying rule 1** - Requires no application.
- **Applying rule 2** - There is no method available whose parameter type matches exactly to `byte` because `byte` doesn't match exactly to `int...` i.e. `int[]` or `Integer`.
- **Applying rule 3** - There is no method available whose parameter type matches to `byte` after widening a `byte` to any of the types a `byte` can be widened to (i.e. `short`, `int`, `long` etc.). Note that a `byte` cannot be widened to `int[]`.
- **Applying rule 4** - There is no method available whose parameter type matches to `byte` after autoboxing it to `Byte`. Remember that a primitive type can only be autoboxed to the same wrapper type. So a `byte` can only be autoboxed to `Byte` and not to anything else such as `Short` or `Integer`.
- **Applying rule 5** - A `byte` can be widened to `int` and an `int` can be accepted by a method that takes `int...`. Therefore, the varargs version of the method will be invoked

Let me show you another example. Let's determine what will happen if we call `processData(10);` with the following two methods available:

```
void processData(Long value){ }  
void processData(Long... values){ }
```

Observe that `10` is an `int`. Do we have a `processData` method that takes an `int`? No. So Rule 2 about exact match doesn't apply. `int` is not a subtype of `Long` or `Long[]` so Rule 3 doesn't apply either. Let's see if Rule 4 is of any help. `10` can be widened to a `long`, `float`, or `double` but we don't have any method that takes any of these types. Finally, as per Rule 5, `10` can be boxed into an `Integer` but that won't work either because there is no method that takes an `Integer`. Remember that `processData(Long)` cannot accept an `Integer` because `Integer` is not a subtype of `Long`.

Well, we don't have anymore options left to try. This means the compiler can't tie this call to any method and will therefore, raise an error saying, "`Error: no suitable method found for processData(int)`".

Method selection is not a trivial topic. The Java language specification spends a considerable number of pages defining the rules of method selection. Since it is easy to get bogged down with all those rules, I have tried to simplify them so they are easy to remember. You will be able to answer the questions in the exam and will also be able to figure out what a piece of code does in real life using the four points I have mentioned above.

Just be aware that there are several situations that cannot be explained by these points. Most of them involve Generics, a topic that is not on the exam. You should go through relevant sections of JLS if you are interested in exploring this topic further.

You will most likely get only a single question on this topic. If you forget the above rules and find yourself taking too much time in figuring out the answer, I suggest you leave that question and move ahead. There is no point in wasting too much time on a question that is very easy to answer incorrectly.

10.3 Passing object references and primitive values into methods

10.3.1 Passing arguments to methods [🔗](#)

This section requires that you have a clear understanding of the difference between an object and a reference. I suggest you go through the "Object and Reference" section in "Kickstarter for Beginners" chapter and the "Difference between reference variables and primitive variables" section in the "Working with Java data types" chapter to refresh your memory before proceeding with this topic.

Pass by value [🔗](#)

To understand how Java passes arguments to methods, there is just one rule that you need to remember - Java uses **pass-by-value** semantics to pass arguments to methods. There are no exceptions. I mentioned this rule right at the beginning because if you keep this rule in your mind, this topic will feel like a piece of cake to you. You will never get confused with any code that the exam throws at you.

Let me start with the following simple code:

```
public class TestClass {  
  
    public static void main(String[] args){  
        int a = doubleIt(100);  
        System.out.println(a); //prints 200  
  
    }  
  
    public static int doubleIt(int x){  
        return 2*x;  
    }  
}
```

```
 }  
}
```

In the above code, the main method invoke the `doubleIt` method and passes the value `100` to the method. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `x` with the value that was passed, i.e., `100`. Thus, `x` now contains the value `100`. The return statement returns the value generated by the statement `2*x` back to the caller. The JVM assigns this value to the variable `a`. Thus, `a` now contains `200`. This is what is printed out.

Quite simple so far, right? Let me modify the code a bit now:

```
public class TestClass {  
  
    public static void main(String[] args){  
        int a = 100;  
  
        int b = doubleIt(a);  
        System.out.println(a+" , "+b);  
    }  
  
    static int doubleIt(int x){  
        return 2*x;  
    }  
}
```

The only thing that I have changed is that instead of passing the value `100` directly to the `doubleIt` method, I am passing the variable `a`. The JVM notices that the method call is using a variable as an argument. It takes the value contained in this variable (which is `100`) and passes it to the method. Rest is exactly the same as before. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `X` with the value that was passed, i.e., `100`. Thus, `X` now contains the value `100`. The return statement returns the

value generated by the statement `2*x` back to the caller. The JVM assigns this value to the variable `b`. Thus, `b` now contains `200`. Finally, `100, 200` is printed.

Observe that `doubleIt` has no knowledge of the variable `a`. It only gets the value contained in the variable `a`, i.e., `100`, which is assigned to `x`.

Are you with me so far? Alright, here comes the twist. Take a look at the following code:

```
public class TestClass {  
  
    public static void main(String[] args){  
        int a = 100;  
        int b = doubleIt(a);  
        System.out.println(a+" , "+b);  
    }  
  
    static int doubleIt(int x){  
        x = 200;  
        return 2*x;  
    }  
}
```

If you have understood the logic I explained earlier, you should be able to figure out what the above code prints. Let's follow the same process for analyzing what is happening here. The JVM notices that the method call is using a variable as an argument. It takes the value contained in this variable (which is `100`) and passes it to the `doubleIt` method. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `x` with the value that was passed, i.e., `100`. Thus, `x` now contains the value `100`. Next, `x` is assigned a new value `200` by the statement `x = 200;` In other words, the `x` is overwritten with a new value `200`. The point to understand here is that this assignment has no effect of the variable `a` that was used in the calling method because `doubleIt` has absolutely no idea where the original value of `100` that it was passed as an argument came from. It merely gets the value `100` as an argument. In other words, the JVM passed only the value `100` to the method and not the variable `a`.

Next, the return statement returns the value generated by the statement $2*x$ back to the caller, which is **400**. The JVM assigns this value to the variable **b**. Thus, **b** now contains **400**. At this time, you should realize that nothing was done to change the value of the variable **a** at all. It still contains the same value as before, i.e., **100**. Therefore, the print statement prints **100, 400**.

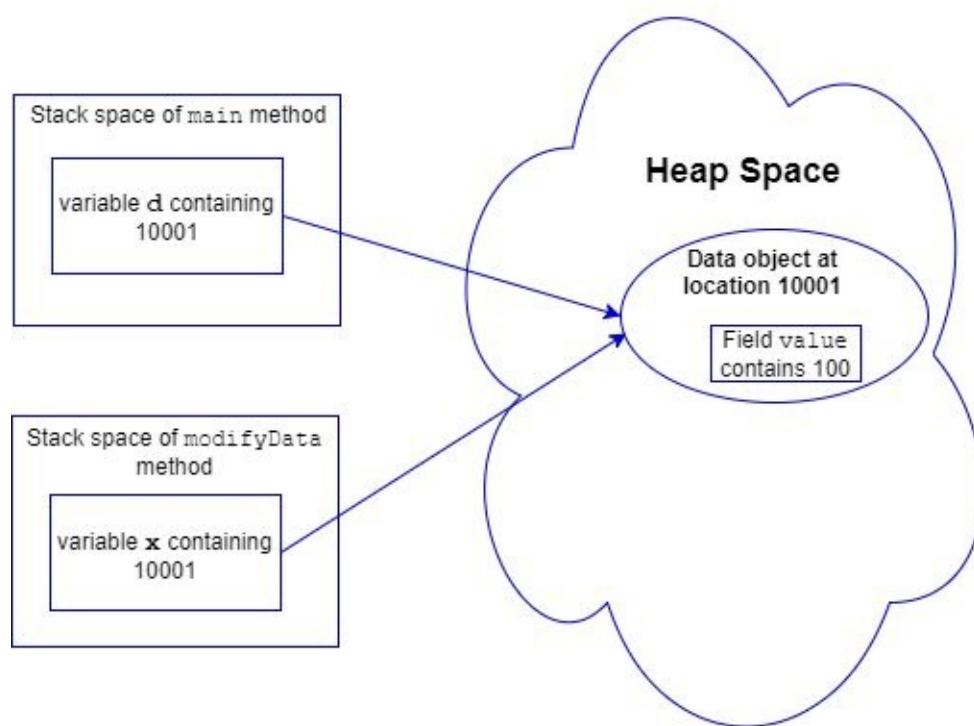
The above example also shows that it is not possible for a method to change the value of the variable that was passed as an argument by the caller because that variable is never sent to the method. Only its value is sent. That is why the term "**pass by value**" is used to describe parameter passing in Java.

10.3.2 Passing objects to methods

In the previous lesson, you saw how passing a primitive variable to a method works. Passing a reference variable works exactly the same way. Check out the following code:

```
class Data{  
    int value = 100;  
}  
  
public class TestClass {  
  
    public static void main(String[] args){  
        Data d = new Data();  
        modifyData(d);  
        System.out.println(d.value);//prints 200  
    }  
  
    public static void modifyData(Data x){  
        x.value = 2*x.value;  
    }  
}
```

In the above code, the main method creates a **Data** object and saves its reference in the variable **d**. It then calls the **modifyData** method. During execution, the JVM notices that the call to **modifyData** method uses a reference variable. Recall our discussion on reference variables in "Working with data types" chapter that a reference variable doesn't actually hold an object. It just holds the address of the memory location where the object resides. While invoking the **modifyData** method, the JVM copies the value stored in the variable **d** into the method parameter **x**. So, for example, if the **Data** object is stored at the memory address **10001**, the variable **d** contains **10001** and the JVM copies this value into the variable **x**. Thus, the variable **x** also now contains the same address as **d** and thus, in a manner of saying, variable **x** also starts pointing to the same object as the one pointed to by the variable **d**. This situation is illustrated in figure 1 below.



The point to note here is that even though it looks as if we passed the **Data** object created in **main** to **modifyData**, all we actually passed was the value stored in variable **d** (which was nothing but the address of the **Data** object) to **modifyData**. The **Data** object itself remained exactly at the same memory

location where it was. We didn't move it, pass it, or copy it, to anywhere.

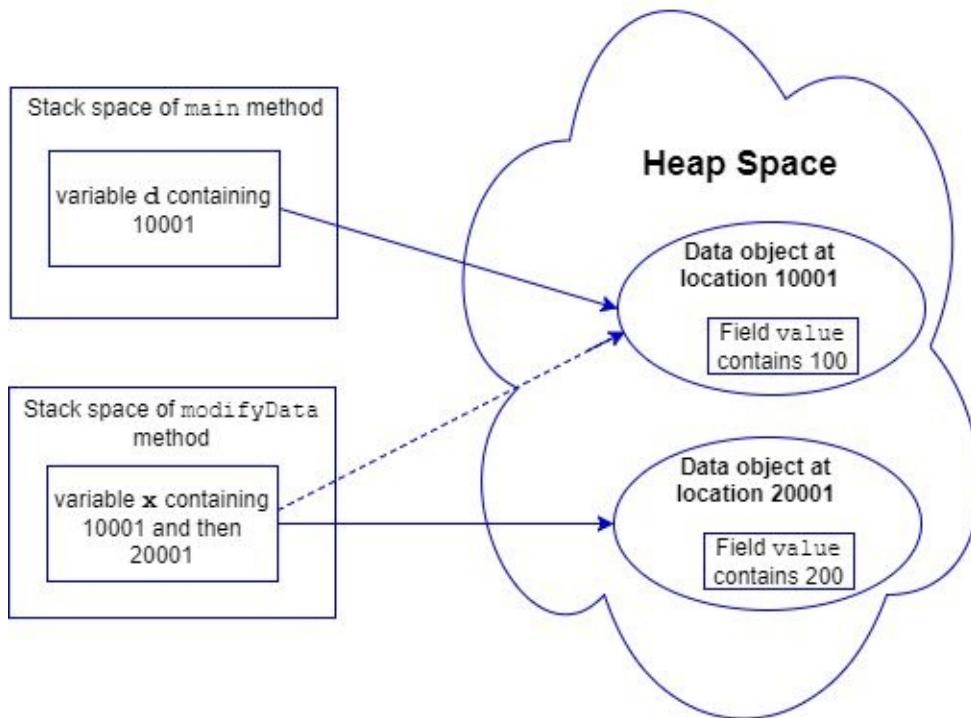
The `modifyData` method uses the reference variable `x` to modify the value field of `Data` object pointed to by `x` to `200`. When the control goes back to the main method, it prints `d.value`, which is now `200`.

Let us make a small modification to the `modifyData` method of the above code:

```
public static void modifyData(Data x){  
    x = new Data();  
  
    x.value = 2*x.value;  
}
```

All we have done is inserted the statement `x = new Data();`. Can you tell what the main method will print now?

When the JVM invokes `modifyData` method, it makes `x` point to the same `Data` object as the one pointed to by variable `d`. This part is the same as before. But the first statement of the updated method creates a new `Data` object and makes `x` point to this new object. For example, if this new `Data` object resides at address 20001, then `x` now contains 20001 (instead of 10001). The next statement updates the value field using the variable `x`. Since `x` now points to the new `Data` object, this update statement updates the value field of the new `Data` object instead of the old `Data` object. When the control goes back to the main method, it prints `d.value`. This prints `100` because we never modified the variable `d` and so, `d` still points to the original `Data` object. This is illustrated in figure 2 below.



In the two examples above, we passed a variable (the local variable **d**) as an argument to the **modifyData** method. Now, take a look at the following code:

```

class Data{
    int value = 100;
}

public class TestClass {

    public static void main(String[] args){
        modifyData(new Data());
    }

    public static void modifyData(Data x){
        x.value = 2*x.value;
    }
}

```

```
}
```

Observe the statement `modifyData(new Data());`. We are not passing any variable to the `modifyData` method here. So, are we really passing the `Data` object directly to the method? No, although we do not have any explicit variable to store the address of the `Data` object, the compiler creates a temporary reference variable implicitly. It creates the `Data` object in the heap space and assigns it to this temporary variable. While making the method call, it is the value of this variable that is passed to the `modifyData` method. Of course, since we have not saved a reference to this `Data` object in the `main` method, we won't be able to refer to this object after the `modifyData` method returns.

This technique is used all the time while printing messages to the console using the `print/println` method. For example, when you do `System.out.println("hello world");` the compiler passes the value of the temporary variable that points to the `String` object containing "hello world" to the `println` method.

10.3.3 Returning a value from a method [↑](#)

Just like parameters, returning a value from a method also uses "pass by value" semantics.

In the case of primitives, a return statement can return either the value directly (e.g. `return 100;` `return true;` `return 'a';` etc.) or the value of a variable (e.g. `return i;` where `i` is an `int` variable).

In the case of objects, a return statement can either return the value of an explicit variable (e.g. `return str;` where `str` is a `String` variable) or the value of an implicit temporary variable that references the object (e.g. `return "hello";` `return new Student();` etc.). In both cases, it is really the address where the object is stored that is returned.

What the caller does with the return value is irrelevant to the method that returns a value. The caller can either use the return value further in the code or ignore the return value altogether.

10.4 Create and overload constructors

10.4.1 Creating instance initializers [🔗](#)

When you ask the JVM to create a new instance of a class, the JVM does four things:

1. First, it checks whether that class has been initialized or not. If not, the JVM loads and initializes the class first.
2. Second, it allocates the memory required to hold the instance variables of the class in the heap space.
3. Third, it initializes these instance variables to their default values (i.e. numeric and char variables to **zero**, boolean variables to **false**, and reference variables to **null**).
4. And finally, the JVM gives that instance an opportunity to set the values of the instance variables as per the business logic of that class by executing code written in special sections of that class. These special sections are: **instance initializers** and **constructors**.

It is only after these four steps are complete that the instance is considered "ready to use". Remember the use of the **new** operator to create instances of a class? The JVM performs the all of the four activities mentioned above and only then returns the reference of the newly created and initialized object to your code.

Out of the four activities listed above, you have already seen the details of the first one in the previous section. The second two activities are performed transparently by the JVM. They don't require the programmer to do anything. The fourth activity, which is the subject of this section, depends on the programmer because it involves the code written by the programmer.

Let us take a look at **instance initializers** first because these are the ones that are executed by the JVM first.

Creating instance initializers ↗

Instance initializers are blocks of code written directly within the scope of a class. Here is an example:

```
class TestClass{  
    {  
        System.out.println("In instance initializer");  
    }  
}
```

Observe that there is no method declaration or anything but just a line of code nested inside the opening and closing curly brackets. Code inside an instance initializer block is regular code. There is no limitation on the number of statements or the kind of statements that an instance initialize can have. You may have any number of such instance initializer blocks in a class. The JVM executes them in the order that they appear in the class. The following code, for example, prints **Hello World!** using two instance initializers in different places in a class:

```
class TestClass{  
    //first instance initializer  
  
    {  
        System.out.print("Hello ");  
    }  
  
    public static void main(String[] args){  
        new TestClass();  
    }  
}
```

```
//second instance initializer

{
    System.out.print("World!");
}

}
```

Observe that the main method does nothing except create an instance of TestClass. As a result of this creation, the JVM executes each of the two instance initializers and then returns the newly created instance to the main method. Of course, the main method does not assign the reference of the newly created instance to any variable but that is okay.

An instance initializer must appear directly within the scope of a class. The block of code in the main method of the following code, therefore, is not an instance initializer. It is just regular method body code. It is valid though.

```
public class TestClass{

{
    System.out.print("Hello ");
}

public static void main(String[] args){
    //this following is not an instance initializer

{
    System.out.print("World!");
}

new TestClass();
}

}
```

Can you tell what it will print? That's right. It will print "World!Hello "

(without the quotes, of course). While executing the main method, the JVM encounters the print statement that prints "**World!**" so that is printed. Next, it encounters an instruction to create an instance of TestClass. Note that this is the first instance of TestClass that it is going to create. While finishing up the creation of this instance, it executes the instance initializer that prints "**Hello**". Thus, the net output on the console is "**World!Hello**".

Accessing members from instance initializers [↳](#)

Instance initializers have access to all of the members of a class. This includes static as well as instance fields and methods. Just like the instance methods, instance initializers have access to the implicit variable "**this**" and "**super**" (I will cover "**super**" later).

Forward referencing [↳](#)

While accessing instance fields, the order of their appearance in relation to the initializer block is important. Although an instance variable declared anywhere in the class is visible to all instance initializers, an instance initializer is not allowed to use the value of a variable if that variable is declared below the initializer. It can assign a value to such a variable though. For example, consider the following code:

```
public class TestClass{  
    {  
        System.out.print(i); // 1  
        - invalid forward reference  
        i = 20; //2 - v  
        valid forward reference  
    }
```

```
int i = 10;

public static void main(String[] args){
    new TestClass();
}

}
```

The line marked //1 in the above code will not compile because it is attempting to use the value of the variable `i` , which is declared below the initializer, but the line marked //2 will compile fine because it is assigning a value to `i` .

This rule is applicable only for the variables and not for methods. Thus, the following code will compile fine:

```
public class TestClass{

{
    printI(); //valid forward reference

}

void printI(){
    System.out.print(i);
}

int i = 10;

public static void main(String[] args){
    new TestClass();
}

}
```

Although the code will compile and run fine, you may be surprised by what it prints. JVM executes instance initializer blocks and instance variable initializer

statements in the order they appear. Therefore, in the above code, the JVM executes the instance initializer block before it executes the initialization of the variable in the statement `int i = 10;`

(Note that the JVM has already allocated space for the variable `i` and has already provided a default value of 0 to `i` as a part of performing steps 1 and 2 that I described at the beginning of this topic.)

While executing the instance initializer, the JVM invokes the `printI` method. The `printI` method, in turn, prints the value of `i`, which is `0` at this time.

10.4.2 Creating constructors [↳](#)

Constructor of a class [↳](#)

A constructor of a class looks very much like a method of a class but there are two things that make a method a constructor:

1. **Name** - The name of a constructor is always exactly the same as the name of the class.
2. **Return type** - A constructor does not have a return type. It cannot even say that it returns void.

The following is an example of a class with a constructor.

```
class TestClass{  
  
    int someValue;  
    String someStr;  
  
    TestClass(int x)  
  
        //No return type specified  
  
    {  
        this.someValue = x; //initializing someValue
```

```
//return; //legal but not required.  
}  
}
```

Observe that the name of the constructor is the same as that of the class and that there is no return type in the declaration of the constructor. Also observe that there is no return statement because a constructor cannot return anything, not even void. However, it is permissible to write an empty return statement as shown in the code above.

It is interesting to note that a class can have a method with the same name as that of the class. For example, consider the following class:

```
class TestClass{  
  
    int someValue;  
    String someStr;  
  
    void  
  
    TestClass(int x) //<-- observe the return type void  
  
    {  
        this.someValue = x;  
    }  
}
```

In the above code, **void TestClass(int x)** is not a constructor but it is a valid method nevertheless and so the code will compile fine.

Besides the above two rules there are several other rules associated with constructors. Some of them are related to inheritance and exception handling and I will cover them later. Here, I will talk about rules that are applicable to constructors in general.

The default constructor [🔗](#)

This is one of the most confusing rules about constructors and it makes for a very good trap in the exam. If I tell you that every class must have a constructor, you might have trouble believing it because so far, I have shown you so many classes that had no constructor at all! Well, here is the deal - it is true that every class must have at least one constructor but the thing is that the programmer doesn't necessarily have to provide one. If the programmer doesn't provide any constructor, then the compiler will add a constructor to the class on its own. This constructor, that is, the one provided by the compiler, is called the "**default**" constructor. This default constructor takes no argument and has no code in its body. In other words, it does absolutely nothing. For example, if I write and compile this class, `class Account { }`, the compiler will automatically add a constructor to this class which looks like this:

```
class Account{
    Account(){ } //default constructor added by the compiler
}
```

This sounds simple but the cause of confusion is the fact that if you write any constructor in a class yourself, the compiler will not provide the default constructor at all. So, for example, consider the following class:

```
class Account{
    int id;
    Account(int id){
        this.id = id;
    }
    public static void main(String[] args) {
        Account a = new Account();
    }
}
```

I have provided a constructor explicitly in the above class. Can you can guess

what will happen if I try to compile this class? It will not compile. The compiler will complain that **Account** class does not have a constructor that takes no arguments. What happened to the default constructor, you ask? Well, since this class provides a constructor explicitly, the compiler feels no need to add one on its own.

The discussion on default constructors also gives me an opportunity to talk about one misconception that I have often heard, which is that constructors must initialize all instance members of the class. This is not true. A constructor is provided by you, the programmer, and you can decide which instance members you want to initialize. For example, the constructor I showed earlier for **TestClass** doesn't touch the **someStr** variable. The default constructor provided by the compiler also doesn't assign any values to the instance members. Remember that the JVM always provides default values to static and instance members anyway.

You will most certainly get a question that tests your knowledge about the default constructor. Watch out for code that assumes the existence of the default constructor while the class provides a constructor explicitly.

Difference between default and user defined constructors [↳](#)

There are only two things that you need to remember about a default constructor:

1. **When is it provided** - It is provided by the compiler **only** when a class does not define **any** constructor explicitly.
2. **What does it look like** - The default constructor is the simplest constructor that you will ever see. It does not take any argument, does not have any throws clause, and does not contain any code.

But one peculiar thing about the default constructor is its accessibility. The accessibility of the default constructor is the same as that of the class. That means, if the class is public, the default constructor will also be public and

if the class has default accessibility, the default constructor will also have default accessibility. If you have no idea about accessibility, don't worry, I will talk about it in the next topic.

The default constructor is also sometimes called the "**default no-args**" constructor because it does not take any arguments. But this is technically imprecise because the default constructor is always a no-args constructor. There is nothing like a default constructor with args!

You can, on the other hand, write a no-args constructor explicitly in a class with a throws clause and with a different accessibility. For example, it is quite common to have a class with private no-args constructor if you don't want anyone to create instances of that class. I will talk more about it in the next topic.

Benefit of constructors

Anything that you can do in a method, you can do in a constructor and vice-versa. There is no limitation on what a method or a constructor can do. Why not just have a method and name it as init() or something? Indeed, you will encounter Java frameworks such as servlets that do exactly that. Then why do you need a constructor? Well, a detailed discussion of the pros and cons of constructors is beyond the scope of this book, but I will give you a couple of pointers.

As I mentioned earlier, an instance is not considered "ready to use" until its constructor finishes execution. Thus, a constructor helps you make sure that an instance of your class will be initialized as per your needs before anyone can use it. Here, "use" essentially means other code accessing that instance through its fields or methods.

It is possible to do the same initialization in a method instead of a constructor but in that case the users of your class would have to remember to invoke that method explicitly after creating an instance. What if a user of your class fails to invoke that special method after creating an instance of your class? The instance may be in a logically inconsistent state and may produce incorrect results when the user invokes other methods later on that instance. Therefore, a constructor is the right place to perform all initialization activities of an object. It is a place where you make sure that the instance is ready with all that it needs to perform

the activities it is supposed to perform in other methods.

Constructors also provide thread safety because no thread can access the object until the constructor is finished. This protection is guaranteed by the JVM to the constructors and is not always available to methods.

10.4.3 Overloading constructors

A class can have any number of constructors as long as they have different signatures. Since the name of a constructor is always the same as that of the class, the only way you can have multiple constructors is if their parameter type list is different.

Just like methods with same names, if a class has more than one constructors, then this is called "**constructor overloading**" because the constructors are different only in their list of parameter types.

There are a couple of differences between method overloading and constructor overloading though. Recall that in case of method overloading, a method can call another method with the same name just like it calls any other method, i.e., by using the method name and passing the arguments in parenthesis. In case of constructor overloading, when a constructor calls another constructor of the same class, it is called "**constructor chaining**" and it works a bit differently. Let me show you how.

Constructor chaining

A constructor can invoke another constructor using the keyword **this** and the arguments in parenthesis. Here is an example:

```
class Account{  
    int id;
```

```
String name;
Account(String name){
    this(111, name); //invoking another constructor
here
```

```
        System.out.println("returned from two args cons
tructor");
    }
Account(int id, String name){
    this.id = id;
    this.name = name;
}
public static void main(String[] args) {
    Account a = new Account("amy");
}
}
```

Observe that instead of using the name of the constructor, the code uses 'this' . That is, instead of calling `Account(111, name);` , the code calls `this(111, name);` to invoke the other constructor. It is in fact against the rules to try to invoke another constructor using the constructor name and it will result in a compilation error.

Invoking another constructor from a constructor is a common technique that is used to initialize an instance with different number of arguments. As shown in the above example, the constructor with one argument calls the constructor with two arguments. It passes one user supplied value and one default value to the second constructor. This helps in keeping all initialization logic in one constructor, while allowing the user of the class to create instances with different parameters.

The only restriction on the call to the other constructor is that it must be the first line of code in a constructor. This implies that a constructor can invoke another constructor only once at the most. Thus, the following three code snippets for constructors of Account class will not compile:

```
Account(String name){
```

```

        System.out.println("calling two args constructor")
;
    this(111, name); //call to another constructor must
    be the first line

}

Account(){
    this(111);
    this(111, "amy"); //call to another constructor must
    be the first line

}

Account(String name){
    Account(111, name); //incorrect way to call to another
    constructor but if the Account class had a method named
    Account, this would be a valid call to that method.

}

}

```

Invoking a constructor

It is not possible to invoke the constructor of a class directly. It is invoked only as a result of creation of a new instance using the "new" keyword. The only exception to this rule is when a constructor invokes another constructor through the use of "this" (and "super", which I will talk about in another chapter) as explained above. In other words, if you are given a reference to an object, you cannot use that reference to invoke the constructor on that object.

Now, consider the following program. Can you tell which line out of the four

lines marked LINE A, LINE B, LINE C, and LINE D should be uncommented so that it will print **111, dummy** ?

```
class Account{  
    int id;  
    String name;  
    public Account(){  
        id = 111;  
        name = "dummy";  
    }  
  
    public void reset(){  
        //this(); //<-- LINE A  
  
        //Account(); //<-- LINE B  
  
        //this = new Account(); //<-- LINE C  
  
        //new Account(); //<-- LINE D  
  
    }  
  
    public static void main(String[] args) {  
        Account a = new Account();  
        a.id = 2;  
        a.name = "amy";  
        a.reset();  
        System.out.println(a.id+, "+a.name);  
    }  
}
```

The answer is none of these. First three are invalid attempts to invoke the constructor - **this()** can only be used within a constructor, **Account()** is

interpreted as method call to a method named Account, but such a method doesn't exist in the given code, and `this = new Account()` attempts to change "`this`", which is a final variable, to point to another object. Thus, none of these three statements will compile.

`new Account()` is a valid statement but it creates an entirely new Account object. The instance variables of this new Account object are indeed set to `111` and `"dummy"`, but this doesn't change the values of the current Account object.

10.4.4 Instance initializers vs constructors [↳](#)

Instance initializers and constructors are meant for the same purpose, i.e., to give the programmer an opportunity to initialize the instance. But there are a few fundamental differences between them:

1. While creating an instance of a class, the JVM executes all of the instance initializers of that class one by one but it executes only one of its constructors (depending on the arguments passed). Of course, as you saw earlier, it is possible for a constructor to invoke another constructor using "`this`". No such explicit chaining of instance initializers is possible.
2. The restriction on the usage of variables declared below an instance initializer is not applicable to constructors. As explained earlier, an instance initializer can forward reference a variable but only as long as it is assigning a value to that variable and not trying to make use of the value of that variable.
3. The third difference is related to exceptions, a topic that I have not yet discussed, so, don't worry if you don't get this completely. The compiler expects an instance initializer to execute without throwing an exception. If the compiler can figure out that an instance initializer will always end up with an exception, it will refuse to compile the code. Here is an example:

```
class TestClass{  
    {  
        throw new RuntimeException(); //this line wil
```

1 not compile

```
}

int i = 10;

public static void main(String[] args){
    new TestClass();
}

TestClass(){
    throw new RuntimeException(); //this line is fine
}

}
```

The above code will fail to compile with an error message saying, "error: initializer must be able to complete normally". The compiler notices that there is no way this initializer will ever be able finish successfully and therefore, there is no way an instance of this class can ever be created.

This restriction is not applicable to constructors. That is why the constructor in the above code is fine.

Other than the above three differences, instance initializers and constructors work the same way. So, when should you use instance initializers? Ideally, never. Instance initializers make the code difficult to follow and are best avoided. A well designed class should not need to use instance initializers. However, if you have a class with multiple constructors and if you want to add some common initialization logic to each of its constructors, you can put that logic in an instance initializer. This will save you from repeating the same code in the constructors. In that sense, instance initializers are a powerful hack :)

In the exam you may see a question on the basics of instance initializers but we haven't seen anyone getting a question that requires the details discussed in this section. I have talked about

them only because it is important for interviews and practical programming.

10.4.5 final variables revisited

Now that you know about initializers and constructors, let me go back to final variables for a moment.

Since the value of a final variable cannot be changed, Java doesn't assign it a default value and instead, forces the programmer to assign it a value explicitly before it can be used by other classes. This idea of explicit initialization drives the following rules about their initialization:

1. **Explicit initialization of a static final variable** - A static variable can be used by other classes only after the class is loaded and made ready to use. This means that the variable must be initialized by the time class loading is complete. Thus, you can either assign a value to a final static variable at the time of declaration or in any one of the **static initializers** (aka **class initializers**) as shown below:

```
class TestClass{  
    static final boolean DEBUG = true; //initializing at the time of declaration  
  
    static final int value; //not initializing value here  
  
    static  
    {  
        value = 10;  
    }  
}
```

```

    }

    public static void main(String[] args){
        System.out.println(value);
    }
}

```

Since a class can have multiple static initializers and all of them are executed when a class is loaded, you can initialize a static final variable in any one of them but not in more than one of them because the first static initializer would have already set its value and the second one would only be changing the existing value, which is not permitted. Thus, the following code will not compile:

```

class TestClass{
    final static int value;

    static {
        value = 10;
    }

    static {
        value = 20; //won't compile because value is already initialized
    }
}

```

2. **Explicit initialization of instance final variables :** Since an instance variable can be used by other classes only after an object of this class is created, it follows that a instance final variable can be initialized at the time of declaration, in any of the instance initializers, or in all of the constructors. For example:

```
class TestClass{
    final int value; //not initializing here

    {
        value = 10; //initializing in an instance
        initializer
    }

}
```

Again, as with static initializers, a class can have multiple instance initializers and since all of them are executed while creating an instance, an instance final variable can be initialized in only one of the instance initializers. If the variable is not initialized in any of the instance initializers, it must be initialized in each of the constructors. For example:

```
class TestClass{
    final int value; //not initializing here

    {
        //not initializing value here either
    }

    TestClass(){
        value = 10;
    }

    TestClass(int x){
        value = x;
    }
}
```

```
    TestClass(int a, int b){ //this constructor wi  
ll not compile because it leaves value uninitialized  
}  
}
```

Observe that initialization in constructors works differently from initialization in instance initializers. The difference stems from the fact that all instance initializers are executed but only one constructor is executed while creation of an instance of that class.

3. **Explicit initialization of local variables (aka method or automatic variables)** - A method variable can only be used within that method and therefore, it may be initialized at anytime before it is accessed. It may even be left uninitialized if it is not used anywhere in the method. For example, the following code compiles and runs fine:

```
class TestClass {  
    public static void main(String[] args){  
        final int x; //not initializing x anywhere  
  
        final int y;  
        System.out.println("hello");  
        y = 10; //initializing y just before it is accessed  
  
        System.out.println(y);  
    }  
}
```

10.5 Apply the static keyword to methods and fields

10.5.1 Apply the static keyword to methods and fields [🔗](#)

I have already discussed the meaning of the word static and what it implies when applied to a method or a field in the "Kickstarter for Beginners" chapters.

In this section, I will dig a little deeper and explain the nuances of static from the perspective of the exam. Let us start with the syntax.

You can declare any member of a class (i.e. a method, a field, or any nested type definition) as static using the keyword static. In the case of a method or a field, this keyword must appear before the return type of the method or the type of the field respectively. For example:

```
class Foo{
    static int bar;
    static String biz;
    private static void baz(){ } //order of modifiers do
esn't matter

    static public final void boz(){ } //order of modifie
rs doesn't matter

}
```

In case of nested type definitions, the static keyword must appear immediately before the type. For example:

```
class Boo{
    static class NestedClass{
    }

    static interface NestedInterface{
    }
```

```
    static enum NestedEnum{  
    }  
}
```

A top level class, interface, or an enum cannot be declared static. It wouldn't make sense either because static is a property of the relationship between the owner and the owned types. For example, class **Boo** owns **NestedClass** statically. Class **Boo** cannot be static because it is not owned by any other type.

Local members, i.e., members defined within a method, cannot be static either. Thus, the following code will fail to compile:

```
class TestClass {  
    public static void main(String[] args){  
        static int x; //can't be static  
  
        static class Y{ //can't be static  
  
    }  
    }  
}
```

Does that mean the variable x and the class Y are an instance variable and an instance class respectively? No, such members are not members of the class at all so, the concept of static-instance does not apply to them. Such members are local to the method and cease to exist as soon as the method execution is complete. They cannot be referenced from outside the method.

10.5.2 Accessing static members

A static member exists as a member of the owning class and not as a member of an instance of the owning class. In other words, a static member does not require

an instance of the owning class to exist. A static member, therefore, can be accessed by specifying the name of the owning class. For example, the static variables of **Foo** and **Boo** can be accessed from another class like this:

```
class TestClass{
    public static void main(String[] args){
        System.out.println(Foo.bar); //prints 0 (why?)

        System.out.println(Foo.biz); //prints null (why?)

        Boo.NestedClass nc = new Boo.NestedClass();
    }
}
```

Accessing a static member using the name of the owning class is the standard and recommended way of accessing static members. However, Java allows a static member to be accessed through a variable as well. For example, the static variable **bar** of class **Foo** can also be accessed like this:

```
class TestClass{
    public static void main(String[] args){
        Foo f = null;
        System.out.println(f.bar); //prints Foo's bar

    }
}
```

Observe that **f** is **null**. But the compiler doesn't care about **f** being **null** because it notices that **bar** is a static variable and an instance of **Foo** is not needed to access **bar**. The compiler knows what **bar** in the statement **f.bar** implies and effectively translates it to **Foo.bar**.

This simple example highlights an important aspect of accessing static members: access to static members is decided by the compiler at compile time by checking

the declared type of the variable. It is not decided by the JVM at run time. The compiler knows that the type of the variable `f` is `Foo` and that `bar` is a static member of `Foo`, that is why the compiler binds the call to `f.bar` to `Foo`'s `bar`. The compiler doesn't care what `f` may point to at run time. Now, armed with this knowledge, can you tell what the following code will do?

```
class TestClass{
    public static void main(String[] args){
        Foo f = null;
        f.boz();
    }
}
```

That's right. It will compile and run fine (of course, without any output). But the interesting thing is that there will be no `NullPointerException`. Since the type of the reference variable `f` is `Foo` and `boz()` is a static method of `Foo`, the compiler binds the call to `f.boz()` to `Foo`'s `boz()`.

This is called **static binding** or **compile time binding** because the compiler doesn't leave the decision of binding a call to the JVM. This concept will play an important role when we discuss inheritance and polymorphism later.

10.5.3 Accessing static members from the same class [↳](#)

A static member of a class can also be accessed directly from static as well as instance members of the same class without the need to use the class name. For example, the following class makes use of a static variable to count the number of instances of that class:

```
class InstanceCounter{
    static int count;
    InstanceCounter(){
        //directly accessing count from a constructor
        count++;
    }
}
```

```

    }

    static void printCount(){
        //directly accessing count from a static method

        System.out.println(count);
    }

    void reduceCount(){
        //directly accessing count from an instance method

        count--;
    }
}

```

The following class shows various ways in which you can access static members of a class from another class:

```

class TestClass {
    public static void main(String[] args){
        InstanceCounter ic = new InstanceCounter();
        ic.printCount(); //accessing static method through
        a reference to an instance

        new InstanceCounter().printCount(); //accessing static
        method through an implicit reference to an instance

        System.out.println(InstanceCounter.printCount()+""
        "+InstanceCounter.count); //accessing static members
        using the class name
    }
}

```

```
 }  
 }
```

You should be able to tell what the above code will print.

10.5.4 Importing static fields [↳](#)

In the "Java Basics" chapter, I discussed **import** statements. I showed you how to import classes as well as static members of a class. If you want to access static members of a class several times, the **import static** statement provides an easy way to save some keystrokes.

It is interesting to know that the **import static** statement was added in Java 7 only to avoid a kludgy work around for accessing global constants. Before Java 7, if a developer needed to define global constants, they would define the constants in an interface like this:

```
public interface Constants{  
    public static final double INTEREST_RATE = 0.15;  
    public static final boolean COMPOUNDING = true;  
    public static final int PERIOD = 1;  
}
```

To use these constants in any class, they would have that class implement the above interface like this:

```
public class InterestCalculator implements Constants  
{  
    public static double compute(double principle, double time){  
        if(COMPOUNDING){
```

```

    //compute and return compound interest

}else{
    //compute and return simple interest

}
}
}

```

The benefit of having the class implement the **Constants** interface is that it saves the developer from typing the class name of the class in which the constant is defined, i.e., instead of typing **Constants.COMPOUNDING**, they can just type **COMPOUNDING**. But many experts believe that this makes the code difficult to understand. It is also an abuse of "implements" keyword because there is no functionality that is being implemented here. The usage of interface is also objectionable because we are not defining any behavior in the interface either.

The **import static** statement avoids the need for this technique by making it simple to import static members of a class. You can now define the constants in a class and import those constants statically like this:

```

package accounting; //need to put this class in a package because recall that it is not possible to import anything from the unnamed package

```

```

public class Constants{
    public static final double INTEREST_RATE = 0.15;
    public static final boolean COMPOUNDING = true;
    public static final int PERIOD = 1;
}

import static accounting.Constants.*;

```

```

public class InterestCalculator {
    public static double compute(double principle, double time){
        if(COMPOUNDING){
            //compute and return compound interest
        }else{
            //compute and return simple interest
        }
    }
}

```

Technically, you can import static members of an interface also but from a design perspective, it is better to use a class for defining constants and reserve the use of interfaces for defining behavior in terms of methods.

10.5.5 Accessing instance members from a static method [↳](#)

Since a static method belongs to a class and not to an object of that class, a static method does not execute within the context of any instance of that class. On the other hand, an instance method is always invoked on a specific instance of a class and so, it executes within the context of the instance upon which it is invoked. An instance method can access this instance using the **implicit** variable "**this**". Since there is no instance associated with a static method, the variable **this** is not available in a static method.

The reason why the "**this**" variable is called an implicit variable becomes important here. This variable is not declared explicitly anywhere and the compiler assumes its existence in an instance method. Whenever the compiler sees an instance member being accessed from within a method directly, the compiler uses the **this** variable to access that member even when you don't type it explicitly in your code. However, when a static method tries to access an instance member, **this** is not available and so, the compiler complains that a

non-static variable cannot be referenced from a static context.

The following code illustrates this point:

```
class Book{  
    int name;  
  
    static void printName1(){  
        System.out.println(this.name); //will not compile  
  
        System.out.println(name); //same as above. will  
        //not compile  
    }  
  
    void printName2(){  
        System.out.println(this.name); //this is fine.  
  
        System.out.println(name); //same as above. this  
        //is fine.  
    }  
}
```

In the above code, the compiler realizes that `name` is an instance variable and so, tries to access it through `this`, i.e., `this.name`. But since `this` is not available in `printName1`, it generates an error. There is no issue with `printName2` because `printName2` is an instance method and `this` is available in an instance method.

Remember that you don't always need to explicitly use the variable `this` to access instance members. You need to use it only if there is also a local variable

with the same name declared in the method and you want to refer to the instance variable instead of the local one. Technically, `this` is used to "unshadow" an instance variable if it is shadowed by a local variable of the same name.

A common misunderstanding amongst beginners is that a static method cannot access instance fields of a class. This misunderstanding exists because they see or hear this statement in many places. However, it is an incomplete statement. The correct statement is that a static method cannot access an instance member without specifying the instance whose member it wants to access. It will be clear when you see the following code:

```
class Book{  
    int name;  
  
    static void printName(){  
        Book b1 = new Book();  
        Book b2 = new Book();  
        System.out.println(b1.name); //this will compil  
e fine  
  
    }  
}
```

In the above code, we are accessing the instance variable `name` from within a static method. Notice that there are two instances of `Book`. Each instance has its own copy of the variable `name`. But because we are using the reference `b1` to access `name`, the compiler knows exactly whose `name` we intend to access. It knows that we want to access the `name` field of the `Book` instance pointed to by the reference variable `b1`. That is why there is no problem. This shows that an instance field can indeed be accessed from a static method as long as we specify exactly the instance whose field we want to access.

10.5.6 Class loading and static initializers [🔗](#)

Recall that executing a Java program or a Java class essentially means executing the "main" method of that class. But before the JVM can pass the control to this main method, it has to first find the class file, and "load" the class into its memory. The loading of a class is done by a class loader. Don't worry, you don't need to worry about class loaders and the whole class loading process for the exam. But you do need to know about one thing that happens to the class after the class is loaded and before the main method is invoked.

While executing code, whenever the JVM encounters the usage of a class for the **first time**, it allocates and initializes space for the static fields of that class. Since static fields belong to the class and not to the objects of the class, there is only one copy of such fields. The JVM initializes these fields to their default values (i.e. zero for numeric and char types and null for reference types) and executes the "static initializers" defined in the class. Static initializers are nothing but blocks of code marked with the keyword static as shown below:

```
class TestClass{
    //the following is a static initializer

    static {
        System.out.println("In static block");
    }

    public static void main(String[] args){
        System.out.println("In main");
    }
}
```

The above program generates the following output:

```
In static block
In main
```

As you can observe from the above output, the main method is executed after the static block. A static block provides an opportunity for the developer to initialize

the static fields of the class and execute any other business logic that the developer wishes to execute before the class is put to use by any other code. The following class shows how a developer might make use of a static initializer:

```
class InterestCalculator{  
    static double RATE;  
    static {  
        System.out.println("In static block. RATE = "+RATE);  
        RATE = 10;  
    }  
  
    public static double computeInterest(double principle, int yrs){  
        return RATE*principle*yrs/100;  
    }  
  
    public static void main(String[] args){  
        double interest = computeInterest(100, 1);  
        System.out.println(interest);  
        interest = computeInterest(100, 2);  
        System.out.println(interest);  
  
    }  
}
```

The `InterestCalculator` class shown above computes simple interest based on the value of `RATE`, which is set in the static initializer. It produces the following output:

In static block. RATE = 0.0

10 . 0

20 . 0

The important thing to observe in the above output is that the value of **RATE** is printed as **0 . 0** . This is because the JVM has already initialized the **RATE** variable to its default value of 0.0. It is that value that is being printed. **RATE** is being set to **10** after this print statement. The second thing that you should notice is that **RATE** is printed only once. This is because a static block is executed only once when a class is loaded and initialized. The JVM never executes the static block again no matter how many times the class is used afterwards.

There are only a few simple rules about static blocks:

1. A class can have any number of static blocks. They are executed in the order that they appear in the class.
2. A static block can access all static variables and static methods of the class. However, if the declaration of a static variable appears after the static block, then you can only set the value of that variable in the static block. Here is an example:

```
class TestClass{  
    static int a;  
  
    static{  
        System.out.println(a); //valid, a is declared  
        before the static block  
  
        System.out.println(b); //INVALID, cannot read  
        b's value because b is declared after the static b  
        lock  
  
        b = 10; //valid because b is being assigned a  
        value.
```

```

        m(); //valid even though m is defined later

    }

    static void m(){
        System.out.println(b); //valid, a method can do anything with a variable that is declared later in the code
    }

    static int b;

    public static void main(String[] args){
    }

    //another static block

    static{
        System.out.println(b);
    }
}

```

3. If the class has a superclass, and if the superclass hasn't been initialized already, the JVM will initialize the superclass first and then proceed with the initialization of this class. The following example explains this process:

```

class Parent{
    static{
        System.out.println("Initializing Parent");
    }
}
class ChildA extends Parent{

```

```

        static{
            System.out.println("Initializing ChildA");
        }
    }
class ChildB extends Parent{
    static{
        System.out.println("Initializing ChildB");
    }
}

public class TestClass{
    public static void main(String[] args) {
        ChildA a = new ChildA(); //will cause class
Parent and then class ChildA to be initialized

        ChildB b = new ChildB(); //will cause only
ChildB to be initialized
    }
}

```

The above code generates the following output:

```

Initializing Parent
Initializing ChildA
Initializing ChildB

```

When the JVM encounters the statement `ChildA a = new ChildA()`, it realizes that `ChildA` is being used for the first time here. So, before creating an object of the `ChildA` class, it needs to initialize the `ChildA` class. Since `ChildA` extends `Parent`, the JVM must first initialize the `Parent` class. This is why `Initializing Parent` is printed before `Initializing ChildA`. Once `ChildA` is loaded and initialized, the JVM creates an object of `ChildA` and assigns its reference to variable `a`. Similarly, when the JVM encounters the statement `ChildB b = new ChildB()`, it tries to initialize the `ChildB` class. While

initializing `ChildB`, the JVM realizes that its super class, i.e., `Parent`, has already been initialized and so, it does not execute `Parent`'s initialization logic again. Thus, only `Initializing ChildB` is printed this time. You might now wonder why we stopped at the `Parent` class. What happened to the `Object` class? After all, `Object` is the super class of `Parent`! Well, `Object` is indeed the super class of `Parent`. But it is also the super class of `TestClass`. The JVM initialized the `Object` class while it was initializing the `TestClass` class.

4. There is no way to access a static block or refer to a static block. That means, you can't "invoke" or "call" a static block explicitly from anywhere. It can only be invoked by the JVM and that too, only once.

Technically, there is one copy of static fields per class per class loader, but a discussion on class loaders is way out of scope of this exam. However, be aware that technical interviewers love to touch upon class loaders and are impressed if the candidate knows about class loading and its impact on static fields. I suggest you go through a few articles on this topic such as <https://blogs.oracle.com/sundararajan/understanding-java-class-loading> if you are interviewing for a job.

10.6 Exercise

1. Create a method named `add` that can accept any number of `int` values and returns the sum of those values.
2. Create another method named `add` in the same class that can accept any number of `int` values but returns a `String` containing concatenation of all those values. What can you do to resolve compilation error due to the presence of the two methods with same signature? Invoke these methods from the main method of this class.
3. Create a class named `Student` with a few fields such as `studentId`, `name`, and `address`. Should these fields be static or non-static? Add the main

method to this class and access the fields from the main method.

4. Add a static field to **Student** class. Access this field from another class. Use appropriate import statement to access the field directly. Change accessibility of the field and see its impact on the code that tries to access it.
5. Create a method named method1 in **TestClass** that accepts a Student object and updates the static as well as instance fields of this object. Pass the same Student object to another method named method2 and print the values. Assign a new Student object to the Student variable of method2 and set its fields to different values. After returning back to method1, print the values again. Explain the output.
6. Add a constructor in Student class that accepts values for all of its instance fields. Add a no-args constructor in Student class that makes use of the first constructor to set all its instance fields to dummy values.
7. Create a class named **Course** in different package. Add a static method named **enroll** in this class that accepts a Student. Use different access modifiers for fields of Student class and try to access them from the enroll method.

Chapter 11 Encapsulation

- Apply access modifiers
- Apply encapsulation principles to a class

11.1 Apply access modifiers

11.1.1 Accessibility

One of the objectives of object-oriented development is to encourage the users of a component (which could be a class, interface, or an enum) to rely only on the agreed upon contract between the user and the developer of the component and not on any other information that the component is not willing to share.

For example, if a component provides a method to compute taxes on the items in a shopping cart, then the user of that component should only pass the required arguments and get the result. It should not try to access internal variables or logic of that class because using such information will tie the user of that component too tightly to that component. Imagine a developer writes the following code for the TaxCalculator class:

```
class TaxCalculator{  
    double rate = 0.1;  
    double getTaxAmount(double price){  
        return rate*price;  
    }  
}
```

Ideally, users of the above class should use the `getTaxAmount` method but let us say they do not and access the `rate` variable instead, like this:

```
//code in some other class

double price = 95.0;
TaxCalculator tc = new TaxCalculator();
double taxAmt = price*tc.rate;
```

Later on, the developer realizes that "tax rate" cannot be hardcoded to 0.1. It needs to be retrieved from the database and so, the developer makes the following changes to the class:

```
class TaxCalculator{
    //double rate = 0.1; //no more hardcoding

    double getTaxAmount(double price){
        return getRateFromDB()*price;
    }

    double getRateFromDB(){
        //fetch rate from db using jdbc
    }
}
```

Since there is no **rate** variable present in this class anymore, all other code that is accessing this variable will now fail to compile. Had they stuck to using the **getTaxAmount** method, they would not have had any problem at all. To prevent compilation failure, the developer will now be forced to maintain the **rate** variable in their new code even though this variable is not required by the class anymore. What if the tax rate changes in the database but is not updated in the **TaxCalculator** class's **rate** variable? What if one rogue user updates the rate variable at an inopportune time while another user is using it to compute taxes? Well, in both the cases the users will be computing taxes incorrectly. There will be no error message to make the developer aware of the problem.

either. This is a serious matter.

But the problem is not just with the users relying on internal details of a class. The **TaxCalculator** class doesn't give any clue as to what features does it support. How are the users supposed to know that they should be using only the **getTaxAmount** method and not the **rate** variable? In other words, the **TaxCalculator** class doesn't make its public contract clear and therefore, it would be unfair to blame only the users of this class. This is where Java's **access modifiers** of Java come into picture.

11.1.2 Access modifiers [↳](#)

Java allows a class and members of a class to explicitly specify who can access the class and the members using three accessibility modifiers: **private**, **protected**, and **public**. Besides these three, the absence of any access modifier is also considered an access modifier. This is known as "**default**" access. These access modifiers make your intention about a member very clear not only to the users of your class but also to the compiler. The compiler then helps you enforce your intention by refusing to compile code that violates your intention. Here is how these modifiers affect accessibility:

private - A private member is only accessible from within that class. It cannot be accessed by code in any other class.

For example, the problem that I showed you with the **TaxCalculator** class could have been easily avoided if the developer had simply declared the **rate** variable as **private**. That would make the variable inaccessible from any other class. The compiler would prevent the users of this class from using the **rate** variable by refusing to compile the code that tried to use it. Since there would have been no dependency on **rate** variable, the developer could have easily removed this variable without any impact on anyone.

"default" - A member that has no access modifier applied to it is accessible to all classes that belong to the same package. This is irrespective of whether the class trying to access a default member of another class is a sub class or a super class of the other class. If two classes belong to the same package, then they can access each other's default members. This is also called "**package private**" or

simply "**package access**" .

protected - A protected member is accessible from two places - if the accessing class belongs to the same package or if the accessing class is a subclass irrespective of the package to which the subclass belongs. The first case is simple because it is exactly the same as default access. All classes belonging to the same package can access each other's protected members. The second case is not so, simple and I will explain it separately in detail.

public - A public member is accessible from everywhere. Any code from any class can make use of a public member of another class. For example, the `getTaxAmount` method could have been made public. That would give a clear signal to the users to use this method if they want to compute the tax amount.

If you order the four access modifiers in terms of how restrictive they are, then `private` is **most restrictive** and `public` is **least restrictive** . The other two, i.e., `default` and `protected` , lie between these two. Thus, the order of access modifiers from the most restrictive to the least would be: `private > default > protected > public`.

In the exam, watch out for non-existent access modifiers such as "friend", "private protected", and "default".

11.1.3 Understanding protected access [↳](#)

I mentioned earlier that a protected member is accessible from any class that belongs to the same package and from any subclass even if the subclass belongs to another package. This sounds straightforward but the second part of that statement is not completely true. Since it involves the concept of inheritance, which I haven't discussed yet, I will explain this case here with an example. Consider the following two classes that belong to two different packages.

//In file Account.java

```

package com.mybank.accts;
public class Account{ //observe public modifier for class

    protected String acctId;

    //code that does something with acctId

}

//In file HRAccount.java

package com.mybank.hr;
import com.mybank.accts.*;
public class HRAccount extends Account{

    public static void main(String[] args){
        Account simpleAcct = new Account();

        simpleAcct.acctId = "111";//will not compile

        HRAccount hrAcct = new HRAccount();
        hrAcct.acctId = "111";//will compile fine

    }
}

```

In the above code, the compiler will not allow **acctId** to be accessed using the reference **simpleAcct** but it will allow **acctId** to be accessed through the

reference `hrAcct`. Ideally, since `acctId` is protected, and since a protected member can be accessed from a subclass irrespective of the package of the subclass, then `simpleAcct.acctId` should have been valid. Then what is the issue?

Actually, `protected` allows a subclass to access its own fields that the subclass inherits from its superclass. `HRAccount` class inherits the `acctId` field from `Account` class, so `HRAccount` class owns the `acctId` field contained in any `HRAccount` object but `HRAccount` class does not own an `Account` object's `acctId` field. That is why `HRAccount` class is allowed to access `HRAccount` object's `acctId` field but is not allowed to access `Account` object's `acctId` field.

In technical terms, a subclass from a different package is allowed to access a protected member of the superclass only if the subclass is involved in the implementation of the class of the reference that it is trying to use to access that member. In the above example, the line `simpleAcct.acctId` violates this rule because `simpleAcct` is a reference of type `Account` but class `HRAccount` is not involved in the implementation of class `Account`. It would not compile even if `simpleAcct` referred to an object of type `HRAccount` because the compiler checks the declared type of the reference and not the type of the actual object to which the reference points at run time. Let me add one more class to the mix:

//In file NewHRAccount.java

```
package com.mybank.newhr;
import com.mybank.hr.*;
public class NewHRAccount extends HRAccount{
    protected String name;
}
```

The above class is a subclass of `HRAccount` and belongs to a third package named `com.mybank.newhr`. Now, what if I modify `HRAccount` code to

access **NewHRAccount** like this:

```
//In file HRAccount.java

package com.mybank.hr;
import com.mybank.accts.*;
import com.mybank.newhr.*;
public class HRAccount extends Account{

    public static void main(String[] args){
        NewHRAccount newHRAcct = new NewHRAccount();

        newHRAcct.acctId = "111";//will this compile?

        newHRAcct.name = "John";//will this compile?

    }
}
```

In the above code, I am trying to access **acctId** field of **NewHRAccount** from **HRAccount**. Does **HRAccount** own the **acctId** field of **NewHRAccount**? Yes, it does because **NewHRAccount** inherits it through **HRAccount**. Therefore, the access to **acctId** in the above code is valid. However, access to **name** is invalid because **name** is defined in **NewHRAccount** and therefore **HRAccount** does not own this field.

It sounds confusing and it is indeed confusing. But if you think about it, it does make sense. Imagine you develop a class. This class provides some functionality in a way you deem appropriate. But now you want other people to use your class and also let them implement, enhance, or tweak that functionality in their own way. You don't want them to change the way your class works, you just want to them to be able to reuse your class and implement that functionality the way they deem fit in their own class. For example, **Account** class of **com.mybank.accts** package may manage its **acctId** field in a certain way.

`HRAccount` class from `com.mybank.hr` package inherits `acctId` and is now free to manage `acctId` in its own way but `HRAccount` should not be able to change how the `Account` class manages `acctId`. If `HRAccount` is allowed to mess with `Account`'s `acctId` field, then `Account` class's internal logic may go completely haywire. This is what **protected** intends to achieve. It gives a subclass full control over the fields that the subclass inherits from its parent without compromising the integrity of the parent itself.

11.1.4 Applying access modifiers to types [↳](#)

So far you have seen the usage of access modifiers on fields and methods of a class. You can apply them to the reference type definitions, i.e., classes, interfaces, and enums definitions as well. However, there are certain restrictions.

Applying access modifiers to class definitions [↳](#)

A top level class (i.e. a class that is not defined inside another reference type) can only have two types of access - public and default. But a nested class can use any of the four access modifier. For example,

public

```
class Outer{ //cannot be private or protected}
```

private

```
class Inner1 { } //valid
```

protected

```
class Inner2 { } //valid
```

```
}
```

Similarly, a top level interface or an enum can also only have public or default access but a nested interface or an enum can have any of the four access modifiers.

Applying access modifiers to members of an interface [↳](#)

As of Java 8, members of an interface are always public. The compiler will generate an error if you define them as private or protected. If you don't specify any access modifier for a member of an interface, compiler will automatically make it public (and not default, unlike in case of a class). For example:

```
interface Movable { //cannot be private or protected  
  
    int STEP = 10; //interface fields are always public  
    even if not declared public  
  
    void move(); //interface methods are always public  
    except when declared private (protected is not allowed  
    for methods in an interface)  
  
}
```

Java 9 allows an interface to have private methods. However, fields of an interface are still always public.

Applying access modifiers to members of an enum [↳](#)

The enum constants are always public even when no access modifier is

specified. On the other hand, enum constructors are always private. Compiler will generate an error if you try to make them public or protected. For example:

```
enum Day{ //can only be public or default  
    WEEKDAY, HOLIDAY; //enum constants are always public  
    Day(){ //enum constructor is always private  
        }  
        private int value; //enum field can be public, private or protected  
        public void count(){ //enum method can be public, private or protected  
            }  
    }
```

We haven't seen candidates getting questions on details of access modifiers as applied to enums. I have mentioned it only for the sake of completeness. You do, however, need to know the application of access modifiers on classes, interfaces, and their members.

11.2 Apply encapsulation principles to a class

11.2.1 Encapsulation

Encapsulation is considered as one of the three pillars of Object-Oriented Programming. The other two being **Inheritance** and **Polymorphism**. In the programming world, the word **encapsulation** may either refer to a language mechanism for restricting direct access to an object's data fields **or** it may refer to the features of a programming language that facilitate the bundling of data with the methods operating on that data. For the purpose of the exam, we will be focusing on the first meaning, i.e., restricting direct access to an object's fields.

As I explained in the previous section on **access modifiers**, letting other classes directly access instance variables of a class causes **tight coupling** between classes and that reduces the maintainability of the code. While designing a class, your goal should be to present the functionality of this class through methods and not through variables. In other words, a user of your class should be able to make use of your class by invoking methods and not by accessing variables.

There are two advantages of this approach:

1. You give yourself the freedom to modify the implementation of the functionality without affecting the users of your class. In fact, users of a well encapsulated class do not even become aware of the internal variables that the class uses for delivering that functionality because the implementation details of the functionality are hidden from the users. In that sense, **encapsulation** and **information hiding** go hand in hand.
2. You can ensure that the value of a variable is always consistent with the business logic of the class. For example, it wouldn't make sense for the **age** variable of a **Person** class to have a negative value. If you make this variable public, anyone could mess with **Person** objects by setting their ages to a negative value. It would therefore, be better if the **Person** class has a public setter method instead, like this:

```
class Person{  
    private int age;
```

```

        public void setAge(int yrs){
            if(yrs<0) throw new IllegalArgumentException
        );
            else this.age = yrs;
        }

    //other code

}

```

Encapsulation in Java [↳](#)

Since the only way to restrict access to the variables of a class from other classes is to make them private, a well encapsulated class defines its variables as private. The more you relax the access restrictions on the variables, the less encapsulated a class gets. Thus, a class with public fields is not encapsulated at all. The visibility of the methods, on other hand, can be anything ranging from private to public, depending on the business purpose of the class.

Encapsulation of static members [↳](#)

Encapsulation is an OOP concept and generally applies to the instance members of a class and not to the static members. However, if you understand the spirit of encapsulation, you will notice that all we want to do is to prevent others from inadvertently or incorrectly accessing stuff that they should not be accessing. This is true of static variables also. Thus, it is better to have even the static variables as private. Ideally, the only variables that deserve to be public are the ones that are constants.

11.2.2 JavaBeans naming convention [↳](#)

In the previous section, I explained why a class should expose its functionality

through methods and not through variables. If you look at a well encapsulated class from a user's perspective, you will not see any variables (because they are private!). For example, consider the following class:

```
package library;
public class Book{
    private String title;
    public String getTitle(){ return title; }
    public void setTitle(String title){ this.title = title; }
}
```

If you try to use this class in your code, you will not see the variable `title`. You will only see methods that get and set the value for its `title` but no variable named `title`. The following code illustrates this point:

```
import library.Book;
class TestClass{
    public static void main(String[] args){
        Book b = new Book();
        //b.title = "Java Gems"; //won't work because title is private

        b.setTitle("Java Gems");
        System.out.println(b.getTitle());
    }
}
```

Thus, as far as you are concerned, a `Book` object doesn't seem to have any variable named `title` even though it does have a notion of having a `title`. In technical terms, this notion is called "property", as in, the `Book` class has a property named `title`.

JavaBeans

JavaBeans is a set of conventions that lets anyone make use of objects by easily

recognizing the properties that these objects have. There are only two rules that you need to remember about JavaBeans for the exam:

1. If a class has a non-private method whose name starts with a "get" followed by an upper case letter, then that class has a property by the name that follows the get. The type of this property is the same as the return type of this method. For example, the **Book** class shown above has a public method named **getTitle** and therefore, it has a property named **title** of type **String**. This method is called the "**getter**" method for the title property.
If the type of the property is **boolean** (or **Boolean**), the name of the getter method may also start with an "**is**" instead of a "get".
2. Similarly, if the same class also has a non-private method whose name matches "set" followed by the property name with first letter capitalized and takes a parameter of the same type as the associated getter method, this property is considered editable and the method is called the "**setter**" method for that property. For example, since **Book** class does have a public **setTitle** method that takes a **String** parameter, the **title** property is editable. Had it not had the **setTitle** method, the **title** property would have been "**read only**".

Observe that there is no mention of the name of the variable behind this property. Let me show you a few examples to make this clear:

```
package library;
public class Book{
    private String writer;

    public String getAuthor(){
        return writer;
    }
    public void setAuthor(String value){
        this.writer = value;
    }

    public String getJunkTitle(){
```

```

        return "junk";
    }

private double price;
public double getprice(){ return price; }

boolean isJunk(){ return true; }

boolean getJunk(){ return true; }

}

```

Let's apply the first rule to the above class:

1. The method `getAuthor` is not private and starts with a get followed by an upper case letter. Therefore, `author` is a valid property of `Book` class and `getAuthor` is the getter method for this property. Observe that the variable that the class internally uses to support this property is named `writer` but that has no relevance here. Since the method is public, the author property is publicly readable.
2. The method `getJunkTitle` is not private and starts with a get followed by an upper case letter. Therefore, `junkTitle` is a valid property of `Book` class and `getJunkTitle` is the getter method for this property. Observe that this method uses no internal variable at all. Furthermore, since this method is public, the `junkTitle` property is publicly readable.
3. The `Book` class does not have any method named `getPrice` (with uppercase p) therefore `Book` class does not have any property named `price` even though it does have a variable named `price`. The method `getprice` (with lower case p) starts with a get but is not followed by an upper-case letter, therefore, `getprice` is not a valid getter method for any property.
4. The method `isJunk` is not private and starts with an is followed by an upper-case letter and returns a `boolean`. Therefore, `junk` is a valid `boolean` property of `Book` class and `isJunk` is a valid getter method for

this property. Similarly, `getJunk` is also a valid getter method for the **junk** property. Since the method has default access, the **junk** property is visible only to the classes of the same package.

Now, let us apply the second rule:

1. Since **Book** class also has a `setAuthor` method, the **author** property is editable.
2. Since **Book** class does not have a `setJunkTitle` method, the **junkTitle** property is uneditable, i.e., read only.
3. Since **Book** class does not have a `setJunk` method, the **junk** property is uneditable, i.e., read only.

Since JavaBeans is a globally accepted convention for naming the properties, even automated tools can see and access the properties of any object. The objects themselves (and not the classes) are called "beans". A JavaBeans developer can make beans provided by different providers work together by having them access each other properties just through declarations in a visual editor without writing any code explicitly.

11.3 Exercise

1. List the access modifiers in the order of increasing restrictiveness.
2. What will be the impact if accessibility of a package protected instance member of a class is changed to public or private?
3. Evaluate the following class code with respect to encapsulation principles:

```
class Book{  
    public String isbn;  
    public String title;  
    private String author;  
    public Book(){  
    }  
}
```

```
    getAuthor(){ return author; }

    static int noOfAuthors;
}
```

4. What can be done to make the above class well encapsulated?

Chapter 12 Reusing Implementations Through Inheritance

- Create and use subclasses and superclasses
- Create and extend abstract classes
- Enable polymorphism by overriding methods
- Utilize polymorphism to cast and call methods, differentiating object type versus reference type
- Distinguish overloading, overriding, and hiding

12.1 Create and use subclasses and superclasses

12.1.1 Understanding Inheritance [🔗](#)

In general terms, when a class extends another class, we say that the class inherits from the other class. But this is a very vague view of inheritance. It is important to understand what exactly this class inherits from the other class and what exactly inheriting something means.

There are two things that a class contains - the instance fields defined in the class, i.e., **state**, the instance methods that provide behavior to the class, i.e., **implementation**. Furthermore, a class, on its own, also defines a **type**. Any of these three things can be inherited by the extending class. Thus, inheritance could be of state, it could be of implementation, and it could be of type.

When a class inherits something, it implies that it automatically gets that thing without you needing to explicitly define it in the class. This is very much like real life objects. When you say that a Poodle is a Dog, or a Beagle is a Dog, you implicitly know that a Poodle has a tail and that it barks. So, does a Beagle. You don't have to convey this information explicitly with a Poodle or a Beagle. In other words, both a Poodle and a Beagle inherit the tail and the barking behavior

from Dog. Note that Poodle and Beagle do not contain a Dog. Poodle is a Dog. Beagle is a Dog. Being something is very different from containing something. If Poodles and Beagles contained a Dog, then both of them would always bark in exactly the same way, i.e., like a Dog. But you know that they don't bark the same way. Both of them do bark but they bark differently.

Inheritance of state [↳](#)

Only a class can contain state and therefore, only a class can extend another class. Furthermore, Java restricts a class from extending more than one class and that is why it is often said that Java does not support multiple inheritance. However, technically, it would be more precise to say that Java does not support multiple inheritance of state.

Inheritance of implementation [↳](#)

Before Java 8, only classes were allowed to have implementation. Starting with version 8, Java allows even interfaces to contain implementations in the form of "default" methods. Thus, a class can inherit implementation by extending a class and/or by implementing interfaces. This allows a class to inherit implementations from more than one types. This is one form of multiple implementation inheritance. However, due to the way default methods are inherited in a class, it is still not possible for a class to inherit more than one implementation of a method in Java. I will talk more about default methods later.

Inheritance of type [↳](#)

Java allows you to define a type using an interface as well as a class (and an enum but that is not relevant to this discussion). Thus, a class can inherit behavior by extending a class and/or by implementing an interface. Since Java allows a class to implement multiple interfaces, it can be said that Java supports multiple inheritance of type.

12.1.2 Inheriting features from a class ↗

To inherit features from another class, you have to extend that class using the **extends** keyword. For example, consider the following two classes:

```
public class Person{
    public String name;
    public String getName() {
        return name;
    }

    public static

    int personCount;
    public static

    int getPersonCount(){
        return personCount;
    }
}

public class Employee extends Person{
    public String employeeId;

    public static void main(String args[]){
        Employee ee = new Employee();
        ee.employeeId = "111";
        ee.name = "Amy";
        System.out.println(ee.getName());
    }
}
```

In the above code, Person is the **parent class** (also called the **super class** or the **base class**) and Employee is the **child class** (also called **sub class** or **derived class**). Since **Employee** extends **Person**, it automatically "gets" the **name** field as well the **getName** method. Thus, it is possible to access the **name** field

and the `getName` method in an `Employee` object as if they were defined in the `Employee` class itself, just like the `employeeId` field.

Similarly, the `Employee` class also "gets" the static variable `personCount` and the static method `getPersonCount` from Person class. Thus, it is possible to access `personCount` and `getPersonCount` in an `Employee` class as if they were defined in the `Employee` class itself.

Members inherited by a class from its super class are as good as the members defined in the class itself and are therefore, passed on as inheritance to any subclass. For example, if you have a Manager class that extends an Employee class, the Manager class will inherit all members of the Employee class, which includes the members defined in Employee class as well as members inherited by Employee class from Person class.

```
public class Manager extends Employee {  
    public String projectId;  
  
    public static void main(String args[]){  
        Manager m = new Manager();  
  
        m.projectId = "OCPJP-I";  
  
        m.employeeId = "111";  
  
        m.name = "Amy";  
        System.out.println(m.getName());  
  
    }  
}
```

There is no limit to how deep a chain of inheritance can go.

Inheriting constructors and initializers [↳](#)

Constructors are not considered members of a class and are therefore, never inherited. The following code proves it:

```
class Person{
    String name;
    Person(String name){
        this.name = name;
    }
}

public class Employee extends Person{

    public static void main(String args[]){
        Employee ee = new Employee("Bob");
    }
}
```

Employee does not inherit the String constructor of Person and that is why the compiler is not happy when you try to instantiate a new Employee using a String constructor.

Similarly, static and instance initializers of a class are also not considered to be members of a class and they are not inherited by a subclass either.

The `java.lang.Object` class [↳](#)

`java.lang.Object` is the root class of all classes. This means that if a class does not explicitly extend another class, it implicitly extends the `Object` class. Thus, every class inherits the members defined in the `Object` class either directly or through its parent. There are several methods in this class but the only methods that you need to be aware of for the OCP Java 11 Part 1 exam are the `equals` and the `toString` method. It is because of the presence of these methods in the `Object` class that you can call them on any kind of object. I will talk more about them later.

Since **Object** is the root class of all classes, it is the only class in Java that does not have any parent.

Extending multiple classes [↳](#)

As I mentioned earlier, Java does not support multiple inheritance of state. Since classes contain state, it is not possible for a class to extend more than one class. Thus, something like the following will not compile:

```
public class Programmer{  
}  
//can't extend more than one class  
  
public class Consultant extends Person, Programmer{  
}
```

12.1.3 Inheritance and access modifiers [↳](#)

You might have noticed that I violated an important principle of OOP in **Person** and **Employee** classes. These classes are not well encapsulated. Recall that in the previous chapter, I explained why it is bad to have public fields. Here however, both the **Person** class, and since the **Employee** class extends **Person**, the **Employee** class have a field that is exposed to the whole world. As per the principle of encapsulation, I should make the **name** field private. However, if I make **name** private, **Employee** fails to compile with the following error message:

Error: name has private access in Person

How come, you ask? It turns out that access modifiers greatly impact inheritance. In fact, only those members that are visible to another class as per the rules of access modifiers are inherited in a subclass. Remember that a private member is not visible from anywhere except from the declaring class itself. Thus, a subclass cannot inherit a private member of the super class at all. That is why **Employee** will not inherit **name** from **Person** if you make **name** private.

With the above rule in mind, let us examine the impact of access modifiers on inheritance.

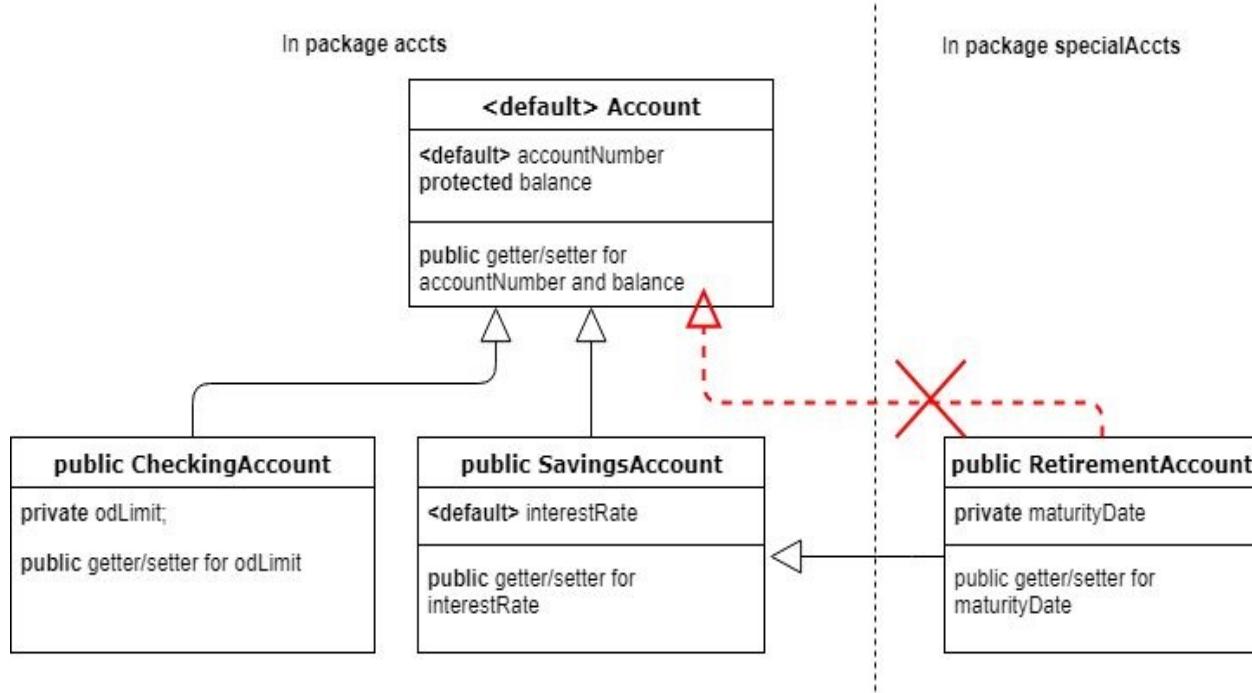
private - As explained above, private members are not inherited by a subclass.

default - Since a member with default access is visible only to a class in the same package, it can only be inherited in the subclass if the subclass belongs to the same package.

protected - This access modifier is actually built to overcome the restriction imposed by the default modifier. Recall that in addition to being visible to all classes of the same package, a protected member of a class is visible to another class that belongs to a different package if that class is a subclass of this class. Thus, even a subclass that belongs to another package inherits protected members of the super class.

public - public members are always inherited by a subclass.

For example, let's say you are designing an application that has three kinds of accounts: **CheckingAccount** , **SavingsAccount** , and **RetirementAccount** . All three classes need to have an account number and methods for checking balance. Besides these common features, each of these account types also have features that are specific only to that account type. The CheckingAccount has an **overdraftLimit** , the SavingsAccount has an **interestRate** , and the RetirementAccount has a **maturityDate** . The figure below shows the hierarchy of the classes.



Since all accounts are to have `accountNumber` and `balance`, I have put these members in a common super class called `Account`. The following is the code for these classes:

```

package accts;
class Account{
    int accountNumber;
    protected double balance;

    //public getter and setter methods for the above fields
}

package accts;
public class CheckingAccount extends Account{
    private double odLimit;
    //public getter and setter methods for odLimit
  
```

```
}

package accts;
public class SavingsAccount extends Account{
    double interestRate;
    //public getter and setter methods for interestRate

}

package specialAccts;

import accts.SavingsAccount;
public class RetirementAccount extends SavingsAccount

{
    private String maturityDate;
    //getter and setter methods for maturityDate

    public static void main(String[] args){
        Account a = new Account(); //will not compile

        RetirementAccount ra = new RetirementAccount();
        //valid

        ra.balance = 100.0;//valid

        ra.setAccountNumber(10);//valid

        ra.setInterestRate(7.0);//valid
    }
}
```

```

SavingsAccount sa = new SavingsAccount(); //valid

sa.balance = 10.0; //will not compile

ra.accountNumber = 10; //will not compile

ra.interestRate = 7.0; //will not compile

}

}

```

To make it interesting, I have put **RetirementAccount** into a different package than the rest of the classes. Let us now see the impact of the various access modifiers on these classes:

1. Since **Account** has default access, it is visible only in the **accts** package and thus **RetirementAccount** cannot access or extend the **Account** class.
2. Since **accountNumber** has default access, it is visible only in the **accts** package and thus, it will be inherited by all the subclasses of **Account** that belong to the same package. Thus, **accountNumber** will be inherited by **CheckingAccount** and **SavingsAccount**.
3. Since **balance** has protected access, it is visible in all the classes of the **accts** package and all the subclasses of the **Account** class irrespective of their package. Thus, it will be inherited by **CheckingAccount** and **SavingsAccount**. Since **Account** is not visible outside the **accts** package, making **balance** protected doesn't

seem to make much sense **but it does**. Observe what happens with `RetirementAccount`.

4. Since `RetirementAccount` extends `SavingsAccount` but belongs to a different package, it only inherits the public and protected (but not default) members of `SavingsAccount`. This means, the protected field `balance` is passed on to the `RetirementAccount` as well. Since the `interestRate` field of `SavingsAccount` has default access, it is not visible in `RetirementAccount` and therefore, is not inherited by `RetirementAccount`.
5. Even though the `balance` field of `Account` is visible and inherited in `RetirementAccount`, `sa.balance` will still not compile because `RetirementAccount` class does not own `SavingsAccount`'s `balance` as explained in the previous chapter. In technical terms, code of `RetirementAccount` class is not responsible for the implementation of `SavingsAccount` class and so, `RetirementAccount` cannot access `SavingsAccount`'s `balance`. It can access its own `balance`, which it inherits from `SavingsAccount`, and which is why `ra.balance` compiles fine, but not `SavingsAccount`'s `balance`.

Memory impact of access modifiers on sub classes



Even if a subclass does not inherit some of the instance variables of a superclass (because of access modifiers), a subclass object still consumes the space required to store all of the instance variables of the superclass. For example, when the JVM allocates memory for a `RetirementAccount` object, it includes space required to store all instance variables of `SavingsAccount` and `Account` irrespective of whether they are inherited in `RetirementAccount` or not. Thus, from a memory perspective, access modifiers have no impact.

So, on one hand, we are saying that a private member is not inherited by a subclass and on the other, we are saying that the subclass object does contain that member in its memory. This contradiction makes an intuitive understanding of the term "inheritance" difficult. Unfortunately, that is how it is. For example, in Section 8.2, The Java Language Specification says, "Members of a class that are declared private are not inherited by subclasses of that class." This shows that the JLS links the term inheritance with access modifiers and doesn't give any consideration to memory.

12.1.4 Inheritance of instance members vs static members [↳](#)

Inheritance of instance variables vs static variables [↳](#)

In an earlier example, you saw that the `Employee` class inherits instance as well as static variables of its superclass `Person`. However, there is a fundamental difference between the way instance and static variables are inherited. This difference is highlighted in the following code:

```
public class Person{  
    public String name;  
  
    public static  
  
        int personCount;  
}  
  
public class Employee extends Person{  
    //inherits instance as well as static fields of Person  
  
}
```

```

class TestClass{

    public static void main(String[] args){

        Person p = new Person();
        Employee e = new Employee();
        p.name = "Amy";
        e.name = "Betty";
        System.out.print(p.name+" ");
        System.out.println(e.name);

        Employee.personCount = 2;
        System.out.print(Person.personCount+" ");
        System.out.println(Employee.personCount);

    }
}

```

The above code generates the following output:

```

Amy  Betty
2 2

```

Observe that it prints different values for `name` - one for each object, but same value for `personCount`. What this means is that both the objects got their own personal copy of `name`, which they were able to manipulate without affecting each other's copy but the static variable `personCount` was shared by the two classes. When the code updated `personCount` of `Employee`, `personCount` of `Person` was updated as well. In fact, `Employee` did not get its own copy of `personCount` at all. It merely got access rights to `Person`'s `personCount`.

Inheritance of instance methods vs static methods [↳](#)

Since methods don't consume any space in an object (or a class), there is just one copy of a method anyway, irrespective of whether it is an instance method or a static method.

Conceptually, however, the difference that I highlighted above for variables also exists for methods. Conceptually, a subclass inherits its own copy of an instance method. This is proven by the fact that a subclass can **completely replace** the behavior of an inherited instance method for objects of the subclass by "**overriding**" it with a new implementation of its own without affecting the behavior of the superclass's implementation for objects of the superclass. On the other hand, a subclass merely gets access rights to a static method of its superclass and so, the subclass cannot change the behavior of the inherited static method. The subclass can "**hide**" the behavior of the superclass's static method by providing a new implementation but it cannot replace the super class's method.

Overriding and **hiding** are technical terms with precise meanings and are closely related to **polymorphism**. Their understanding is crucial for the exam as well as for being a good Java developer. I will dig deeper into these terms after we go over the nuts and bolts of extending classes and implementing interfaces.

12.1.5 Benefits of inheritance

First and foremost, Inheritance allows you to group classes by having them extend a common parent class. Functionality that is common to all such classes need to be defined only once in the parent class. For example, in the class hierarchy consisting of the **Account**, **CheckingAccount**, and **SavingsAccount** classes that I used at the beginning of this chapter, I put common functionality such as account number and method for checking balance in a parent class named **Account** and I put unique features of each of these account types in separate sub classes.

There are several advantages with this approach. Let's examine them one by one.

Code reuse

You can write logic for common fields only in one place and share that logic with all classes. In the above example, there is only one copy of the code to manage the account number. Since this logic is in the parent class, it is automatically inherited by all the child classes.

Having a common base class also makes it easy to add new functionality or modify the existing functionality that is common to all classes. This makes the code more extensible and maintainable overall.

Information hiding [↳](#)

Isolating the common features to a common parent class helps you hide the features of individual sub classes to processes that don't need to know of those features. As discussed before, more exposure means more risk of inadvertent dependency, i.e., tighter coupling, which is not desirable. For example, if you have a process that loops through all accounts and prints their account numbers and balances, then this process doesn't need to know about overdraft limit, interest rate, or maturity date. If you just pass an array of Accounts to this process, that should be enough. Something like this:

```
public class DumpAccountInfo{
    public static void printAccounts(Account[] accts){
        for(Account acct : accts){
            System.out.println(acct.getAccountNumber()+" "+acct.getBalance());
        }
    }
}
```

Of course, the array will include all kinds of **Account** objects (i.e. objects of sub-classes of Account class) but the process will only see them as **Account** objects and not as **CheckingAccount** , **SavingsAccount** , or **RetirementAccount** objects. The above code doesn't care whether an Account object is really a SavingsAccount or a CheckingAccount. It just prints the account number and its balance irrespective of what kind of account it is. This is possible only due to inheritance. Without the common parent class, you would have to have three different methods - one for each of the three account types -

to print this information.

Polymorphism

Finally, and most importantly, inheritance makes polymorphism possible. Polymorphism is a topic in its own right and I will discuss it soon in a section of its own.

12.2 Using super and this to access objects and constructors

12.2.1 Object initialization revisited

Recall that in the "Create and overload constructors" section of the "Working with Methods and Encapsulation" chapter, I talked about the four steps that a JVM takes while instantiating a class. Let's see how these steps are impacted when there is inheritance involved.

1. The first step was to load and initialize the class if it is not already loaded and initialized.
2. The second step was to allocate the memory required to hold the instance variables of the object in the heap space. Since the instance variables defined in a super class are also included in object of a subclass (whether they are accessible to the subclass or not is a different matter), the memory allocated by the JVM for a subclass object must include space for storing instance variables of the super class as well.
3. The third step was to initialize these variables to their default values. This means the inherited variables also need to be initialized to their default values.
4. The fourth step was to give that instance an opportunity to set the values of the instance variables as per the business logic of that class by executing

code written in instance initializers and constructors. This step gets a little more complicated when the class extends another class. Remember that the whole purpose of inheriting features (i.e. variables as well as methods) of a superclass is for the subclass to be able to use those features! It can use these features even at the time of its initialization. But to be able to do that, those features have to be initialized first. This means that subclass cannot initialize its own features unless features of its superclass have been initialized.

You can see where this is going. The superclass cannot be initialized before the superclass of the superclass is initialized and so on until there is no super class left. This chain of initialization will stretch until the Object class because Object is the root class of all classes and it is the only class that has no super class.

You have seen that the initialization is triggered when you try to create an instance of that class using the new keyword. When the JVM encounters the new keyword applied to a class, it invokes the appropriate constructor of that class. But as discussed above, the execution of this constructor cannot proceed until the initialization of the superclass is complete.

12.2.2 Initializing super class using "super"

To ensure the initialization of the fields inherited from the superclass, a constructor must first invoke exactly one of its super class's constructors. This is done using the **super(<arguments>)** syntax. For example:

```
class Person{
    String name;
    Person(String name){
        this.name = name;
    }
}

public class Employee extends Person{
    public Employee(String s){
        super(s);
    }
}
```

```
}

public static void main(String args[]){
    Employee ee = new Employee("Bob");
}

}
```

The question that you might ask at this point is what about **Person** class. This class has no extends clause and that means it implicitly extends **java.lang.Object**. So where is the call to **Object**'s constructor in **Person**'s constructor?

Good question. The call to the super class's constructor is so important that if a class's constructor doesn't call any of the super class's constructors explicitly, the compiler automatically inserts a call to the super class's default constructor in the first line of that constructor. The compiler doesn't check the constructors that the super class has. It just assumes the presence of the default constructor and inserts a call to that constructor. Thus, basically, **Person**'s constructor is modified by the compiler as follows:

```
class Person{
    String name;
    Person(String name){
        super();

        //<-- inserted automatically by the compiler

        this.name = name;
    }
}
```

This means that **Object**'s no-args constructor will be invoked first before proceeding with the execution of **Person**'s constructor. Armed with this knowledge, let us see what happens when I modify **Employee** as follows:

```
public class Employee extends Person{
    public Employee(String s){
        name = s;
    }
    public static void main(String args[]){
        Employee ee = new Employee("Bob");
    }
}
```

Any thoughts? That's right. It won't compile. Since `Employee`'s constructor doesn't call super class's constructor explicitly the compiler inserts a call to `super()`; on its own. This means `Employee`'s constructor really looks like this:

```
public Employee(String s){
    super(); //inserted automatically by the compiler
    name = s;
}
```

But `Person` does not have any no-args constructor! Recall that a default no-args constructor is provided by the compiler to a class only if the class does not define any constructor at all. In this case, `Person` does define a constructor explicitly and so the compiler does not insert a no-args constructor in the `Person` class. Thus, `Employee` will fail to compile.

Let me make one more change. What if I remove all constructors from `Employee` class?

```
public class Employee extends Person{
    public static void main(String args[]){
        Employee ee = new Employee();
    }
}
```

```
}
```

Again, recall the rule about the default constructor. Since **Employee** does not define any constructor explicitly, the compiler inserts a no-args constructor automatically, which looks like this:

```
public class Employee extends Person{
    public Employee(){ //inserted by the compiler
        super(); //inserted by the compiler
    }
    public static void main(String args[]){
        Employee ee = new Employee();
    }
}
```

Observe that this default no-args constructor also contains a call to **super()**; (See how important invoking a super class's constructor is?). But as discussed above, **Person** doesn't have a no-args constructor and therefore, the above code will not compile!

Since an object can be initialized only once, call to **super(<arguments>)**; can also be made only once. If you try to call it more than once, the code will fail to compile.

Invoking another constructor using **this(<arguments>)** ↴

In the previous chapter, you saw how a constructor can invoke another constructor of the same class using the **this(<arguments>);** syntax. Recall that invocation of another constructor is only allowed if it is the first statement of a constructor. But this poses a small problem. How can both - **this(<arguments>);** and **super(<arguments>);**, be the first

statement of a constructor at the same time? Well, they can't be. In fact, Java allows only one of the two statements in a constructor. In other words, if you call `this(<arguments>);` then you can't call `super(<arguments>);` and if you call `super(<arguments>);`, you can't call `this(<arguments>);`.

Note that this does not violate our original premise that the super class's features have to be initialized first before initializing subclass's features. If you call `this(<arguments>);`, you don't need to call `super(<arguments>);` anyway because the other constructor would have called `super(<arguments>);` and initialized the super class's features. If you call `super(<arguments>);`, then by prohibiting you from calling `this(<arguments>);`, Java prevents the invocation of the super class's constructor more than once.

Based on the above discussion, can you determine the output generated when `TestClass` is run from the command line:

```
class Person{
    String name;
    Person(String name){
        System.out.println("In Person's constructor ");
        this.name = name;
    }
}
class Employee extends Person{
    String empId;
    Employee(){
        this("dummy", "000");
        System.out.println("In Employee() constructor ");
    }

    Employee(String name, String empId){
        super(name);
        System.out.println("In Employee(name, empid) constructor ");
    }
}
```

```

}

class Manager extends Employee{
    String grade;
    Manager(String grade){
        System.out.println("In Manager(grade) constructor");
    };
    this.grade = grade;
}
}

class TestClass{
    public static void main(String[] args){
        Manager m = new Manager("A");
    }
}

```

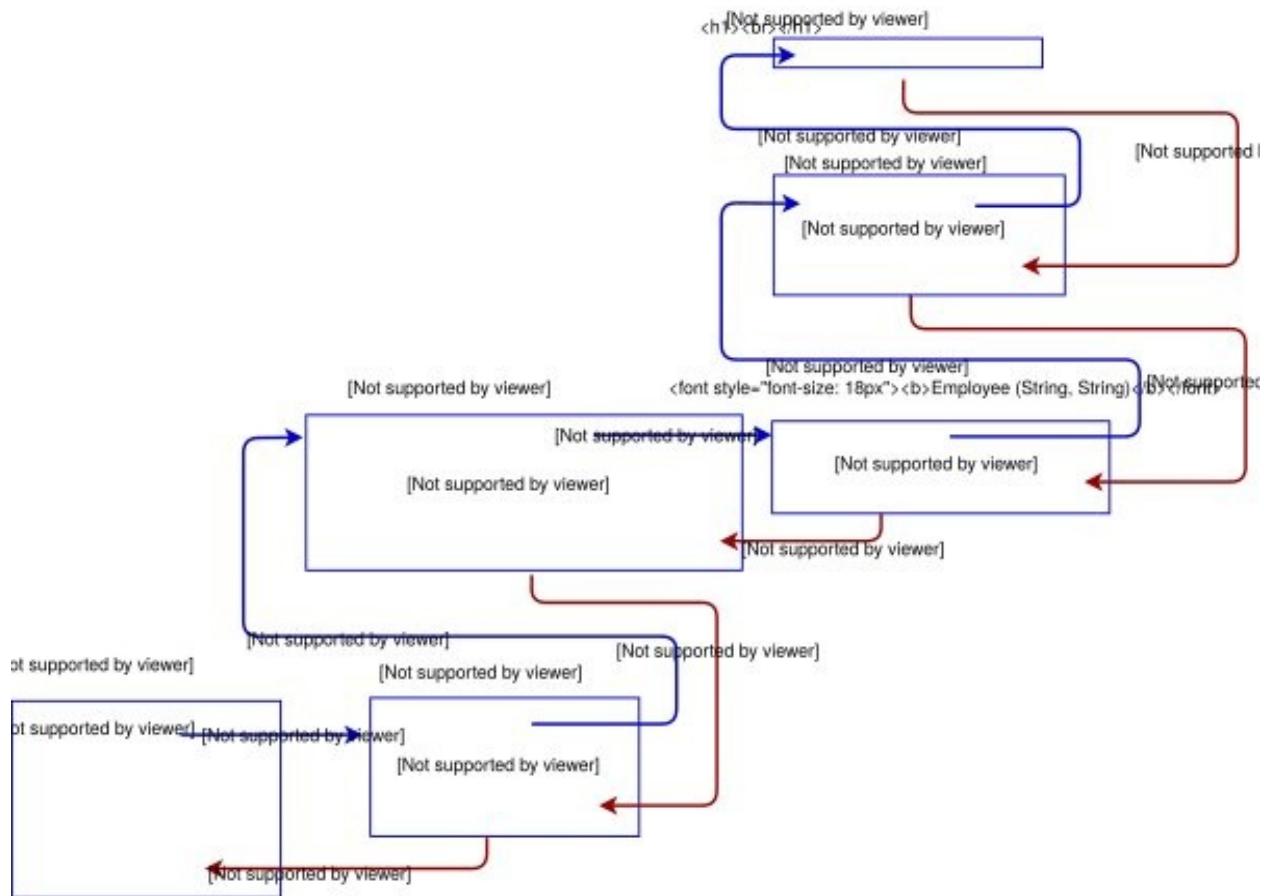
In `TestClass`'s main, an object of `Manager` class is being created using the only constructor it has. This constructor does not invoke superclass's constructor explicitly, therefore, the compiler inserts a call to `super()`; as the first statement of this constructor. This takes us to the no-args constructor of `Employee`. `Employee`'s no-args constructor invokes the two argument constructor of the same class explicitly using `this("dummy", "000")`. The two arguments constructor of `Employee` invokes its superclass's constructor using `super(name)`. This takes us to the `Person`'s single argument constructor. Inside the `Person(String)` constructor, there is no explicit call to its superclass's constructor. So the compiler inserts `super()`; as the first statement. This invokes `java.lang.Object`'s no-args constructor because `Person` implicitly extends `java.lang.Object`. Since `Object`'s class is the root class, this is the last constructor in the chain. `Object`'s constructor doesn't produce any output.

Now, we have to start unwinding this chain of calls. After finishing the execution of `Object`'s constructor, the control returns to next statement in `Person`'s constructor, which is `System.out.println("In Person's constructor ")`. Thus, the first line on the output is `"In Person's constructor"`. Next, the `name` argument is assigned to the `name` instance variable and the call returns back to the `Employee`'s constructor. The next statement in this constructor is `System.out.println("In`

`Employee(name, empid) constructor ");`, which prints "In Employee(name, empid) constructor " and the control goes back to Employee's no-args constructor, which prints "In Employee() constructor ". The control returns to Manager's single argument constructor, which prints, "In Manager(grade) constructor ". This completes the chain of constructor invocation executed during creation of a Manager object. Thus, the output is:

```
In Person's constructor
In Employee(name, empid) constructor
In Employee() constructor
In Manager(grade) constructor
```

This chain of calls is illustrated in the following figure.



12.2.3 Using the implicit variable "super" [↳](#)

When a method is overridden by a subclass, it is impossible for any unrelated class to execute the super class's version of that method on a subclass object. However, the subclass itself can access its super class's version using an implicit variable named "**super**" . Every **instance method** of a class gets this variable. It is very much like the other implicit variable that you saw earlier, i.e., "**this**" . While **this** refers to the current object, **super** refers to the members that this class inherits from its parent class. The following code shows how it is used:

```
class InterestCalculator{
    public double computeInterest(double principle, int
yrs, double rate){
        return principle*yrs*rate;
    }
}

class CompoundInterestCalculator extends InterestCalculator{
    public double computeInterest(double principle, int
yrs, double rate){
        return principle*Math.pow(1 + rate, yrs) - pr
inciple;
    }

    //invoke this method from TestClass's main

    public void test(){
        double interest = super.computeInterest(100, 2,
0.1);
        System.out.println(interest); //prints 20.0

        interest = computeInterest(100, 2, 0.1);
    }
}
```

```

        System.out.println(interest); //prints 21.00000
0000000014

    }

}

public class TestClass{
    public static void main(String[] args){
        CompoundInterestCalculator cic = new CompoundInterestCalculator();
        cic.test();
    }
}

```

The call to `super. computeInterest(100, 2, 0.1);` in the above code causes `InterestCalculator`'s version to be invoked while the call to `computeInterest(100, 2, 0.1)` causes `CompoundInterestCalculator`'s version to be invoked

The variable `super` is available only in instance methods of a class and can be used to access **any inherited member (static as well as instance)** of the super class. Are you now thinking about forming a chain of supers? Something like `super.super.someMethod()`? No, you can't do that because there is no member named `super` in the super class. In fact, you can't define a variable or a method named `super` in any class. This begs the question that how can a class access the grandparent's version of a method that is defined in its grand parent as well as in its parent class? For example, what if you have the following grandchild class and you want to access `InterestCalculator`'s version of `computeInterest` in this class:

```

class SubCompoundInterestCalculator
    extends CompoundInterestCalculator{

    public void test(){

        //invokes CompoundInterest's computeInterest
    }
}

```

```

super.

computeInterest(100, 2, 0.1);

//can't do super.super

//super.super.computeInterest(100, 2, 0.1);

}

}

```

The answer is you can't. There is no way to go more than one level higher. In other words, a subclass can only access its immediate super class's version of methods using super.

Note that this limitation applies only to methods that are overridden by the parent class. If the parent class does not override a method of its super class, then the parent class inherits that method from the grandparent and the child class can access the grandparent's version of the method as if it were the parent class's version.

12.2.4 Order of initialization summarized

You have now seen all the steps, from loading the class to executing a constructor of the class, that a JVM takes to create an object of a class. Here is the order of these steps for quick reference:

1. If there is a super class, initialize static fields and execute static initializers of the super class in the order of their appearance (performed only once per class)
2. Initialize static fields and execute static initializers of the class in the order of their appearance (performed only once per class)

3. If there is a super class, initialize the instance fields and execute instance initializers of the super class in the order of their appearance
4. Execute super class's constructor
5. Initialize the instance fields and execute instance initializers of the class in the order of their appearance
6. Execute class's constructor

The following example illustrates the above steps:

```

class A {
    static{ System.out.println("In A's static initializer"); }

    A(){ System.out.println("In A's constructor"); }

    { System.out.println("In A's instance initializer")
; }
}

public class B extends A {
    static{ System.out.println("In B's static initializer"); }

    { System.out.println("In B's instance initializer")
; }

    B(){ System.out.println("In B's constructor"); }

    public static void main(String[] args) {
        System.out.println("In B.main()");
        B b = new B();
        B b2 = new B(); //creating B's object again
    }
}

```

The following output is generated when class B is run:

```
In A's static initializer
In B's static initializer
In B.main()
In A's instance initializer
In A's constructor
In B's instance initializer
In B's constructor
In A's instance initializer
In A's constructor
In B's instance initializer
In B's constructor
```

You should observe the following points in the above output:

1. static initializers of A and B are invoked before B's main() is executed because the JVM needs to load the class before it can invoke a method on it.
2. Static initializers of A and B are invoked only once even though two objects of B are created.
3. Instance initializer of A is executed before A's constructor even though the instance initializer appears after the constructor in A's code.
4. While creating an object of B, A's instance initializer and constructor are invoked before B's instance initializer and constructor.
5. Instance initializer and constructor of A are invoked each time an object of B is created.

Order of execution of various members of a class and its superclass can sometimes get very difficult to trace. However, the exam does not try to trick you too much on this topic. If you can follow the above example, you will not have any problem with the code in the real exam.

12.3 Create and extend abstract classes

12.3.1 Using abstract classes and abstract methods [↑](#)

I talked about **abstract class** as a **type of type** briefly in the "Kickstarter for Beginners" chapter. An abstract class is used when you want to capture the common features of multiple related object types while knowing that no object that exhibits only the features captured in the abstract class can actually exist. For example, if you are creating an application that deals with cats and dogs, you may want to capture the common features of cats and dogs into a class called Animal. But you know that you can't have just an Animal. You can have a cat or a dog but not something that is just an animal without being a cat, a dog or some other animal. In that sense, Animal is an **abstract concept**. Furniture is another such concept. If you want to have a Furniture, you have to get a chair or a table because Furniture doesn't exist on its own. It is an abstract concept.

Abstract classes are used to capture such abstract concepts. An abstract class can be defined by using the keyword **abstract** in its declaration. For example, the following is how you can define **Furniture**:

abstract

```
class Furniture{  
    private String material;  
    //public getter and setter  
}
```

Observe that **Furniture** is pretty much like a regular class with public and private fields and methods. However, since it is declared abstract, it cannot be instantiated. Thus, if you try to do **new Furniture()**, the compiler will generate an error saying, "**Furniture is abstract; cannot be instantiated**".

But Chair and Table are concrete concepts. They exist on their own. Since both of them are types of Furniture, you can model them as two classes that extend Furniture.

```
class Chair extends Furniture{
```

```
//fields and methods specific to Chair

}

class Table extends Furniture{
    //fields and methods specific to Table

}
```

Since **Chair** and **Table** extend **Furniture**, they will inherit all of the features of **Furniture** just like any subclass inherits all of the features of its super class.

Note that classes that are not abstract are also called "concrete" classes because they represent real objects.

Adding an abstract method to an abstract class [⬆](#)

Let's say every piece of furniture in a store has to provide a method for its assembly. Since the steps to assemble a chair will be different from steps to assemble a table, we can't include the steps for their assembly in Furniture class. In other words, what we are saying is that the declaration of the method to assemble is common to all furniture but their implementation is unique to each type of furniture. Java allows you to capture declaration without implementation in the form of an **abstract method**. An abstract method is declared by applying the **abstract** keyword to the method declaration. It must not have a method body either. For example:

```
abstract class Furniture{
    private String material;
    //public getter and setter

    public abstract
```

```
void assemble(); //ends with a semicolon, no opening  
and closing curly braces  
}
```

The benefit of adding abstract methods to a class is that other components can work with all objects uniformly. For example, a class that assembles furniture doesn't need to worry about whether it is assembling a Chair or a Table. It can simply call `assemble()` on any kind of furniture that it gets. Something like this:

```
class FurnitureAssembler{  
    public static void assembleAllFurniture(Furniture[]  
allFurniture){  
        for(Furniture f : allFurniture){  
            f.assemble();  
        }  
    }  
}
```

But adding an abstract method to the parent class has an impact on child classes in that the child classes are now obligated to provide an implementation for the abstract method. For example, we will now need to update the code for our Chair and Table classes as follows:

```
class Chair extends Furniture{  
    //fields and methods specific to Chair  
  
    public void assemble(){  
        System.out.println("Assembling chair!");  
    }  
}  
class Table extends Furniture{
```

```
//fields and methods specific to Table

public void assemble(){
    System.out.println("Assembling table!");
}
}
```

This makes sense because if a piece of furniture actually exists then we must be able to assemble it as per our definition of **Furniture**. The only reason why a subclass of **Furniture** may be unable to have a method to assemble is if that subclass itself is abstract! For example, what if you have a subclass of **Furniture** called **FoldingFurniture**? **FoldingFurniture** is an abstract concept and can be modeled as an abstract class that extends **Furniture**. Since it is abstract, it can get away without implementing the **assemble** method. But any concrete subclass of **FoldingFurniture** will have to provide an implementation for the **assemble** method.

If you are confused about how the code for **FurnitureAssembler** can invoke the **assemble** methods of Chairs and Tables when it does not even know about their existence, don't worry. I will talk about it in detail soon in the section about **polymorphism**.

12.3.2 Using final classes and final methods

You have seen the usage of **final** keyword while declaring variables. It implies that the variable is a **constant**, i.e., the value of the variable does not change throughout the execution of the program. The **final** keyword can also be applied to classes and methods (static as well as instance) and implies something similar. It means that the behavior of the class or the method cannot be changed by any subclass.

Observe that **final** is diametrically opposite of **abstract**. You make a class or a method abstract because you want a subclass of that class to provide different implementation for that class or that method as per the business logic of the subclass. On the other hand, you make a class or a method final because you

don't want the behavior of the class or of the method to be changed at all by a subclass. In technical terms, we say that a final class cannot be **subclassed** and a final method cannot be **overridden**. I will discuss overriding in detail soon but you can now see why **abstract** and **final** cannot go together.

A final class doesn't necessarily have to have a final method. But practically, since a final class cannot be extended, none of its methods can be overridden anyway. The important thing is a final class cannot have an abstract method. Why? Because there is absolutely no possibility of that abstract method ever getting implemented. It follows that if a final class inherits an abstract method from any of its ancestors, it must provide an implementation of that method.

Here is an example of a final class:

```
public final class Chair extends Furniture{  
  
    //since assemble is an abstract method in Furniture  
    //, it must be implemented in this class  
  
    public void assemble(){  
        System.out.println("Assembling Chair");  
    }  
}
```

Observe that **Chair** doesn't have any final method and that it must implement the abstract method **assemble** that it inherits from its superclass.

A **final** method is defined similarly:

```
public abstract  
  
class Bed extends Furniture{  
  
    //not necessary to implement assemble() because Bed  
    //is abstract
```

```

public final

int getNumberOfLegs(){
    return 4;
}

public static final void make(Bed b){
    System.out.println("Making Bed: "+b);
}
}

public final class DoubleBed extends Bed{

    //must implement assemble because DoubleBed is not abstract

    public void assemble(){
        System.out.println("Assembling DoubleBed");
    }

    //Can't override getNumberOfLegs here because it is final in Bed

    final

    int getHeight(){
        return 18;
    }
}

```

Observe that even though **Bed** has a final method, **Bed** itself is not only not final, it is abstract. I made it abstract just to illustrate that any class irrespective of whether it is final, abstract, or neither, can contain a final method. Java standard class library includes several final classes. Most important of them

is `java.lang.String` class, which I will talk about at length in another chapter.

12.3.3 Valid combinations of access modifiers, abstract, final, and static [↳](#)

Impact of access modifiers on abstract and final [↳](#)

The usage of abstract makes sense only in the presence of inheritance. If something is never inherited then there is no point in talking about whether it is abstract or final. You also know that a private member is never inherited. Based on this knowledge can you tell what will happen with the following code:

```
abstract class Sofa {  
    private abstract void recline();  
}
```

That's right. The compiler will reject the above code with the message, "[illegal combination of modifiers: abstract and private](#)". Because a private method is never inherited, there is no way any subclass of `Sofa` can provide an implementation for this abstract method. What if the `recline` method were protected or default? It would have been ok in that case because it is possible for a subclass to inherit methods with protected and default access.

Let us now see what happens when you make a private method final:

```
class Sofa {  
    private final void recline(){  
    }  
}
```

What do you think will happen? It won't compile, you say? Wrong! If a method

cannot be inherited at all by any subclass, is it even possible to override it? No. Thus, a private method is, practically, already final! The compiler accepts the above code because there is no contradiction between private and final. Marking a private method as final is, therefore, not wrong but rather redundant.

abstract and static [↳](#)

The abstract keyword is strictly about overriding and static methods can never be overridden . Therefore, abstract and static cannot go together. Thus, the following code will not compile:

```
abstract class Bed extends Furniture{  
    static abstract void getWidth();  
}
```

The compiler will generate an error message saying, "illegal combination of modifiers: abstract and static".

final and static [↳](#)

Although a static method cannot be overridden, it is nevertheless inherited by a subclass and can be hidden by the subclass. I will discuss the difference between overriding and hiding in detail later but for now, just remember that final prevents a subclass from hiding the super class's static method. Thus, the following code will compile fine:

```
class Bed{  
    static final int getWidth(){  
        return 36;  
    }  
}
```

But the following will not because `getWidth` is final in `Bed` :

```
class DoubleBed extends Bed

{
    static int getWidth(){ //will not compile

        return 60;
    }
}
```

Summary of the application of access modifiers, final, abstract, and static keywords [↳](#)

Based on the previous discussion, let me summarize the rules of abstract, concrete classes, and final classes and methods:

1. An **abstract class** doesn't necessarily have to have an abstract method but if a class has an abstract method, it must be declared abstract. In other words, a concrete class cannot have an abstract method.
2. An abstract class cannot be instantiated irrespective of whether it has an abstract method or not.
3. A **final class** or a **final method** cannot be abstract.
4. A **final class** cannot contain an **abstract method** but an **abstract class** may contain a **final method** .
5. A **private** method is always **final** .
6. A **private** method can never be **abstract** .
7. A **static method** can be **final** but can never be **abstract** .

The exam tests you on various combinations of access modifiers and final, abstract, and static keywords as applied to classes and methods. You should be very clear about where you can and cannot

apply these modifiers.

The exam, however, does not try to trick you on the order of access modifiers and final and abstract keywords. For example, you will not be asked to pick the right declaration out of, say, the following three:

```
public abstract void m();  
final public void m(){ }  
abstract protected void m();
```

Nevertheless, you should know that the right sequence of modifiers in a method declaration is as follows:

```
<access modifier > <static > <final or abstract >  
<return type > methodName(<parameter list >)
```

If you ever get confused about their order, just recall the signature of the main method as:

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

Of course, the main method does not have be final. I have mentioned it above to show the position of final in a method declaration.

12.4 Enable polymorphism by overriding methods

12.4.1 What is polymorphism

In simple terms, **Polymorphism** refers to the ability of an object to exhibit behaviors associated with different types. In other words, if the same object behaves differently depending on which "side" of that object you are looking at, then that object is **polymorphic**. For example, if you model apples using an Apple class that extends a Fruit class, then an apple can behave as an Apple as well as a Fruit. Later on, if you have a RedApple class that extends the Apple

class, then a red apple will behave as an Apple and a Fruit besides behaving as a RedApple. Similarly, an object of a StockPrice class that you saw earlier behaves as a Readable as well as a Movable besides behaving as a StockPrice! Thus, apples and stock prices are polymorphic objects.

It is very important to understand that the actual object doesn't change at all. The object itself always remains the same. A red apple will always be a red apple. It doesn't suddenly morph into an apple or a fruit. It has always been an apple and a fruit besides being a red apple. It follows that if an object doesn't already support a particular behavior, it won't suddenly start supporting that behavior. A red apple will never morph into a green apple no matter what you do.

Remember that when we talk about the behavior of an object in the context of polymorphism, we are essentially talking about its instance methods. We are not talking about its static methods because static methods define the behavior of a class and not the behavior of the object. (Yes, it is true that you can access a static method using an object reference instead of the class name but that is just a peculiarity of the Java language and has nothing to do with polymorphism). We are not talking about instance variables either because variables merely store data (either primitive data or references to other objects) and do not possess any behavior. Also, the variables would not be visible to any other class anyway if the class is well encapsulated.

Polymorphism in Java [↳](#)

Java allows objects to be polymorphic by letting a class extend another class and/or implement interfaces. Since every class in Java implicitly extends `java.lang.Object`, you can say that every object is polymorphic because every object exhibits the behavior of at least two classes (except an instance of `java.lang.Object`, of course).

Thus, an object can be as polymorphic as the number of classes it extends (directly as well as indirectly) and the number of interfaces it implements (directly as well as indirectly through ancestors).

Importance of Polymorphism [↳](#)

In the "Kickstarter for Beginners" chapter I talked about how a component can be easily replaced with another component if they promise to honor the same behavior. This is made possible only because of **polymorphism**. If an object is allowed to behave only as a single type, then you can never replace that object with another object of a different type.

For example, if all you want is a Fruit, then it shouldn't matter whether you are given an Apple or an Orange. Both of them promise to honor the behavior described by Fruit and are, therefore, equally acceptable fruits. The situation is illustrated by the following code:

```
abstract class Fruit{ //must be declared abstract because it has an abstract method

    abstract void consume();
}

class Apple extends Fruit{
    void consume(){
        System.out.println("Consuming Apple...");
    }
}

class Orange extends Fruit{
    void consume(){
        System.out.println("Consuming Orange...");
    }
}

class Person{
    void eatFruit(Fruit f){
        f.consume();
    }
}

class TestClass{
    public static void main(String[] args){
        Apple a = new Apple();
```

```

        Orange o = new Orange();
        Person p = new Person();
        p.eatFruit(a);
        p.eatFruit(o);
    }
}

```

Observe that a Person object has no knowledge about what it is eating. All it cares about is Fruit. As long as you pass it a Fruit, it is fine. You can easily pass an Orange to a Person instead of an Apple. This is possible only because Apple and Orange are polymorphic, they behave like a Fruit besides behaving like an Apple or an Orange respectively.

Without polymorphism, your code would look something like this:

```

class Apple{
    void consume(){
        System.out.println("Consuming Apple...");
    }
}

class Orange{
    void consume(){
        System.out.println("Consuming Orange...");
    }
}

class Person{
    void eatApple(Apple a){
        a.consume();
    }
    void eatOrange(Orange o){
        o.consume();
    }
}

class TestClass{
    public static void main(String[] args){
        Apple a = new Apple();

```

```

        Orange o = new Orange();
        Person p = new Person();
        p.eatApple(a);
        p.eatOrange(o);
    }
}

```

Observe that in the absence of a common Fruit class, Person must have two different methods - one for eating Apple and one for eating Orange. The user of the Person class, i.e., TestClass, also has to call two different methods. What if you wanted to feed a banana to a person? You would have to code a new method in Person to accept a Banana and also have to change the code in TestClass. And you know these are not the only kinds of fruit in the world, right? You get the idea. Imagine how many eat methods you would have to write!

But that is not the only problem. Let us say you code a method for each of the fruits that you know about in Person class and deliver this class to other people for use. What if you or someone else comes to know about an entirely new kind of fruit and wants to feed that fruit to a Person object? You can't change the code of the Person class at this stage because then you would have to make everyone using the old class get this new version of the Person class from you. Depending on the diversity of people in different projects, this could become a nightmare every time a new fruit is discovered.

If you think logically, it shouldn't matter to a Person if a new kind of fruit is discovered. As long as it is a fruit, a Person should be able to accept it. This is where Polymorphism shines. In our first design **Person**'s eat method accepted just a **Fruit**. If you find a new fruit, say Custard Apple (amazing fruit, btw), all you have to do is to define a **CustardApple** class that extends **Fruit** and pass objects of this class to **Person**'s **eatFruit(Fruit)** method. No change is required in the **Person** class. This is possible only because **CustardApple** honors the contract defined by **Fruit** by saying it extends **Fruit**.

Another important thing about Polymorphism is that it allows components to be switched not only at compile time but also at **run time**. You can actually introduce completely new classes that extend existing classes (or implement existing interfaces) into an application that is already running. An existing class

or interface serves as a contract for a specific behavior and if a new class promises to honor this contract by extending that class or by implement that interface, then any code can make use of instances of this new class in places where it requires instances of the existing class or interface.

Always remember that the whole objective of polymorphism is to enable classes to become standardized components that can be easily exchanged without any impact on other components. The various seemingly confusing rules that you will soon see, only exist to achieve this goal. If you ever get confused about a particular rule, always think about how it affects the interoperability of one component with another.

The ability to transparently replace an object with another is called "substitutability". The substitutability principle states that objects of a type may be replaced with objects of its subtype without altering any of the desirable properties of the program.

Polymorphism and Inheritance [↳](#)

As you saw above, methods of a class or an interface basically form a **contract** that a subclass or the implementing class promises to fulfill. When a class says that it extends another class or says that it implements an interface, it agrees to have the methods the superclass or the interface has. However, the exact things that these methods must do is not covered by this contract. In other words, the contract is not so detailed as to affect the internal logic of the methods. The internal logic of a method is only governed by an informal contract that is implicit in the name of a method. For example, if the superclass declares a method named `computeInterest`, it is reasonable to expect that the subclass' implementation would compute some kind of interest in this method. This is not an issue with abstract methods because a subclass is required to implement abstract methods on its own, but it may be a problem in case of non-abstract methods. What if the superclass computes interest in one way but the subclass wants to compute interest in another way? If you think about it, this is an important aspect of componentizing a class. You want to componentize classes so that you could easily replace one component with another. But why would you replace one component with another if they behaved exactly the

same? You wouldn't.

What you really want is the ability to replace a component with another component with which you can interact in the same way, but which behaves differently. This is exactly what polymorphism in Java achieves. Java allows a subclass to provide its own implementation of the method if it does not like the behavior provided by the inherited method. The technical term for this is "**overriding**". A subclass is free to override a method that it inherits with its own implementation.

12.4.2 Overriding methods [↳](#)

As discussed earlier in this chapter, only **non-private instance methods** can be **overridden**. There are several rules that you need to follow while overriding a method. These rules govern the **accessibility**, **return type**, **parameter types**, and the **throws clause** of the method. The best way to understand and remember these rules is to keep in mind that these rules are there to make sure that an object of a class can be replaced with an object of a subclass without breaking existing code. The objective is to be able to replace one component with another transparently while giving flexibility to a subclass in implementing its methods.

Whenever you override a method in a subclass, think about what will happen to the code that depends on the superclass object and if you pass it a subclass object instead of the superclass object. If this replacement does not require the code to be recompiled, then you have overridden the method correctly. From this perspective, the following rules are applicable for method hiding (which happens for non-private static methods) as well.

Let me now list all of the rules and then I will show you the reasoning behind these rules with an example. (Remember that an "overridden method" means the method in the superclass and an "overriding method" means the method in the subclass).

1. **Accessibility** - An overriding method must not be less accessible than the overridden method. This means that if the overridden method is **protected**, you can't make the overriding method default because

default is less accessible than **protected**. You can make the overriding method more accessible. Thus, you can make it **public**.

2. **Return type** - The return type of the overriding method must either be the same as the return type of the overridden method or it must be a sub type. For example, if the return type of the overridden method is **Fruit**, then the return type of the overriding method can either be **Fruit** or any subclass of **Fruit** such as **Apple** or **Mango**. This is called "**covariant returns**".

In the case of primitives, the return type of the overriding method must match exactly to the return type of the overridden method. This is because there is no sub type relation between any two primitive types. For example, even though a **short** is smaller than an **int**, neither one is a subtype of the other.

3. **parameters** - The list of parameters that the overriding method takes must match exactly to that of the overridden method in terms of types and order (parameter names don't matter). Indeed, if there is a difference between the types and/or order of parameters, it would not be an override but an overload. I have already discussed overloading of methods.
4. **throws clause** - An overriding method cannot put a wider exception (i.e. a superclass exception) in its throws clause than the ones present in the throws clause of the overridden method. For example, if the overridden method throws **IOException**, the overriding method cannot throw **Exception** because **Exception** is a superclass of **IOException**. The overriding method may throw a subclass exception such as **FileNotFoundException**.

The overriding method cannot throw a new exception that is not listed in the throws clause of the overridden method either.

The overriding method may decide to not have a throws clause altogether though.

Note that this rule applies only to checked exceptions because these are the only ones the compiler cares about. There is no rule regarding unchecked

exceptions.

Let's see how these rules works with respect to the following classes:

```
class InterestCalculator{
    Number computeInterest(double principle, double yrs
, double rate) throws Exception {
        if(yrs<0) throw new IllegalArgumentException(
"yrs should be > 0");
        return principle*yrs*rate;
    }
}

class Account{
    double balance; double rate;

    Account(double balance, double rate){
        this.balance = balance;
        this.rate = rate;
    }

    double getInterest( InterestCalculator ic, double yrs ){
        try{
            Number n = ic.computeInterest(balance, yrs,
rate);
            return n.doubleValue();
        }catch(Exception e){
            e.printStackTrace();
        }
        return 0.0;
    }
}

class AccountManager{
    public static void main(String[] args){
        Account a = new Account(100, 0.2);
        InterestCalculator ic = new InterestCalculator()
```

```

;
    double interest = a.getInterest(ic, 2);
    System.out.println(interest);
}

}

```

We have an **Account** class that uses an **InterestCalculator** object to compute interest. We also have an **AccountManager** class that manages accounts and also the way accounts compute interest. **AccountManager** knows the interest calculation logic that accounts must use to compute interest. As of now, **AccountManager** creates an **InterestCalculator** object and passes it on to an **Account** object to compute interest.

Our objective is to make an Account return compound interest. To achieve this, all we need to do is to create a subclass of **InterestCalculator** named **CompoundInterestCalculator** and pass an instance of this new class to **Account**'s **getInterest** method from **AccountManager**'s **main** :

```

public class CompoundInterestCalculator extends InterestCalculator {
    public Double computeInterest(double principle, double yrs, double rate){
        return principle*Math.pow(1 + rate, yrs) - principle;
    }
}

class AccountManager{
    public static void main(String[] args){
        Account a = new Account(100, 0.2);
        InterestCalculator ic = new CompoundInterestCalculator();

        double interest = a.getInterest(ic, 2);
        System.out.println(interest);
    }
}

```

}

Observe that the `Account` class has no idea about the change that we did. The `Account` class does not need recompilation because it does not even know about the existence of `CompoundInterestCalculator`. It interacts with the `CompoundInterestCalculator` object as if it were just another `InterestCalculator` object.

Now, let's see how violating the rules of overriding affects `Account` class.

1. **Accessibility** - If you make `computeInterest` in `CompoundInterestCalculator` private, the JVM will be facing two contradictory directions. The call to `ic.computeInterest(...)` in `Account` expects the JVM to invoke the method on the actual object referred to by `ic`. But the type of the actual object is `CompoundInterestCalculator` and `CompoundInterestCalculator` expects the JVM to reject any attempt to invoke this method from outside the `CompoundInterestCalculator` class. There is no way to obey both the directions. Therefore, the compiler rejects the restriction on making this method private.

Making the method public in `CompoundInterestCalculator` is ok because the `Account` class doesn't care if classes from other packages are able to access `CompoundInterestCalculator`'s `computeInterest`.

2. **Return type** - When `Account`'s `getInterest` calls `ic.computeInterest`, it expects to get back a `Number`. So, when `CompoundInterestCalculator`'s `computeInterest` returns a `Double`, `Account` has no problem because a `Double` is-a `Number`. But if you change the return type of `CompoundInterestCalculator`'s `computeInterest` to, say, `Object`, code in `Account`'s `getInterest` will fail because an `Object` is not necessarily a `Number` and it won't be able to call `doubleValue` on an `Object`.

3. **parameter types** - This is kind of obvious. If you change the parameter types, the compiler will consider this a completely different method and not a replacement for the method in `InterestCalculator`. The JVM will never invoke this new method and `Account` will still be computing simple interest instead of compound interest.
4. **throws clause** - Since the `computeInterest` method of `InterestCalculator` says that it may throw an `Exception`, the caller of this method, i.e., `Account`'s `getInterest` is prepared to deal with this exception. It has a catch block that is meant to catch an `Exception`. The code in `getInterest` will work even if the `getInterest` method throws `IllegalArgumentException` because `IllegalArgumentException` is an `Exception` and it will be caught by `catch(Exception)` clause. It doesn't have any problem if `computeInterest` doesn't throw any exception at all either. But if the overriding method decides to throw a super class exception such as `Throwable`, then the code in `Account`'s `getInterest` will fail because it is not prepared to handle a `Throwable`. You will have to update the catch clause to `catch(Throwable)` and recompile `Account`.

I suggest that you write a similar example to validate the above rules in case of hiding of a static method.

12.4.3 Invalid overrides [🔗](#)

Overriding a static method with an instance method and vice versa [🔗](#)

Java does not allow a subclass to change the static-instance type of a non-private method defined in the superclass. Thus, the following code will not compile:

```
class Foo{  
    static void foo(){  
    }  
}
```

```
void moo(){  
}  
}  
  
class Bar extends Foo {  
    void foo(){ // will not compile because overridden  
method is static  
  
}  
  
    static void moo(){ // will not compile because over  
riding method is static  
  
}  
}
```

Overriding private methods [↳](#)

Since private methods are not inherited by a subclass, it is not possible to override them. But a subclass is allowed to have a method with the same signature as a private method of a superclass.

Overriding final methods [↳](#)

By marking a method as `final`, you prohibit subclasses from overriding that method. Therefore, if you try to have a method with the same signature as a final method of a superclass, it will not compile. The `final` keyword works similarly for static methods as well. It prevents you from hiding a static method in the subclass.

Note that final works very differently in case of fields. A subclass is allowed to have a field by the same name even if that field is declared final in the superclass. As discussed earlier, final just prevents you from changing the value of that field.

12.5 Utilize polymorphism to cast and call methods

12.5.1 Type of reference and type of an object

In the 'Kickstarter for Beginners' chapter I discussed the difference between a reference and an object in detail. I used the analogy of a remote and a TV to explain that a reference is a handle with which you access an object. Just as the remote and the TV it controls are two different things, the reference and the object to which the reference refers are also two different things.

The type of the variable is specified at the time of defining the variable. For example, `String str;` defines a variable `str` of type `String`. Specifying the type of the variable is important because that is how the compiler is able to determine the variables and methods that you are allowed to access through that variable. This is what makes Java a strongly typed language. A Java compiler will not allow you to invoke methods that are not available in the type of the reference that you are using.

An object, on the other hand, is created at run time. Although the compiler may sometimes be able to guess the type of an object that will be created at run time (for example, when you instantiate a class using the `new` keyword), it is not always the case. For example, while compiling the following method, the compiler has no way of knowing the type of actual object that is referred to by the method parameter `obj` :

```
public class Test{
    public void foo(Object obj){
        //do something with obj
    }
}
```

The compiler does not know who may be calling `foo` and what they may be passing to it as an argument. All it knows is that `obj` will refer to an `Object`

but it has no idea about the exact type of that object. For example, it is possible for another developer to write a method in another class that calls `Test`'s `foo` with a `String` or an `Integer` as an argument:

```
public class Other{
    public void bar(){
        new Test().foo("hello");
        new Test().foo(new Integer(1));
    }
}
```

The above example actually shows polymorphism in action. The variable `obj` in `foo` could potentially behave like a `String`, an `Integer`, or like any other subclass of `Object` depending on the actual class of the object that is passed to `foo`.

12.5.2 Bridging the gap between compile time and run time [↳](#)

To write polymorphic code, it is critical to understand that polymorphism happens at **run time** i.e. at the time of execution of the code, but the code that makes it happen is written at **compile time**. There is a difference between the amount of information that the compiler has about the program and the amount of information that the JVM has while it is executing that program. For example, when you define a variable, the compiler only knows the type of that variable, but it cannot know the exact object to which it refers because objects are created only when the program is run, i.e., at run time. Now, Java is a strongly typed language, which means that the type of the variable is defined at compile time itself and it cannot be changed once defined. This means, the compiler must make sure that the code does not assign an object of one type to a variable of another type. It must not allow the code to invoke methods that are not supported by the type of the variable. However, relying only on the type information available at compile time will impose too many restrictions on the code. For example, if you define a reference variable of type `java.util.List`, the compiler must not allow the code to invoke methods defined in some other class using that variable. However, this rigidity will make polymorphism impossible.

because the whole premise of polymorphism is that an object may behave differently at run time than what was promised at compile time.

You can see that there are two contradictory requirements that the compiler needs to fulfill. On one hand, you want the compiler to type check the code to make sure the code doesn't invoke random methods using a reference and on the other hand you want the compiler to let code violate type checking in some cases. Java tries to meet both the objectives by letting the compiler make certain assumptions and by letting the programmer give guarantees to the compiler that it is not trying to do anything fishy.

Let us now see how these two things are done in practice.

The "is a" test [🔗](#)

The **is-a** test is an intuitive test to determine the relationship between two reference types. It allows you to check whether there is a parent child relationship between two reference types and if there is, which one of the two types is the parent class, and which one is the child class. For example, it makes sense if you say that a Dog is a Pet or a Cat is a Pet but not if you say that a Dog is a Cat. A similar relationship exists between an Apple and a Fruit or a Mango and a Fruit. An Apple is a Fruit, A Mango is a Fruit, but an Apple is not a Mango. If you were to model Apple, Mango, and Fruit as classes, you would put all common features of fruits in a **Fruit** class and you would make **Apple** and **Mango** classes extend **Fruit**. The importance of establishing this relationship is that it allows you to use a subclass object anywhere in place of a superclass object. Indeed, if you promised to give someone a fruit, you can certainly give them an apple because an apple is a fruit. The reverse is obviously not true. If you promised to give someone an apple, you cannot just give them any kind of fruit. It has to be an apple. But if you had a Macintosh apple, you could give them that because a Macintosh apple is an apple.

The Java compiler recognizes if an **is-a** relationship exists between two types and allows you to assign a subclass object to a reference of a superclass type. For example:

```
class Fruit{ }
class Apple extends Fruit{ }
class Mango extends Fruit{ }
```

...

```
Apple a = new Apple();
Mango m = new Mango();
Fruit f1 = a; //ok, because Apple is a Fruit
```

m = a; //will NOT compile because an apple is not a mango

```
Fruit f2 = m; //ok, because Mango is a Fruit
```

m = f1; //will NOT compile because all fruits are not mangoes

m = f2; //will NOT compile because all fruits are not mangoes

Note that the actual objects are created only at run time but the compiler knows that the variable **a** will always refer to an **Apple** (i.e. to an object of a class that satisfies the is-a test with **Apple**) at run time because it would not compile the code that tries to assign anything else to **a**. This is proven by the fact that the line **Mango m = a;** does not compile. Similarly, the compiler knows that **f1** and **f2** will always point to a **Fruit** at run time because it will not allow you to write code that tries to assign anything else to **f1** and **f2**.

The last line, i.e., **m = f2;** is important. Even though we know that **f2** really does point to a **Mango**, the compiler refuses to compile this line. Remember that the compiler does not execute any code and therefore, the compiler doesn't know that **f2** really does point to a **Mango**. It only knows that **f2** may refer to any kind of **Fruit**. That fruit could be a **Mango** or an **Apple**. Since the compiler cannot know for sure what **f2** will point to at run time, it does not let you assign **f2** to **m**. If it allowed this line to compile, there would be a violation

of type safety during execution if `f2` pointed to an `Apple` instead of a `Mango`

The same logic discussed above applies to interfaces as well. If a class implements an interface, then an object of that class **is-a** that interface. For example, if `Fruit` implements an interface named `Edible`, then any fruit will satisfy the is-a test for `Edible`. In other words, a `Fruit` **is-a** `Edible`.

The cast Operator (`()`)

You saw above that the compiler does not let the lines `Mango m = f1;` and `Mango m = f2;` compile because the compiler cannot know for sure that `f1` and `f2` will point to `Mango` objects at run time. Instead of letting an `Apple` get assigned to `m` and thereby violating type safety, it rejects the code altogether. However, the programmer knows what she expects `f2` to refer to at run time (because she wrote the code after all!). Java language allows a programmer to use their knowledge about their program to assure the compiler that a reference will point to an object of the correct type at run time using the cast operator. This assurance convinces the compiler to let them do the assignment. Here is how this works:

```
m = (Mango) f1;  
m = (Mango) f2;
```

By casting `f1` and `f2` to `Mango`, the programmer basically guarantees to the compiler that `f1` and `f2` will point to `Mango` objects at run time. The compiler accepts the guarantee because it knows that `Mango` **is a** `Fruit` and it is possible for a variable of type `Fruit` to refer to a `Mango`. The compiler rejected the code earlier because the variables could point to `Apple` objects as well but with the explicit guarantee given by the programmer that they will point to a `Mango` objects at run time, the compiler accepts the code.

The `java.lang.ClassCastException`

If you observe the above code carefully, you will know that `f1` will not really point to a `Mango` at run time. The line `Fruit f1 = a;` actually makes `f1` point to the same object as `a` and `a` points to an `Apple`. Therefore, `f1` will actually point to an `Apple` and not a `Mango` at run time. So basically, we just

fooled the compiler into accepting the code by making a false guarantee. Well, we did fool the compiler but we can't fool the JVM. The JVM knows what kind of object **f1** actually points to and it will not let the cast of **f1** to **Mango** succeed. The JVM will throw a **ClassCastException** when it executes this line.

The JVM is essentially the second line of defence against violation of Java's type safety. It checks at run time what the compiler is unable to check at compile time. If the JVM sees that a reference variable is being cast to a type that does not satisfy the is-a test for the class of the object to which the variable is referring, it will throw a **ClassCastException**. Thus, the type safety of the program is never compromised.

Fooling the compiler in this case doesn't mean that you can fool the compiler by giving a false guarantee every time. Take a look at this code:

```
Mango m = new Mango();  
Apple a = (Apple) m;
```

The compiler rejects the above code in spite of you guaranteeing that **m** will point to an **Apple** at run time. Well, it turns out that compiler is not completely clueless. It knows that there is absolutely no way **m** can point to an **Apple** because the declared class of **m** is **Mango** and there is no is-a relationship between **Mango** and **Apple**. It knows that anything that is a **Mango** can never be an **Apple** and so it calls your bluff. Thus, for a cast to pass compilation, the cast must at least be plausible!

Casting a reference to an interface [🔗](#)

It is easy for a compiler to determine whether a reference of one class can ever point to an object of another class because a class can only extend one class at most. Therefore, if the class mentioned in the cast is a subclass of the declared class of the reference, the compiler knows that the cast will probably succeed at run time.

It is not so easy with interfaces. Since a class can extend one class and can also implement any number of interfaces at the same time, the compiler cannot rule

out cases where a reference can never point to an instance of a class that implements the interface mentioned in the cast. Here is what I mean:

```
interface Poisonous{ }
class TestClass{
    public static void main(String[] args){
        Fruit f = new Mango(); //ok, because Mango is-a
        Fruit
        Poisonous p = (Poisonous) f; //compiles fine but
        t throws a ClassCastException at run time
    }
}
```

The compiler accepts the casting of **f** to **Poisonous** even though it knows that **Fruit** does not implement **Poisonous**. The reason is that the compiler also knows that even though **Fruit** does not implement **Poisonous**, there could be a subclass of **Fruit** that implements **Poisonous** and since that subclass would be a **Fruit**, **f** could potentially point to an object of a class that implements **Poisonous**. Something like this:

```
class StarFruit extends Fruit implements Poisonous{ }
...
Fruit f = new StarFruit();
Poisonous p = (Poisonous) f; //compiles and runs fine
```

The above reasoning implies that you can cast any type of reference to any interface. That's true except in one case. If the declared class of the variable is **final** and if that class does not implement the interface given in the cast, the compiler knows that this class can never have any subclass and therefore, it knows that there is no way the reference can point to an object of a class that extends this class and also implements the given interface. For example,

`String` is a final class and the following code will not compile for the same reason:

```
String s = new String();  
Poisonous p = (Poisonous) s; //will not compile
```

Downcast vs Upcast

Casting a variable of a type to a subtype (e.g. casting the variable `f` of type `Fruit` to `Apple`) is called "downcasting". It is called downcasting because in a UML diagram a subclass is always drawn below the superclass. Also, a variable of type `Fruit` may refer to any kind of fruit but when you cast it to `Apple`, you have essentially reduced the possibilities for the kind of fruits that the variable could be referring to. In other words, when you cast a variable to a subtype, you are narrowing the type of the object to which this variable is pointing, down to a more specific type. Hence, the term "narrowing". It means the same as downcasting. The opposite of downcasting is "upcasting" or "widening".

As explained above, downcasting always requires a check by the JVM to make sure that the variable is really pointing to an object of type that the programmer has claimed it is pointing to. Upcasting, on the other hand, requires no such check and is, in fact, almost always redundant (I will show you the one situation where it is not redundant soon). Since an `Apple` is-a `Fruit`, you can always assign a variable of type `Apple` to a variable of type `Fruit` without any explicit cast.

12.5.3 When is casting necessary

In the previous section I used the analogy of a remote and a TV to show you that the type of a variable and the type of the actual object referred to by that variable are two different things and may not necessarily be the same.

Let me continue with the same analogy to show you how you can deal with the difference between the type of a variable and type of the object referred to by the

variable. Imagine you have the remote for an old model that has a limited number of buttons and you are using this remote to control a TV from a new model that has a lot of functions. Obviously, you will only be able to use those functions for which there are buttons in the remote. Even though the TV supports many more functions, you cannot use them with the old remote because the old remote has no knowledge of the new functions.

The reference and the object behave the same way. The type of the reference is like the model of the remote and type of the object is like the model of the TV. The following code makes this clear:

```
Object obj = "hello";
int h = obj.hashCode(); //ok because hashCode is defined in Object
```

```
int i = obj.length(); //will not compile
```

```
String str = "hello";
int j = str.length(); //OK
```

In the above code, the declared type of the reference variable `obj` is `Object` and the type of the object to which it refers to at run time is `String`. Since `String` is a subclass of `Object`, `String` is-a `Object` and therefore, it is ok to assign a `String` object to `obj`.

You can say that `String` is a kind of new model of `Object` and it has several new features in addition to all the features of `Object`. However, since the declared type of `obj` is `Object`, the compiler will only let you use the functionality that is supported by `Object` because the compiler does not know that `obj` will actually point to a string at run time.

If you want to use the new features of `String` class using `obj`, you will need to cast it to `String` using the cast operator:

```
Object obj = "hello";
```

```

String str = (String) obj; //cast obj to String

int i = str.length();

int j = ((String) obj).length(); //casting and access
//ing at the same time

```

Thus, the answer to the question when is casting necessary is simple. You need casting when you want to use the features (i.e. instance variables and methods) defined in a subclass using a reference whose declared type is of a superclass.

Remember that casting doesn't change the actual object. The purpose of casting is to provide the compiler with the type information of the actual object to which a variable will be pointing to at run time. Thus, casting just changes the perspective from which the compiler views the object.

Impact of casting on static members ↗

Casting is essentially an aspect of object-oriented programming while "static" is not. Nevertheless, due to the peculiarity of the Java language, it is possible to access static members of a type through a variable of that type. Recall from the "Working with Data types" chapter that static members of a type "shadow" the static members of the same name in the super type. Casting a variable to the super type is how you "unshadow" those members. Here is an example:

```

class Fruit{
    static int count = 5;
    static int getCount(){ return count; }
}
class Apple extends Fruit{
    static int count = 10; //shadows Fruit's count

    static int getCount(){ //shadows Fruit's getCount
}

```

```

        return count;
    }
}
class TestClass{
    public static void main(String[] args){
        Apple a = new Apple();
        System.out.println(a.count); //prints 10;

        System.out.println(a.getCount()); //prints 10;

        System.out.println( ((Fruit) a).count ); //prints 5;
        System.out.println( ((Fruit) a).getCount() ); //prints 5;

        Fruit f = a; //observe that no cast is needed here because we are widening
        System.out.println( f.count ); //prints 5;
        System.out.println( f.getCount() ); //prints 5;

    }
}
```

As promised, the above code shows that an upcast is not always redundant. But I

cannot stress enough that the above code is completely unprofessional. You should never access static members through a variable.

While understanding Casting is important, it is used sparingly in professionally designed code because it is, after all, a means of avoiding the type safety mechanism of the compiler. When you cast a reference to another type, you are basically saying that the program does something that is not evident from the code itself and you are browbeating the compiler into accepting that code. This reflects a bad design. Ideally, you should almost never need to use casting.

12.5.4 The instanceof operator [↳](#)

The instanceof operator is not on the OCP Part 1 exam but it is important to know about this operator to develop a good understanding of polymorphism. In fact, the cast operator, which is on the exam, and the instanceof operator often go together and that is why I am going to talk about it in detail.

Since the actual type of the object which a variable is pointing to is not always evident from the code, Java has the `instanceof` operator for the programmer to check whether the actual object is an instance of a particular reference type (i.e. class, interface, or enum) that the programmer is interested in. It takes two arguments - a reference on the left-hand side and a type name on the right-hand side. It returns a boolean value of `true` if the object pointed to by the variable satisfies the is-a test with the type name given and `false` otherwise. For example, if you are passed a `Fruit` as an argument to a method, here is how you can use `instanceof` to know whether you have actually been given a `Mango` :

```
class Fruit{ }
```

```

class Mango extends Fruit{ }

class Apple extends Fruit{ }

public class Juicer{
    public void crush(Fruit f){
        if(f instanceof Mango

){
            System.out.println("crushing mango...");
        } else {
            System.out.println("crushing some other fruit
...");
        }
    }

    public static void main(String[] args){
        Mango m = new Mango();
        new Juicer().crush(m);
    }
}

```

In the above code, the method `crush` doesn't know the type of the actual object that is referred to by `f` , but using the `instanceof` operator you can check if that object **is-a** `Mango` .

The `instanceof` operator is handy when you want to treat an object of a particular kind differently. For example, if `Mango` had a method named `removeSeed` and if you want to invoke that method before crushing it, you would want to know whether you have actually been given a `Mango` or not before you cast your `Fruit` reference `f` to `Mango` , otherwise, you will get a `ClassCastException` at run time if `f` does not refer to a `Mango` . Here is how this can be done:

```

class Mango extends Fruit{
    public void removeSeed(){ }
}

```

```

class Juicer{
    public void crush(Fruit f){

        if(f instanceof Mango){
            Mango m = (Mango) f;
            m.removeSeed();
        }
        System.out.println("crushing fruit...");  

    }
}
...
}

```

Remember that you can't invoke `removeSeed` on `f` because the type of `f` is `Fruit` and `Fruit` doesn't have `removeSeed` method. `Mango` does.

Therefore, as discussed before, you must cast `f` to `Mango` before you can invoke a method that is specific to a `Mango`.

There are a couple of things that you should understand clearly about `instanceof`:

1. The `instanceof` operator cannot tell you the exact type of the object being pointed to by a variable. It can only tell you whether that object is-a something. For example, if you do `f instanceof Fruit`, it will return `true` if the object referred to by `f` is-a `Fruit`, which means it will return `true` even if `f` points to a `Mango` because a `Mango` is-a `Fruit`. In the case of interfaces, it will return `true` if the class of the object pointed to by the reference implements the given interface (directly or indirectly).
2. The compiler will let you use `instanceof` operator only if it is possible for the variable to refer to an object of the type given on the right-hand side. For example, `f instanceof Mango` is valid because the compiler knows that the declared type of `f` is `Fruit` and since `Mango` is a `Fruit`, it is possible for `f` to point to a `Mango`. But `f instanceof String` will not compile because the compiler knows that there is no way `f` can ever point to a `String`. The `instanceof` operator behaves the same

way as the cast operator in this respect.

Just like the cast operator, the `instanceof` operator is also used only sparingly in professionally written code. Usage of `instanceof` reflects a bad design and if you feel the need to use `instanceof` operator in your code too often, you should think about redesigning your application.

12.5.5 Invoking overridden methods [↑](#)

If I haven't hammered it in enough yet, let me say this again - polymorphism is all about the ability to replace one object with another without the need to recompile existing code as long as the objects stick to an agreed upon contract. For example, if a method requires an object of a class as an argument, then it should work well with an object of its subclass. If a method expects an interface as an argument, the class of the object that you pass to it shouldn't matter as long as that class implements the interface. This flexibility makes it easier, and thereby cheaper, to develop and maintain an application.

In technical terms, you must remember that polymorphism works only because of **dynamic binding** of methods calls. When you invoke an instance method using a reference variable, it is not the compiler but the JVM that determines which code to execute based on the class of the actual object referenced by the variable.

Some languages let the programmer decide whether they want to let the compiler bind a method call to the version provided by the declared class of the variable or to let the JVM bind the call at run time based on the class of the object referred to by the variable. In such languages, methods that are not bound by the compiler are called "**virtual methods**" . Java does not give the programmer the ability to customize this behavior. In Java, calls to **non-private** and **non-final instance** methods are bound **dynamically** by the JVM and are therefore, always "**virtual**" . Everything else is bound **statically** at compile time by the compiler.

To take advantage of polymorphism, it is advisable to use interfaces and non-

final classes as method parameter types. For example, the interest calculator example that I showed earlier could be redesigned as follows:

```
interface InterestCalculator{

    //interface methods are public by default

    double computeInterest(double p, double r, double
t);
}

class Account{
    double balance, rate;

    double getInterest(InterestCalculator ic, double t
ime){
        return ic.computeInterest(balance, rate, time)
;
    }
}

class SimpleInterestCalculator implements InterestCalc
ulator{

    //must be public because an overriding method must
not reduce accessibility

    public double computeInterest(double principle, dou
ble yrs, double rate) {
        return principle*yrs*rate;
    }
}

class CompoundInterestCalculator implements InterestCa
lculator{
    public double computeInterest(double principle, dou
```

```
ble yrs, double rate) {  
    return principle*Math.pow(1 + rate, yrs) - prin  
ciple;  
}  
}
```

Observe that the `getInterest` method now takes an `InterestCalculator` as a parameter. This makes it very easy for any other class to compute any kind of interest on an account object. As of now, there are two classes that compute interest - `SimpleInterestCalculator` and `CompoundInterestCalculator`, but in future if you want to compute interest with different compounding, all you have to do is create a new class that implements `InterestCalculator` interface and pass an object of this class to the `getInterest` method. The Account class will not know the difference. The JVM will bind the call to `computeInterest` to the code provided by your new class automatically.

Dynamic binding of method calls may cause unexpected results if you are not careful. Consider the following code:

```
class Account{  
    double balance = 0.0;  
  
    Account(double balance){  
        this.balance = balance;  
        this.printBalance();  
    }  
  
    void printBalance(){  
        System.out.println(balance);  
    }  
}  
  
class DummyAccount extends Account{  
  
    DummyAccount(double b ){  
        super(b);  
    }  
}
```

```

public void printBalance(){
    System.out.println("No balance in dummy account")
;
}
}

public class TestClass{
    public static void main(String[] args){
        Account a = new DummyAccount(100.0);
    }
}

```

Can you guess what it prints?

You may expect the call to `this.printBalance()`; in `Account`'s constructor to be bound to `Account`'s `printBalance` method but observe that this method is overridden by `DummyAccount`. Since the class of the actual object is `DummyAccount`, the JVM will bind the call to `DummyAccount`'s `printBalance` instead of `Account`'s `printBalance`. Therefore, it will print "No balance in DummyAccount".

This example shows that you need to be very careful about calling non-private methods from a constructor. And by careful, I mean, don't do it :) A constructor is meant to initialize the object's state variables to their appropriate values but if you invoke a non-private method from a constructor, a subclass can easily mess with your constructor's logic by overriding that method.

12.5.6 Impact of polymorphism on == and equals method

You know that when used on references the `==` operator checks whether the two operands point to the same object in memory or not. So, can you guess what the following code will print?

```

String s = "hello";
Integer n = 10;
System.out.println(n == s);

```

`false`, you say? Well, the code won't even compile. The compiler applies the same logic that it applies to the cast and the instanceof operators to check whether it is even possible for the two reference variables to point to the same object. It rejects the comparison if it is not. In the above code, there is no way `s` and `n` can point to the same object because `s` and `n` are variables of two different unrelated types. Thus, the compiler knows that this comparison is pointless and is most probably a mistake by the programmer. Here is the same code but with a small change:

```
Object s = "hello";
Integer n = 10;
System.out.println(n == s);
```

The above code indeed prints `false`. The compiler cannot reject the comparison now because the type of `s` is `Object` and therefore, it is possible for `s` to point to an `Integer` object (because `Integer` is-a `Object`).

The `equals` method, on the other hand, behaves differently. Remember that `equals` is defined in the `Object` class and its signature is `equals(Object)`. The type of the input parameter is `Object` and therefore, it must accept a reference of any type. Thus, the compiler has no option but to accept even the illogical invocations of the `equals` method such as `"1234".equals(n);`. It is for the same reason that when a class overrides the `equals` method, the first line of code in the method is usually an instanceof check:

```
class X{
    int val;
    public boolean equals(Object x){
        if( !(x instanceof X) ) return false;

        //now compare the values of instance fields of this
        //and x and return true/false accordingly

        return this.val == ((X) x).val
    }
}
```

```
; } }
```

If you remember the rules of overriding, an overriding method is not allowed to change the type of the input parameter to a narrower type. Thus, if a class tries to override the `equals` method but changes the type of the input parameter from `Object` to a more specific type, it will not be a valid "override". It will be a valid "overload" though. For example, the following code will compile fine, but the `equals` method does not override the `equals` method that `X` inherits from `Object`:

```
class X{
    int val;
    public boolean equals(X x){ //does not override but overloads the equals method
        return this.val == x.val;
    }
    public static void main(String[] args){
        X x1 = new X(); x1.val = 1;
        X x2 = new X(); x2.val = 1;
        System.out.println(x1.equals(x2)); //prints true
    }
}
```

On the face of it, the `equals` method written above doesn't seem to make much of a difference when you try to compare two `X` objects. But let's change the code inside the `main` method as follows and see what happens:

```
X x1 = new X(); x1.val = 1;
Object x2 = new X(); ((X) x2).val = 1;
System.out.println(x1.equals(x2)); //what will it print?
```

If you have understood the rules of method selection that we discussed in the "Working with Methods and Encapsulation" chapter, you should be able to figure out that the above code will print `false`. The compiler sees two versions of the equals method in class `X` to choose from when it tries to bind the `x1.equals(x2)` method call - the version that class `X` inherits from `java.lang.Object`, which takes `Object` as an argument and the version that class `X` implements itself, which takes `X` as an argument. Since the declared type of the variable `x2` is `Object`, the compiler binds the call to the `Object` version instead of the `X` version. The `Object` version returns `false` because `x1` and `x2` are pointing to two different objects.

12.6 Distinguish overloading, overriding, and hiding

12.6.1 Overriding and Hiding [🔗](#)

You have seen how a class can inherit features by extending another class. These features include static and instance fields as well as static and instance methods. In the previous section, I explained how inheriting features is usually a good thing because it allows a class to get functionality without writing any code. But it could pose a problem if the subclass were not able to provide suitable behavior to a method that it inherited.

For example, what if there is a `SpecialInterestCalculator` that extends `InterestCalculator` and inherits a `computeInterest` method, but it wants to change how the interest is computed by this method? Or what if it wants to define a variable `interestRate`, but that variable is already defined in `InterestCalculator`?

Java has specific rules about what features can be tweaked and how they can be tweaked by a subclass. These rules are categorized into two categories:

Overriding and Hiding. The rules of overriding are about **polymorphism** and therefore, only apply to **instance methods** and the rules of hiding apply to everything else, i.e., **static methods** as well as **static and instance variables**.

Remember that in both the cases, a member of a class has to be inherited in the subclass first. Since the private members of a class are not inherited by a subclass, the concepts of overriding and hiding are not applicable to them. Similarly, constructors of a class are not inherited either and are, therefore, out of the purview of overriding and hiding.

Overriding ↗

A class is allowed to completely replace the behavior of an instance method that it inherited by providing its own implementation of that method. What this means is that the behavior provided by the subclass is what will be exhibited by any object of the subclass instead of the behavior provided by the super class. For example:

```
class InterestCalculator{
    public double computeInterest(double principle, int
yrs, double rate){
        return principle*yrs*rate;
    }
}

class CompoundInterestCalculator extends InterestCalculator{

    public double computeInterest(double principle, int
yrs, double rate){
        return principle*Math.pow(1 + rate, yrs) - pr
inciple;//don't worry about Math.pow()!

    }
}
```

In the above code, **CompoundInterestCalculator** has replaced the implementation of **computeInterest** method provided by its super class with its own implementation. If you call the **computeInterest** method on a **CompoundInterestCalculator** object,

`CompoundInterestCalculator`'s version of the method will be called.
The following code proves it.

```
class TestClass{
    public static void main(String[] args){
        InterestCalculator ic = new InterestCalculator();
        double interest = ic.computeInterest(100, 2, 0.1);
        System.out.println(interest); //prints 20.0

        CompoundInterestCalculator cic = new CompoundInterestCalculator();
        interest = cic.computeInterest(100, 2, 0.1);
        System.out.println(interest); //prints 21.0
    }
}
```

Note that I am using the word replace to highlight the fact that it is not possible for any other class to see the behavior of `computeInterest` method as implemented by `InterestCalculator` class in a `CompoundInterestCalculator` object because the behavior of the super class has been replaced by the behavior provided by the subclass. Technically, we say that `CompoundInterestCalculator` has "**overridden**" `computeInterest` method of `InterestCalculator`. Let me make a small change to the above code to make this point clear:

```
class TestClass{
    public static void main(String[] args){
        InterestCalculator ic = new CompoundInterestCalculator();
```

```

        double interest = ic.computeInterest(100, 2, 0
.1);
        System.out.println(interest); //prints 21.0

    }

}

```

In the above code, the declared type of the variable `ic` is `InterestCalculator` but the actual object that it points to is of type `CompoundInterestCalculator`. Therefore, when you call `computeInterest`, `CompoundInterestCalculator`'s version of this method is executed instead of `InterestCalculator`'s version.

The point to understand here is that in the case of instance methods, it is always the method implemented by the class of the object that is invoked.

Hiding

Hiding is a less drastic version of overriding. Like overriding, hiding lets a class define its own version of the features implemented by its superclass, but unlike overriding, hiding does not completely replace them with the subclass' version. Thus, the subclass now has two versions of the same features and any unrelated class can access both the versions. The following code illustrates this point:

```

class InterestCalculator{
    public int yrs = 10;
    public static double rate = 0.1;
    public static String getClassName(){
        return "InterestCalculator";
    }
}

class CompoundInterestCalculator extends InterestCalculator{
    public int yrs = 20;

```

```
    public static double rate = 0.2;
    public static String getClassName(){
        return "CompoundInterestCalculator";
    }
}
```

In the above code, `CompoundInterestCalculator` inherits the instance variable `yrs`, the static variable `rate` and the static method `getClassName` from `InterestCalculator`. At the same time, `CompoundInterestCalculator` defines all of these on its own as well. So, now, `CompoundInterestCalculator` has two `yrs` variables, two `rate` variables, and two versions of the `getClassName` method. Both the versions can be accessed by an unrelated class as shown below:

```
class TestClass{
    public static void main(String[] args){
        CompoundInterestCalculator cic = new CompoundInterestCalculator();

        System.out.println(cic.yrs); //prints 20

        System.out.println( ((InterestCalculator) cic).yrs); //prints 10

        System.out.println(cic.rate); //prints 0.2

        System.out.println( ((InterestCalculator) cic).rate); //prints 0.1

        System.out.println(cic.getClassName()); //prints CompoundInterestCalculator
    }
}
```

```

        System.out.println( ((InterestCalculator) cic).
getClassName()); //prints InterestCalculator

    }

}

```

Observe that I used a special syntax to access the version provided by the superclass. This special syntax is called a "cast". The declared type of the variable `cic` is `CompoundInterestCalculator` but I cast it to `InterestCalculator` to go behind `CompoundInterestCalculator` and access the versions provided by `InterestCalculator`. The cast tells the compiler to treat a variable as if its declared type is the type mentioned in the cast.

I will discuss the rules of casting later but the point to understand here is that static methods, static variables, and instance variables are accessed as per the declared type of the variable through which they are accessed and not according to the actual type of the object to which the variable refers.

Contrast this with overriding, where type of the variable makes no difference to the version of the instance method that is invoked.

12.7 Exercise

1. You are developing an application that allows a user to compare automobiles. Use abstract classes, classes, and interfaces to model `Car` , `Truck` , `Vehicle` , and `Drivable` entities. Declare and define a method named `drive()` in appropriate places.
2. Every vehicle must have a make and model. What can you do to ensure that a method named `getMakeAndModel()` can be invoked on every vehicle.
3. You need to be able to get the Vehicle Identification Number (VIN) of a vehicle by calling `getVIN()` on any vehicle. Furthermore, you don't want any subclass to change the behavior of the `getVIN` method. Where and how will you code the `getVIN` method?

4. Create an interface named **Drivable** with a default method **start()** .
Invoke **start()** on instances of classes that implement **Drivable** .
Override **start()** so that it prints a different message in each class.
5. Ensure that every vehicle is created with a VIN.
6. Create a class named **ToyCar** that extends **Car** but doesn't require any argument while creation.
7. You have a list of features such as height, width, length, power, and boot capacity, on which you want to compare any two vehicles. New feature names will be added to this list in future. Create a **getFeature(String featureName)** method such that it will return "N.A" for any feature that is not supported by a particular vehicle.
8. Create an interface named **VehicleHelper** with a static method **register(Vehicle v)** that prints the VIN of the vehicle. Ensure that VehicleHelper's register method is invoked whenever an instance of a vehicle is created.

Chapter 13 Programming Abstractly Through Interfaces

- Create and implement interfaces
- Distinguish class inheritance from interface inheritance including abstract classes
- Declare and use List and ArrayList instances

13.1 Create and implement interfaces

13.1.1 Using interfaces [↳](#)

An **interface** is used to describe behavior as captured by a group of methods. It doesn't tell you anything about the object behind the behavior except that the methods declared in the interface can be invoked on that object. This group of methods therefore, is a way to interact with the object.

An interface is defined using the keyword "**interface**" . For example:

```
interface Movable{  
    void move(int x);  
}
```

From a purely OOP perspective, an interface should not contain any implementation. It should only contain method declarations. However, Java has always permitted interfaces to contain static fields. Furthermore, from Java 8 , Java has permitted an interface to contain **default methods** and **static methods** and Java 9 has allowed an interface to contain **private methods** (static as well as non-static) as well.

There are several rules about interfaces that you must learn by heart. So, let me

go over them one by one.

Everything is public in an interface by default [↳](#)

Everything declared inside an interface is implicitly **public** except methods that are explicitly declared **private**. This means that members of an interface are always public irrespective of whether you define them as public or not. In fact, you are prohibited from defining them as private or protected. The following example illustrates this point:

```
interface Movable{
    void move1(int x); //OK, move1 is implicitly public

    public void move2(int x); //OK, move2 is explicitly
    public

    private void move3(int x); //NOT OK, will not compil
    e.

    protected void move4(int x); //NOT OK, will not compil
    e.

    int VALUE = 10; //OK, VALUE is implicitly public

    private int PVT_VALUE = 10; //NOT OK, fields cannot
    be protected or private

    private void pvtMethod(){ }; //OK, methods can be pr
    ivate (but not protected)
```

}

This makes sense because the whole purpose of an interface is to put out a way to interact with an object in front of the world. Remember that an interface is not about how an object is implemented. It is about how an object can be interacted with. In that sense, an interface is a **contract** between an object and the rest of the application components and since everything in a contract must be public, everything in an interface must be public.

What if you want to disclose members of an interface only to the members of the same package? Well, make the interface "**default**". As discussed earlier, default access allows something to be visible only to members of the same package. If the interface itself is not visible outside the package, its members certainly won't be.

Private methods do not really fit conceptually in an interface but the need for private methods was felt after default methods were introduced in Java 8. If a method gets too big or if there are multiple methods with a lot of common code, there was no way to refactor them into smaller methods without exposing all of them to the world because everything in an interface had to be public. Java 9 fixes this problem by allowing **private methods** in an interface.

An interface is always abstract [🔗](#)

As mentioned above, an interface is merely a contractual description of some behavior. It is not an entity that can actually fulfill that contract. For example, you may describe the behavior of movement using an interface named `Movable` but you need an object such as an animal or vehicle to exhibit that behavior. Therefore, it does not make sense to instantiate an interface. For this reason, an interface is **implicitly abstract**. Although legally valid, it would be redundant to declare an interface as abstract.

Variable definitions in an interface [🔗](#)

All variables defined in an interface are implicitly **public**, **static**, and **final**. The following example illustrates this point:

```
interface Movable{
    int UNIT1 = 1;
    static int UNIT2 = 1;
    static final int UNIT3 = 1;
    public static final int UNIT4 = 1;
}
```

All four of the variables declarations above are valid. All of the variables are **public** , **static** , and **final** even though the first three have not been declared as such. This implies that you cannot have instance variables in an interface.

Remember that instance variables are meant to store **state** , which means they are really a part of **implementation** . Therefore, instance variables have no place in an interface. To be honest, it is not a good idea to have static variables either in an interface but for reasons best known to the language designers, Java allows static constants in an interface. Many people use this feature to define "**global constants**" but it is simply a bad design choice. Avoid them in your code.

Methods in an interface [↳](#)

An interface can have four kinds of methods :

1. **abstract methods** - You have seen abstract methods in the previous section about abstract classes. They contain just the declaration and no body. It is the same thing here except that the keyword **abstract** is optional. For example:

```
interface Movable{
    void move1(int x); //implicitly abstract
    abstract void move2(int x); //explicitly abstract
```

```
}
```

Both of the methods declared above are abstract.

2. **default methods** - Default methods are a way for an interface to include a default implementation for a method. They are defined using the keyword **default** as follows:

```
interface Movable{
    default

    void move(int x){
        System.out.println("Dummy implementation. Mov
ing by "+x+" points");
    }
}
```

Observe that default methods are just the opposite of abstract methods - abstract methods cannot have an implementation while the whole purpose of default methods is to provide an implementation. Therefore, a method cannot be default as well as abstract at the same time. Thus, the following code will not compile:

```
interface Movable{

    //must be marked default or private because it
    has a body

    void move1(int x){
        System.out.println("Dummy implementation. Mov
ing by "+x+" points");
    }

    void move2(int x); //ok, no body
}
```

```
}
```

3. **static methods** - As the name implies, static methods belong to the interface itself and not to the object implementing that interface. They are defined using the keyword **static** as follows:

```
interface Movable{
    static

    void sayHello(){
        System.out.println("Hello!");
    }
}
```

Since **sayHello** is a static method, you don't need an instance of any object to invoke it. You can invoke it directly on the interface just like you invoke static methods on a class. For example:

```
public class TestClass{
    public static void main(String[] args){
        Movable.sayHello();
    }
}
```

4. **private methods** - Private methods (static as well as no-static) have been allowed since Java 9 and, as explained above, they are helpful when a default static methods gets too big and needs to be refactored into smaller internal methods without exposing the internal methods to the outside world. Here is an example:

```
interface Movable{
    private
```

```
void moveInternal(){ //don't want to make it accessible to others
    System.out.println("in moveInternal");
}
default void move(int n){
    while(n-- > 0) moveInternal();
}
```

Note that, since private methods are not visible outside the interface, they cannot be marked default.

Since methods of an interface are meant to be implemented by the classes that implement the interface, methods of an interface cannot be declared final. Observe that this is opposite to the rule that is applied to variables. Variables of an interface are always final.

Although not important for the exam, it is possible to define classes, interfaces, and enums within an interface. They cannot be anything other than public.

Marker interface [↳](#)

You may encounter an interface that does not contain anything at all. Such interfaces are called "marker interfaces". For example:

```
interface SpecialThing{}
```

In the above code, **SpecialThing** is a marker interface.

The purpose of a marker interface is to tag a class with an extra piece of information about that class itself. For example, if any class implements **SpecialThing**, it implies that that class is a **SpecialThing**. This information, also called "metadata", could be used by some code or some tool

that treats all **SpecialThing**s in a certain way.

The most common marker interface used in Java is **java.io.Serializable** interface. It signifies to the JVM that objects of classes implementing this interface can be serialized and deserialized.

You don't need to know about the term marker interface or the Serializable interface for the Part 1 exam but it is often asked about in technical interviews.

13.1.2 Implementing an interface [↳](#)

A class can implement any number of interfaces by specifying their names in its **implements** clause. For example, in the following code the **Price** class implements two interfaces:

```
interface Movable{
    void move();
}

interface Readable{
    void read();
}

class Price implements Movable, Readable{
    public void move() { System.out.println("Moving...") }
    public void read() { System.out.println("Reading...") }
}
```

Of course, once a class declares that it implements an interface, it must then have the implementation for all of the abstract methods declared in that interface. It could either implement the methods itself or inherit them from its ancestor class. If an interface provides a default implementation for a method in the form of a default method , the implementing class does not necessarily have to provide implementation for that method.

If the class does not have implementation for even one of the abstract methods declared in the interface that it says it implements, the class must be declared abstract . Otherwise, the compiler will refuse to compile the class.

So, for example, if the `StockPrice` class does not provide an implementation for the `read` method declared in `Readable` , it would have to be declared abstract like so:

abstract

```
class StockPrice implements Movable, Readable{  
    public void move() { System.out.println("Moving...")  
; }  
}
```

A class is allowed to extend another class as well as implement any number of interfaces at the same time. In this case, the class declaration will have an **extends** clause as well as an **implements** clause. For example:

```
interface Printable{  
    void print();  
}  
public class StockPrice extends  
  
    Price implements  
  
    Printable{  
        public void print(){  
            System.out.println("Printing StockPrice...");  
        }  
    }
```

In the above code, `StockPrice` implements `Printable` explicitly and since it extends `Price` , it implements `Movable` and `Readable` implicitly. Furthermore, `StockPrice` inherits the move and read methods from `Price` and therefore, it does not need to define them again.

Note that the order of extends clause and implements clause is important. The extends clause must appear before the implements clause.

Inheritance of static methods of an interface [↳](#)

Unlike the static methods of a class, the static methods of an interface cannot be inherited. This difference is illustrated by the following code:

```
class Price{
    static void m(){
        System.out.println("In Price.m()");
    }
}
interface Printable{
    static void p(){
        System.out.println("In Printable.p()");
    }
}
class StockPrice extends Price implements Printable{
}
class TestClass{
    public static void main(String[] args){
        StockPrice.m(); //works fine
        StockPrice.p(); //will not compile
    }
}
```

In the above code, since **StockPrice** extends **Price**, **StockPrice** inherits the static method **m()** defined in **Price**. However, even though **StockPrice** implements **Printable**, it does not inherit the static method **p()** defined in **Printable**. That is why the call **StockPrice.p()** fails to compile.

Inheritance of multiple versions of a default method [↳](#)

Since it is possible for a class to implement multiple interfaces, it is possible for a class to inherit multiple implementations of a default method from more than one interface. For example:

```
interface Task{
    public default void doIt(){
        System.out.println("Doing Task");
    }
}

interface Activity{
    public default void doIt(){
        System.out.println("Doing Activity");
    }
}

//will not compile

class Process implements Task, Activity{}
```

In the above code, **Process** implements two interfaces and since each of them contains a default method named **doIt** with the same signature, **Process** now has two implementations for the same method. This is a problem because when you call **doIt** method on a **Process** object, the JVM will not be able to determine which implementation of **doIt** to invoke.

Java resolves this problem by forcing the class to provide an implementation of the method of its own to remove the ambiguity in invocation.

```
class Process implements Task, Activity{
    public void doIt(){
        System.out.println("Doing Process");
    }
}
```

The JVM now has only one implementation of **doIt** method to invoke and so, the above code compiles fine.

Let us look at a case that is even more interesting. Can you tell what happens when you try to compile the following code:

```
interface Task{
    int SIZE = 10;
    default void doIt(){
        System.out.println("Doing Task");
    }
}

interface Activity{
    long SIZE = 20;
    void doIt();
}

class Process implements Task, Activity{
    public static void main(String[] args){
        Process p = new Process();
        p.doIt();
    }
}
```

Observe that **doIt** method is declared by both the interfaces but only one of them provides an implementation. Thus, **Process** inherits only one implementation for **doIt** method. There is no ambiguity for the JVM in determining which implementation of **doIt** to invoke. Therefore, it should compile, right? Wrong!

Java designers thought that even though there is only one implementation for the **doIt** method, there is no guarantee that this implementation is appropriate for the **doIt** method declared by another interface. The contract of the other

interface could possibly be different from the one that provides the implementation for the `doIt` method. Therefore, it is better for the class to explicitly provide an implementation of the `doIt` method.

Note that this issue does not arise when there is no default implementation available at all because in that case the class will have to provide its own implementation anyway.

Inheritance of multiple versions of a variable [↳](#)

The situation with the `SIZE` variables in the above code is a bit special. Fields of an interface are inherited by a sub class and therefore `Process` does get two versions of `SIZE` variable. Java allows a class to inherit multiple fields with the same name as long as you don't try to use those fields ambiguously. This means that you will get a compilation error only if you try to use `SIZE` within `Process` directly without specifying which `SIZE` field are you trying to refer to. This is illustrated below:

```
interface Activity{
    long SIZE = 20;
    //void doIt(); //let's get rid of it to avoid the method ambiguity issue for now
}

class Process implements Task, Activity{

    public static void main(String[] args){
        System.out.println(SIZE); //will not compile
    }
}
```

The above code will fail to compile with an error message, "reference to `SIZE` is ambiguous" because the compiler is not able to figure out which

SIZE you are trying to use. But the following code will compile fine because the compiler has no confusion about the **SIZE** that are you referring to here:

```
class Process implements Task, Activity{  
    public static void main(String[] args){  
        System.out.println(Activity.SIZE);  
        System.out.println(Task.SIZE);  
    }  
}
```

13.1.3 Extending an interface

It is possible for an interface to **extend** any number of interfaces. For example:

```
interface Readable{  
    int SIZE = 0;  
    void read();  
}  
interface Writable{  
    void write();  
}  
  
interface ReadWriteable extends Readable, Writable{  
    //inherits SIZE and read() from Readable  
  
    //inherits write() from Writable  
  
    void delete();  
}
```

Remember that a class cannot extend an interface, it can only implement an interface. Whereas, an interface cannot implement any interface it can only extend an interface. Thus, the following definitions of Writer will not compile:

```
interface Writer implements Writable{ //interface cann  
ot "implement" any interface  
  
    public void write(){ }  
}  
  
class Writer extends Writable{ //class cannot "extend"  
any interface  
  
    public void write(){ }  
}
```

The extending interface inherits all the members **except static methods** of each of the other extended interfaces.

Inheriting multiple versions of a default method [↳](#)

It is possible for an interface to inherit a field or an abstract method with the same signature from two of its super interface. But inheriting multiple default methods or one default and one or more abstract methods with the same signature has the same problem that you saw earlier in a class that implements multiple interfaces. Java does not allow it. Here is an example:

```
interface Readable{  
    int SIZE = 10;  
  
    void read();  
  
    static void staticMethod(){  
        System.out.println("In Readable.staticMethod");  
    };
```

```
    default void defaultMethod(){
        System.out.println("In Readable.defaultMethod");

    };
}

interface Writable{
    int SIZE = 20;

    void write();

    static void staticMethod(){
        System.out.println("In Writable.staticMethod");
    };

/* commenting the following two methods out

    default void defaultMethod(){
        System.out.println("In Writable.defaultMethod");

    };

    void defaultMethod();
*/

}

interface ReadWriteable extends Readable, Writable{
    //inherits SIZE, read(), and defaultMethod() from Readable

    //inherits SIZE and write() from Writable
```

```
}
```

The above code compiles fine. But if you uncomment either of the `defaultMethod`s in `Writable`, `Readable` will fail to compile because it would be inheriting two different implementations (or one implementation and one declaration) of `defaultMethod`. The solution is the same as what we did with the class. `Readable` must provide its own implementation of the `defaultMethod` to resolve the ambiguity:

```
interface ReadWritable extends Readable, Writable{
    default
        void defaultMethod(){
            System.out.println("In ReadWritable.defaultMethod");
        };
}
```

Observe that `staticMethod` is also defined in both the super interfaces but it does not cause any problem because static methods of an interface are never inherited, which means `Readable` does not get even a single implementation of `staticMethod` from either of its superinterfaces, let alone two, which would cause ambiguity.

Inheriting multiple variables with same name [↳](#)

The situation with the `SIZE` variable (remember that it is implicitly static and final) is, again, the same as what you saw in a class that inherits multiple versions of a field from multiple interfaces. Static fields of an interface are inherited by a sub interface and therefore `Readable` does get two versions of `SIZE` variable. Java allows an interface to inherit multiple fields with the same name as long as you don't try to use those fields ambiguously. For example, the following code will not compile :

```
interface ReadWritable extends Readable, Writable{  
    int NEWSIZE = SIZE; //will not compile because SIZE  
    is being used ambiguously  
}
```

The above code will fail to compile with an error message, "reference to SIZE is ambiguous" because the compiler is not able to figure out which SIZE are you trying to use. But the following code will compile fine because the compiler has no confusion about the SIZE that you are referring to here:

```
interface ReadWritable extends Readable, Writable{  
    int NEWSIZE = Readable.SIZE; //fine  
}
```

13.1.4 Instantiating abstract classes and interfaces [↳](#)

Abstract classes and interfaces are abstract. Objects of their kind do not exist, which is why they are called abstract in the first place. Therefore, they cannot be instantiated, period.

I have seen many students getting confused when they see code that looks like it is instantiating an abstract class. For example:

```
abstract class Animal{  
    public static void main(String[] args){  
        Animal a = new Animal(){ };  
    }
```

```
}
```

The above code seems to be instantiating **Animal** even though **Animal** is abstract! The following is an example that seems to instantiate an interface:

```
interface Dummy{  
}  
public class TestClass{  
    public static void main(String[] args){  
        Dummy d = new Dummy(){ };  
    }  
}
```

Well, first thing that you should observe is the presence of `{ }` between `new Animal()` and `;`. This is not the syntax for instantiating a class. For instantiating **Animal**, you would have to write `new Animal();` and if you try to do that you will get a compilation error that says, "**Animal is abstract; cannot be instantiated**".

In fact, the above code uses the syntax for declaring as well as instantiating a concrete inner class that extends **Animal** without giving this class a name. Since no name is given to this class, it is called an **anonymous class**. When the compiler sees this code, it actually creates a class, gives this class a weird looking name, and generates a separate a class file for this class. It is this class that is instantiated at run time. Since **Animal** class doesn't have any abstract method, this anonymous class doesn't need to implement any method. Had **Animal** had an abstract method, you would have to implement this method inside the curly brackets.

The same thing is happening in the case of the interface example above. The code is defining and instantiating in a single statement an anonymous class that implements the **Dummy** interface.

Since the topic of inner classes (including anonymous inner class) is not on the OCP Java 11 Part 1 exam, I will not discuss this any further in this book.

13.2 Distinguish class inheritance from interface inheritance including abstract classes

13.2.1 Difference between Interface and Abstract Class [↳](#)

"What is the difference between an **interface** and an **abstract class** " is usually the first question that is asked in a Java "tech-check". While being a simple ice breaker, it also an excellent question to judge a candidate's understanding of OOP.

Candidates usually start parroting the technical differences such as an interface cannot have method implementations (which it can, since Java 8), while abstract class can. An interface cannot have static methods (which it can, since Java 8) or instance fields while an abstract class can and so on.

All that is correct, but is not really impressive. The fundamental difference between an interface and an abstract class is that an interface defines just the behavior. An interface tells you nothing about the actual object other than how it behaves. An abstract class, on the other hand, defines an object, which in turn, drives the behavior.

If you understand this concept everything else about them will fall in place. For example, "movement" is a behavior that is displayed by various kinds of objects such as a Car, a Cat, or a StockPrice. These objects have no commonality except that they move. Saying it the other way round, if you get an object that "moves", you don't get any idea about what kind of an object are you going to deal with. It could be a Car, a Cat, or a StockPrice. If you were to capture this behavior in Java, you would use an interface named **Movable** with a method named **move()**.

On the other hand, if you talk about Automobile, a picture of an object starts forming in your head immediately. You can sense that an Automobile would be something that would have an engine, would have wheels, and would move. You intuitively know that a StockPrice or a Cat cannot be an Automobile even though they both do move. An abstract class is meant exactly for this purpose, when, once you identify a conceptual object, you do not need to worry about its behavior. The behavior kind of flows automatically. If you create an abstract

class named Automobile, you are almost certain that it would have methods such a move, turn, accelerate, or brake. It would have fields for capturing inner details such Engine, Wheels, and Gears. You get all that just by saying the word Automobile.

From the above discussion, it should be clear that interfaces and abstract classes are not interchangeable. Even though an abstract class with no non-abstract method looks functionally similar to an interface, both are fundamentally different. If you are capturing a behavior, you must use an interface. If you are capturing a conceptual object, you must use an abstract class.

13.3 Declare and use List and ArrayList instances

13.3.1 Introduction to Collections and Generics

Processing multiple objects of the same kind in the same way is often a requirement in applications. Printing the names of all the Employees in a list of Employees, computing interest for a list of Accounts, or something as simple as computing the average of a list of numbers, require you do deal with a list of objects instead of one single object. You have already seen a data structure that is capable of holding multiple objects of the same kind - array. An array lets you hold references to multiple objects of a type and pass them around as a bunch. However, arrays have a couple of limitations. First, an array cannot grow or shrink in length after it is created. If you create an array of 10 Employees and later find out you have 11 Employees to hold, you will need to create a new array with 11 slots. You can't just add one more slot in an existing array. Second, inserting a value in the middle of an array is a bit difficult. If you want to insert an element just before the last element, you will have to first shift the last element to the next position and then put the value. Imagine if you want to insert an element in the middle of the list!

Although both of the limitations can be overcome by writing some extra code, these requirements are so common that writing the same code over and over again is just not a good idea. `java.util.ArrayList` is a class that incorporates these, and several other, features out of the box. The following is an

example that shows how easy it is to manage a list of objects using an ArrayList:

```
import java.util.ArrayList;
public class TestClass{
    public static void main(String[] args){
        ArrayList al = new ArrayList();

        al.add("alice");//[alice]

        al.add("bob");//[alice, bob]

        al.add("charlie");//[alice, bob, charlie]

        al.add(2, "david");//[alice, bob, david, charlie]

        al.remove(0);//[bob, david, charlie]

        for(Object o : al){ //process objects in the li
st

            String name = (String) o;
            System.out.println(name+" "+name.length());
        }

        //dump contents of the list

        System.out.println("All names: "+al);
    }
}
```

```
}
```

You will get a few warning messages when you compile the above program. Ignore them for now. It prints the following output when run:

```
bob 3
david 5
charlie 7
All names: [bob, david, charlie]
```

The above program illustrates the basics of using an `ArrayList`, i.e., adding objects, removing, iterating over them, and printing the contents of the `ArrayList`. I suggest you write a program that does the same thing using an array of strings instead of an `ArrayList`. This will give you an appreciation for the value that `ArrayList` adds over a raw array.

I showed you the above code just to give you a glimpse of one of the readymade classes in the Java library. There is a lot more that you can do with it but before we get to that, you need to understand the bigger picture, the grand scheme of things, into which classes such as `ArrayList` fit.

Collections and Collections API [🔗](#)

`ArrayList` belongs to a category of classes called "**collections**". A "**collection**" is nothing but a group of objects held up together in some form. You can visualize a collection as a bag in which you put several things. Just like you use a bag to take various grocery items from a store to your home, you put the objects in a collection so that you can pass them around to other objects and methods. A bag knows nothing about the items themselves. It merely holds the items. You can add items to it and take items out of it. It doesn't care about the order in which you put the items or whether there are any duplicates or how they are stored inside the bag. The behavior of a bag is captured by an interface called `Collection`, which exists in the `java.util` package.

Now, what if you want a collection that ensures objects can be retrieved from it

in the same order in which they were added to it? Or what if you want a collection that does not allow duplicate objects? You can think of several features that can be added on top of a basic collection. As you have already learnt previously, subclassing/subinterfacing is the way you extend the behavior of an existing class or interface. The Java standard library takes the same route here and extends `java.util.Collection` interface to provide several interfaces with various features. In fact, the Java standard library provides a huge tree of interfaces along with classes that implement those interfaces. These classes and interfaces are collectively known as the "**Collections API**" .

So what has `ArrayList` got to do with this, you ask? Well, `ArrayList` is a class that implements one of the specialized collection interfaces called `List` . `java.util.List` extends `java.util.Collection` to add the behavior of ordering on top of a simple collection. A `List` is supposed to remember the order in which objects are added to the collection. Okay, so that takes care of the "List" part of `ArrayList`, what about the "Array" part? Again, as you have learnt in previous chapters, an interface merely describes a behavior. It doesn't really implement that behavior. You need a class to implement the behavior. In this case, `ArrayList` is the class that implements the behavior described by `java.util.List` and to implement this behavior, it uses an array. Hence the name `ArrayList` . Remember I talked about writing code to overcome the limitation of an array that it is not flexible enough to increase in size automatically? `ArrayList` contains that code. When you add an object to an `ArrayList` , it checks whether there is space left in the array. If not, it allocates a new array with a bigger length, copies the elements from the old array to this new array, and then adds the passed object to the new array. Similarly, it contains code for inserting an object in the middle of the array. All this is transparent to the user of an `ArrayList` . A user simply invokes methods such as `add` and `remove` on an `ArrayList` without having any knowledge of the array that is being used inside to store the object.

Generics ↗

Now, regarding the warning messages that I asked you to ignore. Observe the `for` loop in the above code. The type of the loop variable `o` is `Object` (and not `String`). To invoke the `length` method of `String` on `o` , we need to cast `o` to `String` . This is because, in the same way that a bag is unaware of what is

inside of it, so too does the `ArrayList` object have no knowledge about the type of the objects that have been added to it. It is the responsibility of the programmer to cast the objects that they retrieve from the list to appropriate types. In this program, we know that we have added `String`s to this list and so we know that we can cast the object that we retrieve from this list to `String`. But what if we were simply given a pre-populated list as an argument? Since the list would have been populated by another class written by someone else, what guarantee would we have about the type of objects we find in the list? None, actually. And as you are aware, if we try to cast an object to an inappropriate type, we get a `ClassCastException` at run time. Getting a `ClassCastException` at run time would be a coding mistake that will be discovered only at run time. Discovering coding mistakes at run time is not a good thing and the compiler is only trying to warn us about this potential issue by generating warning messages while compiling our code.

The first warning message that it prints out is, "Warning: unchecked call to add(E) as a member of the raw type `java.util.ArrayList`" for the code `al.add("alice");` at line 8. It prints the same warning every time an object is added to the list. The compiler is trying to tell us that it does not know what kind of objects this `ArrayList` is supposed to hold and that it has no way of checking what we are adding to this list. By printing out the warning, the compiler is telling us that it will not be held responsible for the objects that are being put in this list. Whoever wants to retrieve objects from this list must already know what type of objects are present inside the list and must cast those objects appropriately upon retrieval at their own risk. In other words, this list is basically "**type unsafe**". It is unsafe because it depends on assumptions made by humans about the contents of the list. This assumption is not checked or validated by the compiler for its truthfulness. One can put any kind of object in the list and others will not realize until they are hit with a `ClassCastException` at run time when they try to cast the retrieved object to its expected type.

Java solves this problem by allowing us to specify the type of objects that we are going to store in a list while declaring the type of the list. If, instead of writing `ArrayList al = new ArrayList();`, we write `ArrayList<String> al = new ArrayList<String>();` all the warning messages go away. The compiler now knows that the `ArrayList` pointed to by `al` is supposed to hold only `String`s. Therefore, it will be able

to keep a watch on the type of objects the code tries to add this list. If it sees any code that attempts to add anything that is not a **String** to the list, it will refuse to compile the code. The error message generated by the following code illustrates this point:

```
ArrayList<String> al = new ArrayList<String>();  
al.add(new Object());
```

The error message is:

```
Error: no suitable method found for add(java.lang.Object)  
       method java.util.Collection.add(java.lang.String)  
is not applicable
```

The compiler will not let you corrupt the list by adding objects that this list is not supposed to keep. Similarly, the compiler will not let you make incorrect assignments either. Here is an example:

```
List<String> al = new ArrayList<String>(); //al points  
to a List of Strings.
```

```
al.add("hello"); //valid
```

```
String s = al.get(0); //valid, no cast required
```

```
Object obj = al.get(0); //valid because a String is-a  
Object
```

```
Integer in = al.get(0); //Invalid. Will not compile.
```

The error message is:

Error: incompatible types: java.lang.String cannot be converted to java.lang.Integer

Observe that we supplied information about the type of object the list is supposed to keep by appending `<String>` to the variable declaration and the `ArrayList` creation. This way of specifying the type information is a part of a feature introduced in Java 5 known as "**generics**".

The above discussion should tell you that the topics of List/ArrayList, Collections, and Generics are interrelated. All three of them combine together to provide you a powerful mechanism to write type safe and bug free code. Although the exam objectives mention only List and ArrayList explicitly, we have seen questions that require you to have a basic understanding of the **Collections API**, the **Collection** and **Map** interfaces, the **HashMap** class and generics as well in the exam. So, before I move on to the API details of the important classes, I will take a diversion to discuss Generics in a little more detail.

13.3.2 Generics

Although Generics is a huge and complicated topic that deserves a chapter or two of its own, I will try to introduce it to you in an easy to understand manner. My objective here is not to overwhelm you with too much complexity but to provide you with the foundation required by the OCP Java 11 Part 1 exam.

The whole purpose of introducing generics in Java is to enhance type safety of the code. Type safety is important because it helps prevent a category of bugs that occur due to wrong assumption about the type of objects at run time. For example, if the developer assumes that an object pointed to by a variable will be of type `CheckingAccount` and writes the code accordingly, and if at run time it turns out that the object is actually of type `SavingsAccount`, the code will fail. Had the compiler warned the developer that their assumption will turn out to be wrong at run time, the developer would have fixed the code right then and there.

Here, "making an assumption" essentially means casting a variable that was declared to be of one type to another type. You have seen cases where the compiler will not allow you to make such a cast. For example, the compiler will simply refuse to compile the cast in this code - `Integer i = 10; String s = (String) i;`. The compiler knows that the declared type of `i` is `Integer` and since `String` and `Integer` classes do not have a parent-child relationship, there is no way `i` can ever point to an object of type `String`. The compiler correctly points out that this cast is a programming mistake. But, as you saw with the code snippet involving an `ArrayList` in the previous section, there are cases where the compiler does not have enough information to prevent the developer from making such incorrect assumptions. In that code, the compiler had no knowledge about the type of the objects contained in the list and so, it had no option but to allow the compilation to succeed. It simply generated a warning to let the programmer know that the code may run into issues at run time.

Parameterized classes [↳](#)

The first challenge, therefore, in enhancing the type safety in a piece of code is to add type information to a class, which the compiler can use to prevent inappropriate casts. One could achieve that by creating a class with more specific types instead of `Object`. For example, one could copy the code from `ArrayList`, make some changes to it and create a `StringList` class that would use `String`, instead of `Object`, as the type of input parameters and return types in its methods. This solution has an obvious shortcoming. You will need to define a new class every time you need to create a list of a new type. Just imagine the horror if you find out a bug in the code. You would have to fix it in all the cloned classes. It is an impractical solution. The second challenge, therefore, is to prevent duplication of the code. So, what we really need is a way to write a class in a generic fashion, without hardcoding it to any particular type, and also allow that class to be typed to any type as per the requirement, at the time of using it.

This is exactly what "parameterized" classes in Java allow you to do. Instead of using an actual type while coding a class, you use a placeholder. Whoever uses the class later must specify the value for that placeholder so that the class can be typed to that type. I know it sounds complicated but the following example will

make this clear:

```
public class DataHolder<E> {  
    E data;  
    public E getData(){ return data; }  
    public void setData(E e){ data = e; }  
}
```

In the above code, instead of using any specific class as the type of **data** variable, I have used a placeholder named **E**. The name of the placeholder doesn't have to be **E**. You could name it anything but conventionally, only a single capital letter is used for this purpose. Observe the part where it says **<E>** in the class declaration. This tells the compiler that **E** is just a placeholder and that **E** will be replaced by the actual type later on when this class is actually used by another piece of code. The code in **DataHolder** is written as if the type of the **data** variable is **E**. The code in this class is generic enough to not care about the type with which **E** is replaced. In technical parlance, **DataHolder** is a **parameterized** class, because it uses a **type parameter** (**E** is the type parameter, in this case) for a type that it uses in its code.

Let us now see how this class can be used by other people. Let us say there is a class that consumes a **DataHolder** object containing **String**. Here is the code for such a class:

```
public class SomeClass {  
    public static void consumeData(DataHolder<String>  
        stringData){  
        String s = stringData.getData(); //no cast re  
        quired  
  
        //do something with s  
  
        //Integer i = (Integer) stringData.getData();  
        //will not compile
```

```
    }  
}
```

Observe the usage of `<String>` while declaring `stringData` parameter. Here, `String` is the parameter (a **type argument**, to be precise) that is being passed to the parameterized class `DataHolder`. In technical parlance, we call it "typing" `DataHolder` to `String`. It tells the compiler that only a `DataHolder` object containing `String` can be passed to this method. With this declaration, the compiler gets all the information that it needs to prevent a programmer from making wrong assumption about what the argument contains. Note that this typing applies only to the `stringData` variable and not to all `DataHolder` objects in the program. Only the `DataHolder` object referred to by the `stringData` variable is being guaranteed to contain `String` here. A program could certainly use another `DataHolder` variable typed to a different type.

There are three important points to understand in the above code.

First, no cast is required to assign the value returned by `getData` to `s`. The compiler knows that the `DataHolder` object pointed to by `stringData` has been typed to `String`. So, `stringData.getData()` will always return a `String`.

Second, the compiler will not let you cast the value returned by `stringData.getData()` to any type that is not a `String`. The reason is the same. The compiler knows that `stringData.getData()` can never return anything other than a `String` and so casting the return value to `Integer` is a programming error.

Finally, third, the compiler will not let you invoke the `consumeData` method with anything other than a `DataHolder` typed to `String`. For example, the following code in yet another class will not compile:

```
public class TestClass {  
    public static void main(String[] args){
```

```

        DataHolder<Integer> dh = new DataHolder<Intege
r>();
        SomeClass.processData(dh); //will not compile

    }
}

```

The compiler knows that `SomeClass.processData` expects a `DataHolder` containing `String`. So, it will not let the programmer make the mistake of invoking this method with a `DataHolder` typed to `Integer`.

Observe that the compiler is able to guarantee that the call to `stringData.getData()` inside `processData()` method will return a `String` only because it prevents an invocation of `processData` method with a `DataHolder` typed to anything other than `String` from compiling in the first place! No loose ends, right?

You may wonder at this point how the above code is better than simply declaring the `data` variable as `Object`. Something like this:

```

public class DataHolder{
    Object data;
    public Object getData(){ return data; }
    public void setData(Object e){ data = e; }
}

```

After all, the above code also allows a `DataHolder` to contain any type of object. But there lies the problem. We don't know the type of the object that a given `DataHolder` object contains. In other words, we do want the `DataHolder` object to be able to contain any type of object but not without us knowing what kind of object it contains at any given point. Furthermore, we don't want `DataHolder` object to contain any other type of object once we decide what type of object it should contain.

Parameterized Methods

Parameterized methods are similar to parameterized classes. The only difference is that the type parameter is valid only for that method instead of the whole class. Here is an example:

```
public class ListProcessor{  
    public static <T> T processList(List<T> listOfT){  
        return listOfT.get(0);  
    }  
}
```

In the above code, **T** is used as a placeholder for the actual type in the **processList** method. This method takes a **List** of **T**s and returns a **T**. Unlike the placeholder that was used in the **DataHolder** class, **T** is valid only within this method. So, if you pass a **List<String>** to this method, it will return a **String**.

Java 5 parameterized the code for all the classes/interfaces in the Collections API so that you could use type safe collections in your code. That is why you are able to type **ArrayList** to any type you want using the **< >** syntax. You may go ahead and look at the source code of any of the Collection API classes to get a feel of how exactly it has been parameterized.

Using parameterized types [↳](#)

You will not be tested on the syntax of defining a parameterized class or a parameterized method but you should be aware of it because such code may appear in the real exam questions. Furthermore, even though defining parameterized types is not on the part 1 exam (it is on the part 2 exam), using a parameterized type or a method is. The code snippets presented in exam questions use parameterized types quite liberally. So, you need to know the following ways in which you can declare type safe variables and instantiate parameterized types to understand what the code presented in a question is trying to do:

1. A type safe variable can be declared by specifying the type within `< and >` .
For example,

```
ArrayList<String> stringArray;  
List<Integer> iList;  
ArrayList<com.acme.Foo> fooList;
```

`List<> iList; //invalid, type must be specified inside <> while declaring a variable`

2. A parameterized class can be instantiated similarly by specifying the type within `< and >` . For example,

```
new ArrayList<String>();  
new ArrayList<Integer>();  
new ArrayList<com.acme.Foo>();
```

3. If you are instantiating a type safe `ArrayList` while assigning it to a type safe variable in the same statement, then you may omit the type name from the instantiation as shown below:

```
ArrayList<String> al = new ArrayList<>  
( );
```

Upon encountering `<>` , the compiler infers the type of the `ArrayList` object from the type of the variable. The `<>` operator is known as the "**diamond operator**". It can be used only in a `new` statement. Thus, the statement `ArrayList<> al;` will not compile but `new ArrayList<>();` will. This operator was introduced in Java 7 with the sole purpose of saving you from typing a few extra keystrokes.

Remember that you cannot apply the `< >` syntax to just about any class. You can

apply it only to classes that have been parameterized.

Generics and Type Erasure [🔗](#)

Generics were added quite late to Java (only in 1.5) and so backward compatibility was an important concern. Furthermore, since it was a fairly complicated feature, it was expected that their wide spread usage will take some time. Thus, it was deemed necessary that code that uses Generics must work with code that doesn't use Generics. Basically, there are two scenarios that had to be supported. Generic aware Java source code should be able use preexisting classes that did not use generics and non-generics source code should be able to use new generic aware libraries.

To achieve this design goal, the mechanism of **type erasure** was used, where all the generic information is stripped from a class at run time. In other words, the JVM does not get to know any of the type information that is present in a class file in the form of generics. This information is used only by the compiler to perform its type checking of the source code at the time of compilation. Thus, even if you write `ArrayList<String> al = new ArrayList<>();` in your code, the JVM will only see `ArrayList al = new ArrayList();`. The type information is effectively "erased" from a class for the JVM. This implies that the Java code `String str = al.get(0);` gets translated to `String str = (String) al.get(0);` in the class file. Observe that this is completely opposite to the behavior displayed by arrays. As discussed in the Arrays chapter, arrays are reified, which means the type information stays with an array at all times.

Type erasure eliminates the need to create separate class files for differently typed usage of a parameterized class. This ensures backward compatibility because there is essentially no difference between generics aware code and regular code from the JVM's perspective.

Type erasure has a serious impact on overloading and overriding of methods. Although not mentioned explicitly, the exam has questions that touch upon this aspect. So, let's dig a little deeper into it.

Impact of Type Erasure on Method Overloading [🔗](#)

As explained in the previous chapters, overloaded methods are methods that differ in their signature. Recall that a method signature is made up of method name plus parameter types. Since, due to type erasure, generic information is removed at run time, it follows that generic information cannot be a part of the signature. Thus, two methods that differ only in their type parameters are not valid overloads. The compiler will refuse to compile such code because it knows that the JVM will not be able to distinguish between them at run time. For example, the following code won't compile:

```
public class TestClass{
    public void processData(DataHolder<String> stringData){
        String s = stringData.getData();
        System.out.println(s);
    }

    public void processData(DataHolder<Integer> intData)
    {
        Integer i = intData.getData();
        System.out.println(i);
    }
}
```

The error message will be `error: name clash: processData(DataHolder<Integer>) and processData(DataHolder<String>) have the same erasure.` Indeed, once the generic information is removed, signature of both the methods are same and there is no way for the JVM to bind an invocation of this method to any one version.

Impact of Type Erasure on Method Overriding [🔗](#)

Overriding a method that uses generic types has the same issue that I talked about above. The following code fails to compile:

```

class Base{
    public void processData(DataHolder<String> stringData){
        String s = stringData.getData();
        System.out.println(s);
    }
}
public class SubClass extends Base{
    public void processData(DataHolder<Integer> intData){
        Integer i = intData.getData();
        System.out.println(i);
    }
}

```

The error message in this case is: `error: name clash: processData(DataHolder<Integer>) in SubClass and processData(DataHolder<String>) in Base have the same erasure, yet neither overrides the other.`

In this case, the overriding method is trying to break the contract promised by the base class. The base class's `processData` version expects a `DataHolder` containing `String`, while the subclass's version expects a `DataHolder` containing `Integer`. Although after erasing the type information, the subclass method seems to correctly override the base class's method. However, the compiler notices that the cast to `Integer` in the subclass's method will certainly fail at run time because `intData.getData()` will return a `String`! Therefore, it refuses to compile the code.

The presence of generics can throw up complicated situations with respect to Overloading and Overriding. However, the exam does not grill you on this topic. You are only expected to identify simple cases of overloading and overriding involving generics. It is important to **not panic** upon seeing such questions. You will be able to answer it by applying the rules explained above.

There are still several important concepts about Generics such as mixing generic and non-generic code, multiple type parameters, and bounded type parameters that I have not discussed in this book because they are way beyond the scope of the part 1 exam. However, you should read about them if you are going to face technical interviews.

13.3.3 Quiz

Given the following two classes:

```
class Base{  
    public <T> List<T> processList(List<T> list){ return  
        null; }  
}
```

and

```
public class SubClass extends Base{  
    //add method here  
}
```

Which of the following options are true if the method is added to **SubClass** ?

- A. `public <T> List<T> processList(List<T> list){ return
 null; }` correctly **overrides** `processList` of `Base` .
- B. `public <T> Collection<T> processList(List<T> list){
 return null; }` correctly **overrides** `processList` of `Base` .

C. `public List<String> processList(List<String> list){ return null; }` correctly **overrides** `processList` of `Base`.

D. `.. public <T> ArrayList<T> processList(List<T> list){ return null; }` correctly **overrides** `processList` of `Base`.

E. `public <T> List<T> processList(ArrayList<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

F. `public <T> List<T> processList(List<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

G. `public <T> List<T> acceptList(List<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

Correct answer is A, D, E

Remember that when a subclass contains a method with the same signature as a method in the super class, it is called an **override** and when a class contains (or inherits) multiple methods with the same name but different parameter types, it is called an **overload**.

Option **A** is obviously **correct** because the method signature of a method in the subclass matches exactly to the method signature of a method in the super class. It is a valid override.

Option **B** could have been a valid override but the return type of the overriding method cannot return a wider type. Recall the rule of **covariant returns**. So, it is **incorrect**.

Option **C** is **incorrect** because of a mismatch in the generic types of the method parameters and the return type as explained in this section.

Option **D** is **correct** because the method signatures match and the subclass method follows the rule of covariant returns. `ArrayList` is a narrower type than `List` because `ArrayList` implements `List`.

Option **E** is **correct** because the parameter type of the method in subclass is

different. The subclass, therefore, has two methods (one of its own and one inherited) with the same name but different parameter types. Thus, it is a valid overload.

Option F is **incorrect** because it is a valid override and not a valid overload.

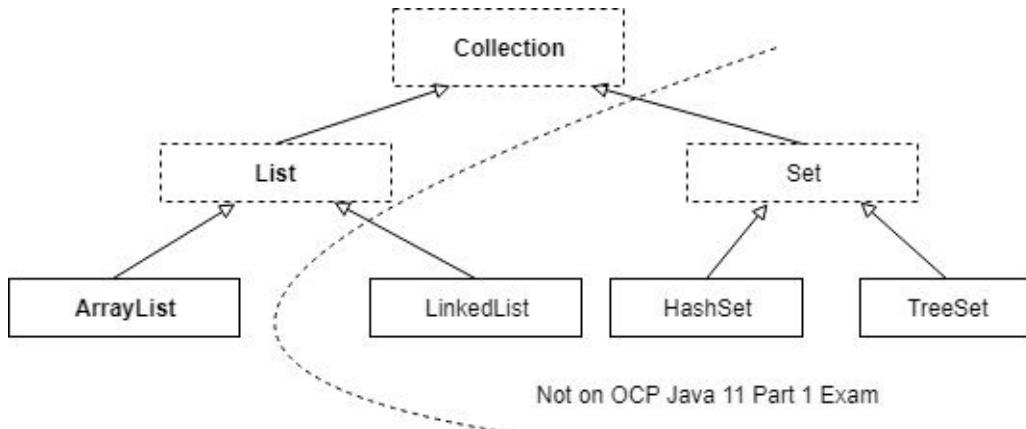
Option G is **incorrect** because the method name doesn't match. It is neither an overload nor an override. It is a valid method though.

13.3.4 Collection and List API [↳](#)

Collection API [↳](#)

java.util.Collection is the root interface in the collections hierarchy. It abstracts the concept of a collection in general. It declares only the methods that are applicable to all sorts of collections and leaves the methods that deal with specialized features such as ordering or uniqueness of elements to sub-interfaces such as List and Set. For example, **Collection** declares **add(Object e)** and **remove(Object e)** methods but does not declare **add(int index, Object e)** and **remove(int index)** because the concept of ordering is not applicable to all collections. We use the Collection interface when we don't want to make any assumption about the characteristics of the group of objects. In other words, if we get a Collection object, all we know is that it contains a bunch of elements. We don't know whether the elements are ordered in any way, whether the group contains duplicate elements, or what is the data structure used to keep the elements in that collection.

The following figure shows the hierarchy of interfaces and classes that you will see on the exam. The rectangles with dashed borders are interfaces and the rectangles with solid borders are classes.



Although the exam does not expect you to remember which method is declared in **Collection** and which is not, it is still a good idea to go through the JavaDoc API description of this interface and check out the methods declared in this interface. The method names are quiet descriptive and the methods do what their names suggest. Some of the methods that you should pay attention to are: **add** , **addAll** , **remove** , **removeAll** , **removeIf** , **contains** , **containsAll** , **isEmpty** , and **clear** .

List API [🔗](#)

java.util.List interface defines the behavior of collections that keep objects in an order. Elements of a list can be accessed using an index. It also allows elements to be inserted in or removed from any given index. Duplicate elements as well as nulls are permitted. Since it is just an interface, it doesn't specify how the functionality is implemented. Actual implementation classes such as **ArrayList** and **LinkedList** use different mechanisms to implement the behavior. As you will see later with **ArrayList** , implementation classes may provide additional features on top of the features defined in **List** .

Again, you should go through all of the methods of the **List** interface from the JavaDoc. The exam doesn't trick you on method names and doesn't require you to recall whether a given method exists in List or not. The exam questions are based on the understanding of how a given method behaves. You are expected to know what happens to the existing list of objects after the invocation of a method. The trick is in knowing "boundary conditions". For example, what happens if you try to remove a non-existing object or a null from a list or what happens if you try to add null.

I will list the important methods of `List` interface below. Remember that index is zero based, which means, the index of the first element is zero.

1. `E get(int index)` : Returns the element at index.
2. `E set(int index, E e)` : Replaces the existing element at index with the passed object and returns the original element that was replaced.
3. `void add(E e)` : Adds an object at the end of the list.
4. `void add(int index, E e)` : Adds an object at the specified index and shifts the existing elements at a higher index to the right.
5. `void addAll(Collection c)` : Adds all the elements of the given collection to this list at the end.
6. `void addAll(int index, Collection c)` : Adds all the elements of the given collection at the given index.
7. `E remove(int index)` : Removes an object from the specified index and shifts the existing elements at a higher index to the left. It returns the original element that was removed from the list.
8. `boolean remove(E e)` : Removes the first occurrence of given object from the list. Returns true if the object was found. The passed object is matched with the objects in the list using the equals method.
9. `void removeAll(Collection c)` : Removes from this list all the elements that are present in the passed collection.
10. `void retainAll(Collection c)` : Removes from this list all the elements that are not in the passed collection.
11. `void clear()` : Removes all the elements from this list.
12. `int size()` : Returns the number of elements in this list.
13. `default void forEach(Consumer<? super T> action)` : Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
You will see this method used with lambda expressions a lot.

The `index` value must lie between 0 and list's current size (both inclusive) otherwise an `IndexOutOfBoundsException` is thrown. A typical question that tests your understanding of the above methods would be the predict to output of the following code:

```
List<Integer> list = new ArrayList<Integer>();  
list.add(0, 1);
```

```
list.add(0, 2);
list.add(0, 3);
System.out.println(list);
```

If you think the output is [1, 2, 3], you have been tricked. The output is [3, 2, 1] because the elements are being inserted at the 0 th index.

Here are few methods that help you inspect a list:

1. `boolean isEmpty()` : Returns true if the list has no elements.
2. `boolean contain(Object o)` : Returns true of the list contains the given object.
3. `boolean containsAll(Collection c)` : Returns true of the list contains all of the elements present in the given collection.
4. `List subList(int fromIndex, int toIndex)` : Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. If fromIndex and toIndex are equal, the returned list is empty.
5. `int indexOf(Object o)` : Returns the index at which the given object is found in the list. Returns -1 if it is not found.
6. `int lastIndexOf(Object o)` : Returns the last index at which the given object is found in the list. Returns -1 if it is not found.
7. `Object[] toArray()` : Returns a array of Objects containing all the elements of this list.
8. `<T> T[] toArray(T[] a)` : Returns an array of the type specified in the argument containing all the elements of this list. The objects of the list must be of the type passed in the argument. This method is helpful when you want to convert a list into an array of a specific type. For example, `String[] strArray = listOfString.toArray(new String[0]);` will return a String[] containing all the elements of the list.

The `subList(int fromIndex, int toIndex)` method in the above list of methods is particularly interesting. It does not return a new independent list but just a view into the original list. Thus, any change you do to the view is reflected in the original list as shown the example below:

```
List<String> al = new ArrayList<String>();
```

```
al.addAll(Arrays.asList( new String[]{"a", "b", "c", "d", "e" } ));  
List<String> al2 = al.subList(2, 4);  
System.out.println(al2); //prints [c, d]
```

```
al2.add("x");  
System.out.println(al2); //prints [c, d, x]
```

```
System.out.println(al); //prints [a, b, c, d, x, e]
```

```
al.add("y"); //structural modification to the original  
list
```

```
System.out.println(al2); //throws java.util.Concurrent  
ModificationException
```

Observe the output of the last two lines of the output. `x` was added to the end of the list pointed to by `al2`. However, since `al2` is just a view of a portion of the original list pointed to by `al`, `x` is visible in the middle of this list. Furthermore, all structural modifications must be done through the view. If you structurally modify the backing list and then try to use the view, the results will be unpredictable. Structural modification means any change in size of the list. That is why, the last line of the above code throws a `java.util.ConcurrentModificationException` exception.

List.of and List.copyOf methods ↗

Java has had the `java.util.List` interface ever since the Collections API was introduced in Java 1.2. It was enhanced in Java 9 and 10 with several static methods and the exam expects you to know them. These are the twelve

overloaded `of` (added in Java 9) methods and the `copyOf` method (added in Java 10)..

The overloaded `of` methods are merely convenience methods that take 0 to 10 parameters and return an **unmodifiable** List. For example, if you want to create a list of 3 Strings, you could do `List<String> list1 = List.of("a", "b", "c");`.

Similarly, the `copyOf` method is a convenience method to create an **unmodifiable** list using the elements of an existing collection. For example, `List<String> list2 = List.copyOf(list1);`.

You will see the usage of these methods all over exam questions even when the question is not about lists. Since understanding the behavior of the lists returned by these methods is very important for the exam, let me first jot down their characteristics as given in JavaDoc:

1. They are **unmodifiable**. Elements cannot be added, removed, or replaced. Calling any mutator method on the List will always cause `UnsupportedOperationException` to be thrown. However, if the contained elements are themselves mutable, this may cause the List's contents to appear to change.
2. They disallow **null** elements. Attempts to create them with null elements result in `NullPointerException`.
3. They are serializable if all elements are serializable. (Not important for the Part 1 exam)
4. The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.
5. They are value-based. Callers should make no assumptions about the identity of the returned instances. Factories are free to create new instances or reuse existing ones. Therefore, identity-sensitive operations on these instances (reference equality (`==`), identity hash code, and synchronization) are unreliable and should be avoided.

Here is the same code that I showed above but with a small modification:

```
List<Integer> list = List.of();
```

```
list.add(0, 1);
list.add(0, 2);
list.add(0, 3);
System.out.println(list);
```

Before jumping on to answer [3, 2, 1] this time, you need to appreciate the fact that the list returned by `List.of` is unmodifiable! So, the code will compile but the first call to `list.add` will throw an `java.lang.UnsupportedOperationException` exception at run time.

13.3.5 ArrayList API

As discussed earlier, `ArrayList` is one of the implementation classes of the `List` interface. It has three constructors .

1. `ArrayList()` : Constructs an empty list with an initial capacity of 10.
Just like you saw with the `StringBuilder` class, capacity is simply the size of the initial array that is used to store the objects. As soon as you add the 11th object, a new array with bigger capacity will be allocated and all the existing objects will be transferred to the new array.
2. `ArrayList(Collection c)` : Constructs a list containing the elements of the specified collection.
3. `ArrayList(int initialCapacity)` : Constructs an empty list with the specified initial capacity. This constructor is helpful when you know the approximate number of objects that you want to add to the list. Specifying an initial capacity that is greater than the number of objects that the list will hold improves performance by avoiding the need to allocate a new array every time it uses up its existing capacity.

It is possible to increase the capacity of an `ArrayList` even after it has been created by invoking `ensureCapacity(int n)` on that `ArrayList` instance. Calling this method with an appropriate number before inserting a large number of elements in the `ArrayList` improves

performance of the add operation by reducing the need for incremental reallocation of the internal array. The opposite of `ensureCapacity` is the `trimToSize()` method, which gets rid of all the unused space by reducing its capacity to the match the number of elements in the list.

Here are a few declarations that you may see on the exam:

```
List<String> list = new ArrayList<>(); //ok because ArrayList implements List
```

```
var al = new ArrayList<Integer>(50); //initial capacity is 50
```

```
ArrayList<String> al2 = new ArrayList<>(list); //copying an existing list, observe the diamond operator
```

```
var list1 = new ArrayList<>(); //ok, list1 is of type ArrayList<Object>, observe the combination of var declaration and the diamond operator
```

```
List list2 = new List(list); //will not compile because List is an interface, it cannot be instantiated
```

Important methods of `ArrayList` ↴

`ArrayList` has quite a lot of methods. However, since `ArrayList` implements `List` (which, in turn, extends `Collection`), several of

`ArrayList`'s methods are declared in `List` and `Collection` interfaces. The exam does not expect you to make a distinction between the methods inherited from `List /Collection` and the methods declared in `ArrayList`

The following are the ones that you need to know for the exam:

1. `String toString()` : Well, `toString` is not really the most important method of `ArrayList` but since we will be depending on its output in our examples, it is better to know about it anyway. `ArrayList`'s `toString` first gets a string representation for each of its elements (by invoking `toString` on them) and then combines into a single string that starts with [and ends with]. For example, the following code prints [a, b, c] :

```
var al = new ArrayList<String>();
al.add("a");
al.add("b");
al.add("c");
System.out.println(al);
```

Observe the order of the elements in the output. It is the same as the order in which they were added in the list. Calling `toString` on an empty `ArrayList` gets you []. I will use the same format to show the contents of a list in code samples.

Methods that add elements to an `ArrayList` :

1. `boolean add(E e)` : Appends the specified element to the end of this list. As discussed in the Generics section, `E` is just a place holder for whichever type you specify while creating the `ArrayList`. For example, if you create an `ArrayList` of Strings, i.e., `ArrayList<String>`, `E` stands for `String`.

This method is actually declared in `Collection` interface and the return

value is used to convey whether the collection was changed as a result of calling this method. In case of an `ArrayList`, the `add` method always adds the given element to the list (even if the element is null), which means it changes the collection every time it is invoked. Therefore, it always returns `true`.

2. `void add(int index, E element)` : Inserts the specified element at the specified position in this list. The indexing starts from `0`. Therefore, if you call `add(0, "hello")` on an list of `Strings`, "hello" will be inserted at the first position.
3. `boolean addAll(Collection<? extends E> c)` : Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator. Again, for the purpose of the Part 1 exam, you don't need to worry about the "`?` `extends E`" or the Iterator part. You just need to know that you can add all the elements of one list to another list using this method. For example:

```
ArrayList<String> sList1 = new ArrayList<>(); //observe the usage of the diamond operator
```

```
sList1.add("a"); // [a]
```

```
ArrayList<String> sList2 = new ArrayList<>();
sList2.add("b"); // [b]
```

```
sList2.addAll(sList1); // sList2 now contains [b, a]
```

4. `boolean addAll(int index, Collection<? extends E> c)` : This method is similar to the one above except that it inserts the elements of the passed list in the specified collection into this list, starting at

the specified position. For example:

```
ArrayList<String> sList1 = new ArrayList<>();  
sList1.add("a"); // [a]
```

```
ArrayList<String> sList3 = new ArrayList<>();  
sList3.add("b"); // [b]
```

```
sList3.addAll(0, sList1); // sList3 now contains [a,  
, b]
```

Methods that remove elements from an [ArrayList](#) :

1. [E remove\(int index\)](#) : Removes the element at the specified position in this list. For example,

```
ArrayList<String> list = ... // an ArrayList conta  
ining [a, b, c]
```

```
String s = list.remove(1); // list now has [a, c]
```

It returns the element that has been removed from the list. Therefore, [s](#) will be assigned the element that was removed, i.e., "b". If you pass an invalid int value as an argument (such as a negative value or a value that is beyond the range of the list), an [IndexOutOfBoundsException](#) will be thrown.

2. [boolean remove\(Object o\)](#) : Removes the first occurrence of the specified element from this list, if it is present. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, a]  
  
list.remove("a"); // [b, a]
```

Observe that only the first `a` is removed.

You have to pay attention while using this method on an `ArrayList` of `Integers`. Can you guess what the following code will print?

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3} ));  
list.remove(1);  
System.out.println(list);  
list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3} ));  
list.remove(new Integer(1));  
System.out.println(list);
```

The output is:

```
[1, 3]  
[2, 3]
```

Recall the rules of method selection In case of overloaded methods. When you call `remove(1)`, the argument is an `int` and since a remove method with `int` parameter is available, this method will be preferred over the other remove method with `Object` parameter because invoking the other method requires boxing `1` into an `Integer`.

This method returns `true` if an element was actually removed from the list as a result of this call. In other words, if there is no element in the list that matches the argument, the method will return `false`.

3. `boolean removeAll(Collection<?> c)` : Removes from this list all of its elements that are contained in the specified collection. For example the following code prints [c] :

```
ArrayList<String> al1 = new ArrayList<>(Arrays.asList( new String[]{"a", "b", "c", "a" } ));  
ArrayList<String> al2 = new ArrayList<>(Arrays.asList( new String[]{"a", "b" } ));  
al1.removeAll(al2);  
System.out.println(al1); //prints [ c ]
```

Observe that unlike the `remove(Object obj)` method, which removes only the first occurrence of an element, `removeAll` removes all occurrences of an element.

This method returns `true` if an element was actually removed from the list as a result of this call.

4. `void clear()` : Removes all of the elements from this list.

Methods that replace an element in an `ArrayList` :

1. `E set(int index, E element)` : Replaces the element at the specified position in this list with the specified element. It returns the element that was replaced.

Example:

```
ArrayList<String> al = ... // create a list containing [a, b, c]
```

```
String oldVal = al.set(1, "x");
```

```
System.out.println(al); //prints [a, x, c]
```

```
System.out.println(oldVal); //prints b
```

Methods that read an `ArrayList` without modifying it:

1. `boolean contains(Object o)` : The object passed in the argument is compared with each element in the list using the `equals` method. A `true` is returned as soon as a matches is found, a `false` is returned otherwise. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();  
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c" } ));  
System.out.println(al.contains("c")); //prints true
```

```
System.out.println(al.contains("z")); //prints false
```

```
System.out.println(al.contains(null)); //prints true
```

Observe that it does not throw a `NullPointerException` even if you pass it a `null`. In fact, a `null` argument matches a `null` element.

2. `E get(int index)` : Returns the element at the specified position in this list. It throws an `IndexOutOfBoundsException` if an invalid value is passed as an argument.

3. `int indexOf(Object o)` : The object passed in the argument is compared with each element in the list using the equals method. The index of the first element that matches is returned. If this list does not contain a matching element, -1 is returned. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c", null } ));
System.out.println(al.indexOf("c")); //prints 3
```

```
System.out.println(al.indexOf("z")); //prints -1
```

```
System.out.println(al.indexOf(null)); //prints 1
```

Observe that just like `contains` , `indexOf` does not throw a `NullPointerException` either even if you pass it a `null` . A `null` argument matches a `null` element.

4. `boolean isEmpty()` : Returns `true` if this list contains no elements.
5. `int size()` : Returns the number of elements in this list. Recall that to get the number of elements in a simple array, you use the variable named `length` of that array.

The examples that I have given above are meant to illustrate only a single method. In the exam, however, you will see code that uses multiple methods. Here are a few points that you should remember for the exam:

1. **Adding nulls** : `ArrayList` supports `null` elements.
2. **Adding duplicates** : `ArrayList` supports duplicate elements.
3. **Exceptions** : None of the `ArrayList` methods throw `NullPointerException` . They throw `IndexOutOfBoundsException` if you try to access an element beyond the range of the list.

4. **Method chaining** : Unlike `StringBuilder` , none of the `ArrayList` methods return a reference to the same `ArrayList` object. Therefore, it is not possible to chain method calls.

Here are a few examples of the kind of code you will see in the exam. Try to determine the output of the following code snippets when they are compiled and executed:

1. `var al = new ArrayList<Integer>(); //observe that the type specification is on the right side`

```
al.add(1).add(2);
System.out.println(al);
```

2. `ArrayList<String> al = new ArrayList<>();//observe the usage of the diamond operator`

```
if( al.add("a") ){
    if( al.contains("a") ){
        al.add(al.indexOf("a"), "b");
    }
}
System.out.println(al);
```

3. `ArrayList<String> al = new ArrayList<>();
al.add("a"); al.add("b");
al.add(al.size(), "x");
System.out.println(al);`

4. `var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>();
list1.add("a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.remove("b");
System.out.println(list1);`

5.

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
System.out.println(list1);
list1.remove("b");
System.out.println(list1);
```

6.

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.removeAll("b");
System.out.println(list1);
```

Size vs Capacity ↴

Size is the number of elements that are currently stored in the `ArrayList`. While **capacity** is the length of the internal array that an `ArrayList` uses to store its elements.

You know that in Java, arrays are of fixed length. You must specify the length of an array while creating it and you can never change this length. So, what happens when you add more elements to an `ArrayList` than its capacity? Simple. A new array with bigger length is allocated and elements are transferred from the old array to the new array. Since all this is managed internally by the `ArrayList`, you don't have to worry about it most of the time.

However, it has implications on the performance of an `ArrayList`. Since an `ArrayList` has no idea how many

elements are going to be added to it, it starts with a capacity of 10 and keeps incrementing the capacity as soon as it is full (the exact algorithm for incrementing the capacity is not important here). Now, imagine if you wanted to add a thousand elements to an ArrayList. This could potentially require the ArrayList to perform the allocation and transfer operation, which is quite expensive, several times. This will reduce its performance.

This is where the `ArrayList(int capacity)` constructor and `ensureCapacity(int capacity)` method come in handy. If you know the number of elements you are going to add to the ArrayList, you should use this constructor/method to make sure that the internal array of the ArrayList is big enough to hold all the elements.

*Remember that insertion of an element depends the **size** (and not **capacity**) of the ArrayList. For example, if the size of an ArrayList is 5, you can't insert an element at index 6 even if the capacity of that ArrayList is 10. It will throw an `IndexOutOfBoundsException`.*

13.3.6 ArrayList vs array

You may get a few theoretical questions in the exam about the advantages and disadvantages of an ArrayList over an array. You have already seen all that we can do with ArrayLists and arrays, so, I am just going to summarize their advantages and disadvantages here.

Advantages of ArrayList

1. **Dynamic sizing** - An ArrayList can grow in size as required. The

programmer doesn't have to worry about the length of the ArrayList while adding elements to it.

2. **Type safety** - An ArrayList can be made type safe using generics.
3. **Readymade features** - ArrayList provides methods for searching and for inserting elements anywhere in the list.

Disadvantages of ArrayList [↳](#)

1. **Higher memory usage** - An ArrayList generally requires more space than is necessary to hold the same number of elements in an array.
2. **No type safety** - Without generics, an ArrayList is not type safe at all.
3. **No support for primitive values** - ArrayLists cannot store primitive values while arrays can. This disadvantage has been mitigated somewhat with the introduction of autoboxing in Java 5, which makes it possible to pass primitive values to various methods of an ArrayList. However, autoboxing does impact performance.

Similarities between ArrayLists and arrays [↳](#)

1. **Ordering** - Both maintain the order of their elements.
2. **Duplicates** - Both allow duplicate elements to be stored.
3. **nulls** - Both allows nulls to be stored.
4. **Performance** - Since an ArrayList is backed by an array internally, there is no difference in performance of various operations such as searching on an ArrayList and on an array.
5. **Thread safety** - Neither of them are thread safe. Don't worry, thread safety is not on the exam, but you should be aware of the fact that accessing either of them from multiple threads, may produce incorrect results in certain situations.

13.3.7 Map and HashMap [↳](#)

Map is data structure that allows you to lookup one piece of information, i.e., value, using another piece of information, i.e., key. For example, a map of country codes and country names will allow you to look up a country name

using the country code, a map of student id and student name will allow you to look up a student name using the student id, and so on. Here, countrycode and student id are the keys and country name and student name are the values. In that sense, a Map is a collection of key-value pairs.

The `java.util.Map` interface of the Collections API captures this behavior. There are several implementation classes that implement the Map interface. The one that is used the most is `java.util.HashMap`.

It is important to know that Map does not extend `Collection` interface. Map is the root of a separate hierarchy of classes and interfaces that is unrelated to the Collection hierarchy. Although the exam does not mention Map and HashMap explicitly, we have seen questions on the exam that require you to know the following methods of Map.

1. `V get(Object key)`: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
2. `V put(K key, V value)`: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value and the old value is returned. It returns null if there was no mapping for key.
3. `V remove(Object key)`: Removes the mapping for a key from this map if it is present. Returns the value or null, if not present.
4. `Set<K> keySet()`: Returns a Set of keys present in the map. Observe that keys are unique, so the return type is Set (not Collection or List).
5. `Collection<V> values()`: Returns a Collection of the values present in the map. Observe that values may be duplicate, so, the return type is Collection.
6. `void clear()`: Removes all entries stored in the map.
7. `int size()`: Returns the number of entries stored in the map.
8. `default void forEach(BiConsumer<? super K, ? super V> action)`: Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. **Note:** This is way out of scope for the Part 1 exam, yet, we have seen a question on lambda expressions that refers to this method in the exam. It is possible that it came up in the Part 1 exam by mistake.

The above methods and their descriptions are enough for the exam but there are several interesting and useful methods in Map and it would be a good idea to go through the JavaDoc to learn more.

The following code illustrates how a map is typically used:

```
import java.util.*;
public class TestClass{
    static Map<String, String> idNameMap = new HashMap<>()
();
    static List<String> keys = new ArrayList(List.of("0"
, "1", "2"));
    static List<String> values = List.of("a", "b", "c");

    public static void buildMap(){
        //add key-value pairs in the map

        for(int i=0; i<keys.size(); i++){
            idNameMap.put(keys.get(i), values.get(i));
        }
        System.out.println(idNameMap); //prints {0=a, 1=b,
2=c}

    }
    public static void main(String[] args) {
        buildMap();
        String value = idNameMap.get("2");
        System.out.println(value);//prints c

        //Integer ivalue = idNameMap.get("2"); //will not
compile

        keys.clear(); //removes all elements from keys Lis
t, doesn't affect the map
```

```
    System.out.println(keys.size()); //prints 0, keys  
is now empty
```

```
    System.out.println(idNameMap.size());//prints 3, m  
ap still has the key value entries
```

```
    idNameMap.remove("1"); //remove the key 1 and its  
associated value from the map
```

```
    idNameMap.clear(); //remove all entries from the m  
ap
```

```
    System.out.println(idNameMap); //prints {}
```

```
}
```

Following are a couple of points that you should observe in the above code.

1. **Map** and **HashMap** are parameterized classes. However, unlike a single type parameter that you saw in the case **List** and **ArrayList**, they use two type parameters. One for the type of key (denoted by K) and one for the type of value (denoted by V). In the above program, the type of the key and the value is the same, i.e., **String**.
2. Since the Map's key and value have been typed to String, the compiler will neither allow you to put anything else in it nor will allow you to retrieve anything other than a String from it.

Map and HashMap are used a lot in Java development and are, therefore, a favorite topic in technical discussions. You should definitely read about them if

you are going to face technical interviews.

13.3.8 Quiz ↗

Given the following two classes:

```
class Base{  
    public void process(Collection c){  }  
}
```

and

```
public class SubClass extends Base{  
    public void process(Map<Integer, String> map){  
        //...code here  
    }  
}
```

Which of the following options are correct?

- A. The **process** method in **SubClass** correctly overrides the **process** method in **Base** .
- B. The **process** method in **SubClass** correctly overloads the **process** method in **Base** .
- C. A call to **super.process(map.values());** can be inserted in the **process** method of **SubClass** .
- D. A call to **super.process(map.keySet());** can be inserted in the **process** method of **SubClass** .
- E. A call to **process(map);** can be inserted in the **process** method of

SubClass .

Correct answer is B, C, D .

Since the type of the arguments of the **process** methods do not match, this is not a case of overriding. It is overloading. Thus, option A is wrong and option B is correct. Since **values()** and **keySet()** methods of **Map** return **Collection** and **Set** respectively (**Set** is a **Collection**), options C and D are correct. Since **Collection** and **Map** do not have a parent child relationship, a **Map** is not a **Collection** and so, option E is incorrect.

13.4 Exercise

- Given the following two classes:

```
public class TestClass{
    public static void main(String[] args){
        Document d = new PdfDocument();
        System.out.println(d.getType()); //should print "pdf"
    }
}
class Document{
    private String type = "dummy";
    private byte[] data;
    //insert appropriate getters and setters
}
```

The above code refers to a class named **PdfDocument** . Write code for

this class such that **TestClass** will print "pdf" when executed.

- Given the following code:

```
class Radio {  
    private double frequency=1.1;  
    //insert appropriate getter and setter  
  
}  
class TV {  
    private int channel = 5;  
    //insert appropriate getter and setter  
  
}  
public class TestClass{  
    public static void main(String[] args){  
        TV t = new TV();  
        Radio r = new Radio();  
        reset(t);  
        reset(r);  
        System.out.println(r.getFrequency()); //should print 0.0  
  
        System.out.println(t.getChannel()); //should print 0  
    }  
}
```

Write code for the reset method in **TestClass** such that **TestClass** will print **0.0** and **0** when executed.

- You are expected to reset several electronic devices in future. Refactor the code given above such that **TestClass**'s **reset** method is able to reset any new device without requiring any change in the method code.
- Given the following code:

```
class Pie{  
    public void makePie(){  
        System.out.println("making pie");  
    }  
}
```

Create two classes **PumpkinPie** and **ApplePie** that extend **Pie**. Override the **makePie** method in these classes. Ensure that the overridden method is also invoked whenever **makePie** is invoked on objects of these classes.

5. Add a static method named **getCalories** in **Pie**:

```
public static int getCalories(){  
    return 100;  
}
```

Create a new class named **Nutritionist** with a method named **printCalories**. This method should take any kind of **Pie** and print the correct number of calories as applicable for the given kind of **Pie**.

6. Given the following two classes:

```
class XMLTransformer {  
    public String transform(String Data){ return "xm  
l data"; }  
}  
  
class NetworkTransformer {  
    public String transform(String Data) throws IOEx  
ception { return "data from network"; }  
}
```

Refactor the two classes by adding an abstract super type named **Transformer**.

7. Create a class named **TransformerFactory** with a method named **getTransformer**. This method should return different types of transformers based on an argument. Use this factory class to get different types of transformers and invoke the **transform** method on them.
8. Create a method that takes an array of strings and returns an **ArrayList** containing the same strings.

9. Update the above method to remove the duplicate elements from the `ArrayList` before returning.
10. Create a method with the signature `switchIt(ArrayList al, int a, int b)`. This method should return the same list but after switching the elements at positions `a` and `b`.

Chapter 14 Lambda Expressions

- Understanding Lambda Expressions

14.1 Understanding Lambda Expressions

14.1.1 Lambda Expressions

While taking training sessions, I have observed that many Java beginners feel intimidated by lambda expressions. Most of the time this fear is because of the name "lambda". The word lambda itself does not create any useful mental picture and that makes it a bit difficult to relate to. On top of that, most books and tutorials start explaining lambda expression as a way to do "functional" programming, which is another buzz word that is hard to relate to when a person is just learning "regular" programming!

Well, I can tell you that there is no need to be afraid. Lambda Expressions are actually very simple. While writing test programs, have you ever been frustrated while typing the words "public static void main(String[] args)" over and over again? I am sure you have been. If you are using an IDE such as NetBeans or Eclipse, then you are probably aware of shortcuts such as `psvm`. You type `psvm`, hit the tab key and the IDE replaces psvm with the text "public static void main(String[] args)" immediately. A lambda expression is just like that but for the compiler. Think of a lambda expression as a shortcut for the compiler that does two things - defines a class with a method and instantiates that class. As soon as the compiler sees a lambda expression, it expands the expression into a class definition and a statement that instantiates that class. If you think of it as a tool that saves you from typing a lot of keystrokes, you will start loving it. You will love it so much that you will look for opportunities to use it as much as possible. Let me show you how cool it is.

Imagine that you are working on an application for a car shop. You have a list of cars and you need to filter that list by various criteria. You may need to filter it

by company, you may need to filter it by price, or by any other property that the users may want. The following code shows how one might do it. You should actually copy all of the following code in a single file named `TestClass.java` so that you can try it out. Just add `import java.util.*;` at the top.

```
class Car

{
    String company; int year; double price; String type;
    Car(String c, int y, double p, String t){
        this.company = c; this.year = y;
        this.price = p; this.type = t;
    }
    public String toString(){ return "("+company+" "+year+")"; }
}
```

The `Car` class represents a car with a few properties. Agreed, `Car` is not well encapsulated. Ideally, `Car` should have had private fields and public accessors. But since encapsulation is not relevant here, I haven't shown these methods to conserve space.

```
class CarMall

{
    List<Car> cars = new ArrayList<>();

    CarMall(){
        cars.add(new Car("Honda", 2012, 9000.0, "HATCH"));
        cars.add(new Car("Honda", 2018, 17000.0, "SEDA
N"));
        cars.add(new Car("Toyota", 2014, 19000.0, "SUV
"));
        cars.add(new Car("Ford", 2014, 13000.0, "SPORT
"));
```

```

        S"));
        cars.add(new Car("Nissan", 2017, 8000.0, "SUV")
));
}
}

List<Car> showCars(CarFilter
cf){
    ArrayList<Car> carsToShow = new ArrayList<>();
    for(Car c : cars){
        if(cf.showCar(c)) carsToShow.add(c);
    }
    return carsToShow;
}
}

interface CarFilter
{
    boolean showCar(Car c);
}

```

`CarMall` represents the shop. It creates a list of a few `Car` objects. This list contains details of all the cars that the shop has. It has a `showCars` method that returns a list of cars based on any given criteria. Instead of specifying the actual criteria for filtering the cars inside the `showCars` method, it uses a `CarFilter` instance to determine whether a car needs to be listed or not. The `CarFilter` interface declares just the basic structure of a filter. Note that it is not really possible to code the actual filtering criteria inside the `showCars` method because the criterion is determined by the user of the `CarMall` class. By accepting an interface as an argument, the `showCars` method lets the caller decide the criterion.

```

public class TestClass{
    public static void main(String[] args) {
        CarMall cm = new CarMall();
    }
}

```

```

        CarFilter cf = new CompanyFilter("Honda");

        List<Car> carsByCompany = cm.showCars(cf);
        System.out.println(carsByCompany);
    }
}

class CompanyFilter implements CarFilter

{
    private String company;
    public CompanyFilter(String c){
        this.company = c;
    }
    public boolean showCar(Car c){
        return company.equals(c.company);
    }
}

```

TestClass represents a third party class that uses **CarMall**. It wants to get the details of all cars from a particular company, say, **Honda**. To do that, it defines a **CompanyFilter** class that contains the actual logic for filtering cars based on company name. At run time, it creates a **CompanyFilter** object and passes it to **CarMall**'s **showCars** method, which returns a filtered list of cars.

Now look at the following code for **TestClass** that uses a lambda expression:

```

public class TestClass{
    public static void main(String[] args) {
        CarMall cm = new CarMall();
        List<Car> carsByCompany = cm.showCars(c -> c.c
ompany.equals("Honda"))

    };
        System.out.println(carsByCompany);
    }
}

```

Observe that there is no separate class that implements `CarFilter` and there is no explicit instantiation of a `CarFilter` object either. Both of these tasks have been replaced by a very short statement - `c -> c.company.equals("Honda")`. That's it. We have actually eliminated 10 lines of code with that change! Go ahead, count them :)

As I said before, the lambda expression used above is just a shortcut for the compiler. The compiler actually expands this expression into a fully-fledged class definition plus code for instantiating an object of that class. Once you understand how the compiler is able to do this expansion, lambda expressions will seem like a piece of cake to you.

In fact, all the information that is required to do the expansion is available in the context of `cm.showCars(...)` method call already. The compiler knows that it must pass an object of a class that implements `CarFilter` to the `showCars(CarFilter cf)` method. It knows that this class must implement the `boolean showCar(Car c)` method because that is the only abstract method declared in the `CarFilter` interface. The compiler gathers from the signature of this method that this method receives an argument of type `Car` and returns a `boolean`. From this information, it can easily generate the following code on its own:

//This class is created by the compiler. It can give any random name to the class here!

```
class XYZ implements CarFilter{
    public boolean showCar(Car <<parameterName>>
){
    return <<an expression that returns a boolean must appear here>>
;
}
}
```

The only thing that the compiler cannot generate on its own is the boolean expression that goes inside the `showCar` method. It is not a coincidence that that is exactly what our lambda expression `c -> c.company.equals("Honda")` contains. The compiler simply takes the variable name from the left-hand side of `->` of the lambda expression, i.e., `c`, plugs in the expression given on the right-hand side of `->` i.e. `c.company.equals("Honda")`, into the body of the above method and it has the complete code for a class that implements `CarFilter`! Finally, it throws in an instantiation expression `new XYZ();` as a bonus! These are the same two things that are needed to invoke `cm.showCars(CarFilter cf)` method, i.e., code for a class that implements `CarFilter` and an expression that instantiates an object of that class!

I suggest you go through the above discussion a couple of times to absorb the steps that the compiler takes for expanding a lambda expression into a fully-fledged class before moving forward. As an exercise, try to expand the lambda expression `x -> x.price > 10000` into a class that implements `CarFilter`.

Observe that the lambda expression does not specify the method name, the parameter types, and the return type. The compiler infers these three pieces of information from the context in which you put the lambda expression. Thus, a lambda expression must exist within a context that can supply all this information. Furthermore, you know that in Java, a method cannot exist on its own. It can exist only as a member of a class (or an interface, for that matter). This implies that the method generated by the compiler for a lambda expression cannot exist on its own either. The compiler must create a class as well to contain the method. Since a lambda expression has no name, the class must be such that it requires implementation of exactly one abstract method whose name the compiler can use for the generated method. The only way this can happen is if the class generated by the compiler implements an interface with exactly one abstract method or extends an abstract class with exactly one abstract method. If the interface or the class has more than one abstract method or no abstract method, the compiler wouldn't know which method the code in the lambda expression belongs to.

Java language designers decided not to allow lambda expressions for abstract classes to reduce complexity. Thus, the only question remaining is which

interface should the generated class implement. That depends on the context. The type that the context expects is described as the "**target type**" and is the interface that the class generated by the compiler must implement. In the above code, the `showCars` method expects a `CarFilter` object (i.e. object of a class that implements `CarFilter`). Therefore, the generated class must implement `CarFilter`. In technical terms, `CarFilter` is the target type of the lambda expression that is passed as an argument to `showCars` method.

You may wonder at this point why the compiler can't provide a made up name to the generated method just like it did for the generated class. Well, it could provide a made up name to the method but what would that achieve? Since the programmer wouldn't know that made up name, how would they call this method? Remember that the programmer doesn't need to know the name of the class because the compiler passes an object of this class and the programmer knows the name of the parent class or the interface. You have learnt in the Polymorphism section that a reference of type parent class/interface can be used to invoke a method on a subclass. That is exactly what is happening here. The receiver of the lambda expression doesn't care about the type of the actual object that it receives because the receiving code invokes a method on that object using an interface reference. Take a look at the `showCars` method of `CarMall` again. It uses a reference of type `CarFilter` to invoke the `showCar` method. It doesn't matter to this code what name the compiler gives to the generated class. This code only cares about the name of the method.

From the above discussion, it should be clear that a lambda expression can be written only where the target type is an interface with exactly one abstract method. Java has a special name for such an interface: **Functional Interface**. I will get to Functional interfaces soon, but first, you need to know the various ways in which you can write a lambda expression.

14.1.2 Parts of a Lambda expression

You have seen that a lambda expression is basically code for a method in compact form. It has two parts separated by the "arrow" operator, i.e., `->`. The left side is for variable declarations and the right side is for the code that you

want executed. Just like a method, a lambda expression also can have any number of parameters and can return (or not return) a value. Java allows lambda expressions to be written in several different ways. The reason for having so many ways is to cut out as much redundant code as possible. The exam expects you to know all these ways. You will be asked to identify valid and invalid lambda expressions in the exam. From this perspective, I have categorized the variations into two categories - variations on the parameter section and variations on the body section.

There are three possibilities for the parameters section:

1. **No parameter** - If a lambda expression takes no parameters, the parameter part of the expression must have an empty set of brackets, i.e., (). For example,

```
( ) -> true //valid  
-> 1 //invalid, missing variable declaration part
```

2. **One parameter** - If a lambda expression takes exactly one parameter, the parameter name may be specified within brackets, i.e., (pName) or without the brackets, i.e., pName. For example,

```
a -> a*a //valid
```

```
(a) -> a*a //valid
```

You may also include the parameter type for the parameter name if you want but then you will need to use brackets. For example:

```
(int a) -> a*a //valid
```

```
int a -> a*a //invalid
```

3. **More than one parameters** - If a lambda expression takes more than one parameter, all the parameter names must be specified within the brackets, i.e., (pName1, pName2, pName3). For example,

```
(a, b, c) -> a + b + c //valid
```

a, b -> a+b //invalid, parameters must be within ()

Again, parameter types are optional. For example.

```
(int a, int b, int c) -> a + b + c //valid
```

If you are specifying parameter types, you must specify them for all the parameters. Thus, (int a, int b, c) -> a + b + c would be invalid because it does not specify the type of c

Java 11 has added one more way to declare the parameters. You can use the var type for declaring the parameters like this:

(var a) -> a*a //valid. Observe the brackets, they are required if you are using var.

Normally, there is no need to specify the type for the lambda parameters. However, it is not possible to apply annotations on the parameters unless you specify the type. For example, the following will not compile because annotations (a topic not on the part 1 exam) cannot be specified directly on a variable: (@NotNull a) -> a*a .

To be able to apply an annotation, you need to have a type for the variable, like

this: (@NotNull Integer a) -> a*a . So, if you want to use an annotation on a lambda parameter but don't want to specify the type, you can use the var declaration, like this: (@NotNull var a) -> a*a .

The syntax of the code part of a lambda expression is simple. It can either be an expression or a block of code contained within curly braces. Given that the body may or may not return a value, there are four possibilities:

1. **Expression with or without a return value** - This is the most common use case and is therefore, the smallest. You can simply put an expression on the right side of -> . If the expression has a return value and if the lambda expression is supposed to return a value, the compiler will figure out that the value generated by the expression is to be returned and will insert a return statement on its own. You must not write the return keyword. For example,

```
a -> a + 2 //valid  
a -> return a + 2 //invalid, must not have return  
keyword
```

Similarly, an expression that doesn't return any value can also be used directly as the body of the lambda expression. For example,

```
(a, b) -> System.out.println(a+b)  
//method call is a valid expression
```

2. **Block of code with or without a return value** - If you have multiple lines of code, you must write them within curly braces, i.e., { }. If the expression is supposed to return a value, you must use a return statement to return the desired value.

This is pretty much the same as writing a method body with or without a return value. You can use this syntax even if you have just one statement in the body. For example, here is Lambda expression that returns a value:

```
(a) -> {  
    int x = 2;  
    int y = x+a;  
    return y;  
}
```

and here is one that doesn't:

```
() -> {  
    int x = 2;  
    int y = 3;  
    System.out.println(x+y);  
}
```

Observe that unlike lambdas with just an expression as their bodies, the statements within the block end with a semi-colon. This is just like a regular code block. All the rules that apply to code within a method body apply to code within a lambda expression's code block as well. After all, the compiler uses this code block to generate a method body.

The OCP Java 11 Part 1 exam does not try to trick you with complicated lambda expressions. If you learn the basic rules shown above, you will not have any trouble identifying valid and invalid lambda expressions.

14.1.3 Using Predicate [🔗](#)

Let's take a look at the `CarFilter` interface that we defined in our `CarMall` example again:

```
interface CarFilter{  
    boolean showCar(Car c);  
}
```

The whole purpose of this interface is to let you check whether a `Car` satisfies a given criteria so that you could filter a list of Cars. Filtering through a list of

objects is such a common requirement in Java application that the Java standard library includes a generic interface for this purpose - `java.util.function.Predicate`. It looks like this:

```
interface Predicate<T>{
    boolean test(T t);
}
```

The `<T>` part means that this interface can be typed for any class as explained in the Generics section of the previous chapter. If the name of that class a `Predicate` is typed to is `T`, then the method `test` will accept an object of type `T` and return a `boolean`.

Let's change the code for `CarMall`'s `showCars` method to use `Predicate` interface:

```
List<Car> showCars(Predicate<Car> cp){
    ArrayList<Car> carsToShow = new ArrayList<>();
    for(Car c : cars){
        if(cp.test(c)) carsToShow.add(c);
    }
    return carsToShow;
}
```

Observe that we have typed `Predicate` to `Car` in the above code. Apart from that, the above code is the same as previous one. But by using the `Predicate` interface instead of writing a custom interface, we have eliminated another three lines of code.

There is no change in the code that calls `showCars` method. The lambda expression that we used earlier, i.e., `cm.showCars(c -> c.company.equals("Honda"))` works for this new method as well. It works because the lambda expression never required us to use the name of any interface or method. Therefore, the lambda expression was not tied to a particular interface or method. It was only tied to a particular behavior -, i.e., to a method that takes `Car` as an argument and returns a `boolean`. We relied on the compiler to produce an appropriate class with an appropriate method. We

supplied only the raw code for the method body. Earlier the compiler generated a class that implemented the `CarFilter` interface and it now generates a class that implements the `Predicate<Car>` interface using the same code that we wrote in the lambda expression! In fact, the change is so subtle that if you have both the versions of `showCars` method in `CarMall`, the compiler will reject the line `cm.showCars(c -> c.company.equals("Honda"))` with the error message, "reference to showCars is ambiguous. Both method `showCars(java.util.function.Predicate<Car>)` in `CarMall` and method `showCars(CarFilter)` in `CarMall` match".

I showed only one method in `Predicate` interface but it actually has three `default` methods and one `static` method in addition to the abstract `test` method. I didn't mention them before because they have nothing to do with lambda expressions. You will notice that these methods are basically just helpful utility methods. We haven't ever seen anyone getting any question on these but, it is better to be aware of these methods.

1. `default Predicate<T> and(Predicate<? super T> other)` : Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. This is helpful when you have more than one checks to be performed. For example, the following code checks whether a `Car` satisfies two predicates using two separate invocations of `test()` :

```
Predicate<Car> p1 = c -> c.company.equals("Honda");
;
Predicate<Car> p2 = c -> c.price > (20000.0);

Car c = ...
if(p1.test(c) && p2.test(c)
    ) System.out.println("yes");
```

Instead of making two calls to `test()`, you could combine the two predicates into one and use only one call to test, like this:

```

Predicate<Car> p3 = p1.and(p2)

;

Car c = ...
if( p3.test(c)

) System.out.println("yes");

```

You do not need to worry about the `? super T` part. It is related to "bounded types" in generics, which is not on the part 1 exam.

2. `default Predicate<T> negate()` : Returns a predicate that represents the logical negation of this predicate. For example, if you have `Predicate<Car> p = c -> c.price < 20000`; then `Predicate<Car> notP = p.negate()`; represents a predicate where `price is >= 20000`.
3. `default Predicate<T> or(Predicate<? super T> other)` : Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. For example, if you have `Predicate<Car> isHonda = c -> c.company.equals("Honda")`; and `Predicate<Car> isToyota = c -> c.company.equals("Toyota")`; then `Predicate<Car> isHondaOrToyota = isHonda().or(isToyota)`; represents a predicate where the company name is "Honda" or "Toyota".
4. `static <T> Predicate<T> isEqual(Object targetRef)` : Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`. This method is a way to convert a class's equals method into a `Predicate`. For example, normally, you would compare two Car objects using `c1.equals(c2)`. You could create a `Predicate` out of the equals method like this - `Predicate equals = Predicate.isEqual(c1);` . and then compare `c1` with other `Car` objects using this `Predicate`, i.e., `equals.test(c2)` .

14.1.4 Functional Interfaces

The **Predicate** interface that you saw just now is one among several interfaces defined in **java.util.function** package. The Predicate interface captures the use case for testing a condition. In the same way, Java designers have identified several use cases that are encountered in day to day Java development and have created readymade interfaces for them so that every project doesn't have to define the same kind of interfaces again and again. These interfaces belong to a category of interfaces known as "Functional interfaces". Don't be alarmed by the fancy sounding name. Any interface that has exactly one abstract method is a functional interface. It may have other static or default methods but it must have one and only one abstract method. This abstract method is also called the "functional method" of that functional interface. For example, the functional method of the **Predicate** interface is **boolean test(T t)**. Having exactly one abstract method is an important property because it allows you to implement such interfaces using Lambda expressions.

The reason they are called functional interfaces is that they represent a single function without maintaining any state. A functional interface is meant to do exactly one thing. For example, if you take a look at the Predicate interface, you will notice that it is meant to check a condition. But the logic for the actual condition is not there in the functional method (because it is an abstract method). The logic is provided by the code that implements the interface, for example, through a lambda expression. The benefit of this approach is that it allows you to separate the logic from the data upon which that logic is to be applied. That is why it is called "functional programming". Observe that this is exactly opposite to the approach adopted by Object-Oriented Programming, in which an object contains data as well as the logic to operate on that data.

OCP Java 11 Part 1 exam does not mention the topic of Functional interfaces explicitly but it contains questions that refer to a few functional interfaces defined in the **java.util.function** package, namely, **Predicate**, **Consumer**, and **Supplier**. The exam expects you to know the basic usage of these interfaces without going into the details too much. You have already seen the usage of the **Predicate** interface. I will talk about the rest next.

java.util.function.Consumer ↗

A consumer is meant to "consume" an object. By consume, we mean that we want to do something with the given object. The functional method of the **Consumer** interface is named **accept(T t)**. You just need to write the code for what you want to do with the object passed in this method. For example:

```
//creating a Consumer that consumes a String.
```

```
Consumer<String> strConsumer = s -> System.out.println(s.length());
strConsumer.accept("hello"); //prints 5
```

In the above code, the lambda expression contains only the logic for consuming the String argument. It doesn't contain the actual data on which this logic is to be applied. The actual data comes in the next statement.

java.util.function.Supplier ↗

A **Supplier** supplies an object whenever invoked. Its functional method is **T get()**. Here is an example that illustrates its usage:

```
public List<Car> getCars(Supplier<Car> carSupplier){
    List<Car> cars = new ArrayList<>();
    for(int i=0; i<10; i++) cars.add(carSupplier.get());
    return cars;
}
```

The above method could be invoked like this:

```
List<Car> cars = getCars(() -> new Car()); //assuming Car has a default constructor.
```

14.1.5 Using Functional Interfaces with Collections API [🔗](#)

Generics made Java collections type safe, while Lambda expressions together with functional interfaces gave them super powers. Most of the classes and interfaces of the collections API were updated in Java 8 to include methods that accept functional interfaces so that programmers can invoke those methods with lambda expressions. Things such as iterating, filtering, and replacing elements, that took several lines of code can now be done by half a line of code.

Since the Part 1 exam explicitly mentions List and ArrayList, let's see a few examples of how functional interfaces and Lists are used together.

The forEach method [🔗](#)

As you have seen before, iterating through a collection is very common requirement. Prior to Java 8, a common way to iterate through a collection was to use a regular for loop. The **Collection** interface actually extends **java.util.Iterable** interface and so, it was also possible to use the **enhanced for** loop (aka the **for-each** loop) for this purpose.

With Java 8, a default method named **forEach(Consumer<E> consumer)** was added to the **Iterable** interface, which made it an absolute delight to iterate through the elements of any collection. The following snippet shows the old and the new way to do the same:

```
List<String> list = List.of("a", "b", "c");  
//old way
```

```
for(String s : list){  
    System.out.println(s);  
}  
  
//new way  
  
list.forEach(s->System.out.println(s));
```

The new way has pretty much taken over completely because of its simplicity.

Similarly, the **Map** interface defines a default **forEach** method that takes a **BiConsumer** instead of **Consumer**. A **BiConsumer** is similar to **Consumer** except that it take two arguments instead of one. Its functional method is **void accept(T, U)**. Here is an example of how it is used to process the elements of a Map:

```
BiConsumer<String, Integer> bc = (s, i) -> System.out.  
println(s+" is mapped to "+i);  
Map<String, Integer> map = new HashMap<>();  
map.put("One", 1);map.put("Two", 2);  
map.forEach(bc);  
  
//you could use the lambda expression directly as an argument  
  
map.forEach((s, i) -> System.out.println(s+" is mapped  
to "+i));
```

You should get familiar with the usage of the **forEach** method because you will see it used on the exam. There are no trick question in the exam on this method though.

The removeIf method [↳](#)

Another common operation performed with collections is filtering. The `Collection` interface has a default `removeIf(Predicate<? super E> filter)` method for this purpose. This method removes all of the elements of this list that satisfy the given predicate. Here is a simple example:

```
List<Integer> iList = List.of(1, 2, 3, 4, 5, 6);
Predicate<Integer> p = x->x%2==0;
iList.removeIf(p);
System.out.println(iList);
```

The above code prints `[1, 3, 5]`. As you have probably guessed, the lambda expression returns `true` if an element is even. The `removeIf` method executes this lambda expression for each of the elements in the list and removes the element if the expression returns `true` for that element.

Again, we didn't really have to use the variable `p` in the above code. We could have passed the lambda expression directly to the `removeIf` method, i.e.,
`iList.removeIf(x->x%2==0);`

The sort method [↳](#)

A collection has no notion of order but a list does. It makes sense, therefore, that `List` interface has a default `sort(Comparator<? super E> comparator)` method which allows you to sort the elements of this list using the sorting order determined by the comparator. The `java.util.Comparator` interface has been around since Java 1.2 but it has been made a functional interface in Java 8. Its functional method is `int compare(T o1, T o2)`, which compares its two arguments and returns a negative integer, zero, or a positive integer if the first argument is less than, equal to, or greater than the second. This helps us sort a list in a single line of code as the following example shows:

```
List<String> games = new ArrayList<>(List.of("football",
", "cricket", "baseball", "tennis"));
games.sort( (a, b)->a.compareTo(b));
```

```
games.forEach(s->System.out.println(s));
```

The output is:

```
baseball  
cricket  
football  
tennis
```

If you want to sort the list in the reverse order, you can change the lambda expression to, `(a, b)->-a.compareTo(b)` . Observe the minus sign in front of `a.compareTo(b)` .

None of the methods above are mentioned explicitly in the official exam objectives. However, we have seen questions on the exam that refer to these methods. The questions are not tricky and you should be able to answer them with the information given above.

14.1.6 Scope of variables in a lambda expression

There is just one more thing that you need to know about lambda expressions. The variables that you define in the variable section of a lambda expression exist in the same scope as which the lambda expression itself exists. This means, you cannot redefine the variables that already exist in that scope. This is illustrated in the following code:

```
List<String> names = Arrays.asList(new String[]{"alex", "bob", "casy", "abel"});  
for(String n : names){  
    Predicate p = n->n.startsWith("a"); //will not compile
```

```

        if(p.test(n))  {
            System.out.println(n);
        }
}

```

The above code will fail to compile with an error message that says, "**variable n is already defined**". This is because **n** is already in scope within the for loop as well as in the lambda expression. So, when the lambda expression tries to define **n**, the compiler complains.

It is possible to access a variable that is in scope of the lambda expression from within the lambda expression's body but only if that variable is declared as final or is "effectively final". Effectively final means that even though the variable is not explicitly declared as final, its value is not changed throughout the scope in which it exists, and so, the compiler assumes it as final. Here is an example:

```

List<String> names = Arrays.asList(new String[]{"alex",
    "bob", "casy", "abel"});
int x = 0;
for(String n : names){
    Predicate<String> p = k->{
        System.out.println(n); //valid
        //System.out.println(x); //will not compile

        return k.startsWith("a");
    };
    if(p.test(n))  {
        System.out.println(n);
    }
}
x = 1; //x is being changed here

```

In the above code, the variable **n** is never changed after it is assigned a value.

Thus, **n** is effectively final. But **x** is not because it is assigned a new value later in the code.

14.1.7 Quiz

Q. Given the following class:

```
public class Item{  
    String name; double price;  
    public Item(String name, double price){  
        this.name = name; this.price = price;  
    }  
}
```

and the following code appearing in another class

```
double price = 100.0;  
Supplier<Item> itemSupplier = ()-> new Item("dummy", p  
rice);  
price = 200.0;  
System.out.println(itemSupplier.get().price);
```

What will be the output?

- A.** 100.0
- B.** 200.0
- C.** Nothing is printed.
- D.** Compilation failure.
- E.** An exception is thrown at run time.

Correct answer is D.

Observe that the variable `price` is not effectively final because its value is being changed in the code. Therefore, it cannot be used in the lambda expression.

14.2 Exercise

- Given the following lambda expressions, define appropriate interfaces that can be implemented using these lambda expressions.

```
( ) -> true  
k -> k>5
```

- Given the following interfaces, create lambda expressions that can be used to capture these interfaces.

```
interface Shape{  
    double computeArea();  
}  
  
interface Operation{  
    void operate(String name, double[] params);  
}
```

- Write a method that takes a list of `Image` objects and a `Predicate` as arguments, and returns another list containing only those Images that satisfy the predicate.
- Assuming that the `Image` class has `width` and `height` properties, invoke the above method that filters out images that are smaller than 100 x 100.

Chapter 15 Handling Exceptions

- Describe the advantages of Exception handling and differentiate among checked, unchecked exceptions, and Errors
- Create try-catch blocks and determine how exceptions alter program flow
- Create and invoke a method that throws an exception

15.1 Create try-catch blocks and determine how exceptions alter program flow

15.1.1 Java exception handling [↳](#)

Exceptions are for managing exceptional situations. For a file copy program, the normal course of action could be - open file A, read the contents of file A, create file B, and write the contents to file B. But what if the program is not able to open file A? What if the program is not able to create or write to file B? There could be many reasons for such failure such as a typo in the file name, lack of permission, no space on disk, or even disk failure. Now, you don't expect these problems to occur all the time but it is reasonable to expect them some times. Since we don't expect these problems to occur regularly in normal course of operation, we call them "exceptional".

In exceptional situations, you may want to give the user a feedback about the error or you may even want to take an alternative course of action. For example, you may want to let the user to input another source file name if the input file is not found or another target location if the given location is out of space. Even if you don't want to take any special action upon such situations, you should at least want your program to end gracefully instead of crashing unexpectedly at run time. This means, you should provide a path for the program to take in such situations. One way would be to check for each situation before proceeding to copy. Something like this:

```
if(checkFileAccess(file1)){
    if(checkWritePermission(targetDirectory){
        //code for normal course of action

    }else{
        System.out.println("Unable to create file2");
    }
}else{
    System.out.println("Unable to read file1");
}
```

You can see where this is going. You will end up having a lot of if-else statements. Not only is it cumbersome to code, it will be a nightmare to read and maintain later on.

There is another serious problem with the above approach. It doesn't provide way to write code for situations that you don't even know about at the time of writing the code. For example, what if the user runs this code in an environment that requires file names to follow a particular format? So, now, you have two kinds of "exceptional situations". One that you know about at the time of writing the code, and one that you don't know anything about.

Java exception mechanism is designed to help you write code that covers all possible execution paths that a program may take - 1. path for normal operation 2. paths for known exceptional situations, and 3. path for unknown exceptional situations. Here is how the above mentioned program can be written:

```
try{
    //code for normal course of action

}catch(SecurityException se){
    //code for known exceptional situation
```

```
        System.out.println("No permission!");
    }
catch(Throwable t){
    //code for unknown exceptional situations

    System.out.println("Some problem in copying: "+t.get
Message());
    t.printStackTrace();
}
```

Observe that the perspective is reversed here. In the if/else approach, you check for each exceptional situation and proceed to copy if everything is good, while in the try/catch approach, you assume everything is good and only if you encounter a problem, you decide what to do depending on the problem. The benefits of the try/catch approach are obvious. It provides a clean separation between code for normal execution and code for exceptional situations, which makes the code easier to read and maintain. It allows an alternative path to the program to proceed even in cases where the programmer has not anticipated the problem.

Exception handling in Java is not a bag of all goodies. There has been a good amount of debate on what constitutes "exceptional situations" and what should be the right approach to handle such situations. My objective in this chapter is to teach you Java's approach to exception handling and so, I will not go into an academic discussion on whether it is good or bad as compared to other languages. However, it is a very important topic of discussion in technical interviews. I suggest you google criticism of Java's exception handling and compare it with C#'s.

15.1.2 Fundamentals of the try/catch approach [🔗](#)

When you think of developing code as developing a component, you will realize that there are always two stakeholders involved - the provider/developer of the

component and the client or the user of the component. For the client to be able to use the component, it is imperative for the provider to tell the client about "how" his component works. The how not only includes the input/output details of the component but also the information about exceptional situations, i.e., about situations the component knows may occur but does not deal with.

For example, let's say you are developing a method that copies the contents of one file to another and that this method is to be used by a developer in another team. You must convey to the other developer that you are aware of the Input/Output issues associated with reading and writing a file but you won't do anything about them. In other words, you must convey that your method will try its best to copy the file but if there is an I/O issue, it will abandon the attempt and let her know about the failure so that she can deal with it however she wants to. This is done through the use of a "**throws**" clause in method declaration:

```
public void copyFile(String inputPath, String outputPath) throws IOException
{
    //code to copy file
}
```

In the above code, if the **copyFile** method encounters any I/O issue while copying a file, it will simply abort the copying and throw an **IOException** to the caller. If it does not encounter any I/O issue, the method will end successfully.

On the other side of the component is the user of that component, who uses the information provided by the component provider to develop her code. The user has to decide how she wants to handle the exceptional situation. If she believes that she has the ability to "resolve" the situation, she will put the usage of that component in a "**try**" block with an associated "**catch**" block that contains code for "resolution" of the problem. When that exceptional situation actually occurs, the control goes to the catch block instead of proceeding to the next statement after the method call that threw the exception.

If the user decides that she cannot handle the exceptional situation either, she propagates the exception to the caller of her component.

For example, a program that creates backup of a file may use the file copy program internally to copy a file. If the file copy program throws an **IOException**, the backup creator program may catch that exception and show a message to the user.

```
public void createBackup(String input) {  
    String output = input+".backup";  
    try{  
        copyFile(input, output);  
        System.out.println("backup successful");  
    }catch(IOException ioe  
){  
    System.out.println("backup failure");  
}  
}
```

If the **copyFile** method completes without any issue, the control will go to the **println** statement. If the **copyFile** method throws an **IOException**, the control will go to the catch block, thus, providing an opportunity to take a different course of action. Here, the code may show a failure message to the user. From the perspective of the developer of the backup program, showing the error message to the user is the resolution of the I/O problem. One can also write code to have the user specify another file or directory to create a backup.

An exception is considered "handled" when it is caught in a catch block. If the backup creator method doesn't know what to do in case the **copyFile** is not successful, then it should let the exception propagate to its caller by using a **throws** clause of its own:

```
public void createBackup(String input) throws IOException  
{  
    String output = input+".backup";
```

```
        copyFile(input, output);
        System.out.println("backup successful");
    }
```

In the above code, if the `copyFile` method completes without any issue, the control will go to the `println` statement. But if the `copyFile` method throws an `IOException`, the `createBackup` method will end immediately and the caller will receive an `IOException`. Note that this will be the same `IOException` that it receives from the `copyFile` method. This is how an exception propagates from one method to the other. There could be a long chain of method calls through which an exception bubbles up before it is handled. If an exception is not caught anywhere while it is bubbling up the call chain, it ultimately reaches the JVM code. Since the JVM has no idea about the business logic of the program that it is executing, it "handles" the exception by killing(aka terminating) the program (actually, the JVM kills only the thread that was used to invoke the chain of method calls but threading is not on the OCP Java 11 Part 1 exam, so you can assume that a program is composed of only one thread and killing of that thread is the same as killing the program).

15.1.3 Pieces of the exception handling puzzle [🔗](#)

Java exception handling mechanism comprises several moving parts. What makes this mechanism a bit complicated is that you need to put each part in its right place to make the whole thing work. So let me introduce these parts first briefly.

The `java.lang.Throwable` object - [🔗](#)

`Throwable` is the root class of all exceptions. It captures the details of the program and its surroundings at the time it is created. Among other things, a `Throwable` object includes the chain of the method calls that led to the exception (known as the "stack trace") and any informational message specified by the programmer while creating that exception. This information is helpful in determining the location and the cause of the exception while debugging a program. You may have seen crazy looking output on the console containing

method names upon a program crash. This output is actually the stack trace contained in the **Throwable** object.

Throwable has two subclasses - **java.lang.Error** and **java.lang.Exception**, and a huge number of grand child classes. Each class is meant for a specific situation. For example, a **java.lang.NullPointerException** is thrown when the code tries to access a null reference or an **java.lang.ArrayIndexOutOfBoundsException** is thrown when you try to access an array beyond its size. You can also create your own subclasses by extending any of these classes if the existing classes don't represent the exceptional situation appropriately. For example, an accounting application could define a **LowBalanceException** for a situation where more money is being withdrawn from an account than what the account has.

What is called "**exception**" (i.e. exception with a lower case 'e') in common parlance, is in reality is an object of class **java.lang.Throwable** or one of its subclasses.

One subclass of **java.lang.Exception** that is particularly important is **java.lang.RuntimeException**. **RuntimeException** and its subclasses belong to a category of exception classes called "**unchecked exceptions**". You will see their significance soon.

The **throw** statement

The **throw** statement is used by a programmer to "throw an exception" or "raise an exception" explicitly. A programmer may decide to throw an exception upon encountering a condition that makes continuing the execution of the code futile. For example, if, while executing a method, you find that the value of a required parameter is invalid, you may throw an **IllegalArgumentException** using a **throw** statement like this:

```
public double computeSimpleInterest(double p, double r  
, double t){  
    if( t<0) {  
        IllegalArgumentException iae = new IllegalArgumentException("time is less than 0");  
        throw iae;  
    }  
    return (p * r * t) / 100;  
}
```

```
    throw iae;  
  
}  
//other code  
  
}
```

Usually, an exception object is created only to be "thrown" and so there is no need to store its reference in a variable. That is why it is often created and thrown in the same statement as shown below:

```
public double computeSimpleInterest(double p, double r  
, double t){  
    if( t<0) throw new IllegalArgumentException("time is  
less than 0");  
  
    //other code  
  
}
```

Throwing an exception implies that the code has encountered an unexpected situation with which it does not want to deal. The code shown above, for example, expects the time argument to be greater than zero. For this code, time being less than zero is an unexpected situation. It does not want to deal with this situation (probably because the programmer is not sure what to do in this case) and so it throws an exception in such a situation. This also means that this method passes on the responsibility of determining what should be done in case time is less than zero to the user of this method.

Note that only an instance of `Throwable` (or its subclasses) can be thrown using the `throw` statement. You cannot do something like `throw new Object();` or `throw "bad situation";`

Explicitly throwing an exception using the `throw` statement is not the only way

an exception can be thrown. JVM may also decide to throw an exception if the code tries to do some bad thing like calling a method on a null reference. For example:

```
public void printLength(String str){  
    System.out.println(str.length());  
}
```

If you pass `null` to the above method, the JVM will create a new instance of `NullPointerException` and throw that instance when it tries to execute `str.length()`.

The `throws` clause

Java requires that you list the exceptions that a method might throw in the `throws` clause of that method. This ties back to Java's design goal of letting the user know of the complete behavior of a method. It wants to make sure that if the method encounters an exceptional situation, then the method either deals with that situation itself or lets the caller know about that situation. The `throws` clause is used for this purpose. It conveys to the user of a method that this method may throw the exception mentioned in the `throws` clause. For example:

```
public double computeSimpleInterest(double p, double r  
, double t) throws Exception  
{  
    if( t<0) throw new Exception("time is less than 0");  
    //other code  
  
}
```

Now, anyone who uses the above method knows that this method may throw an exception instead of returning the interest. This helps the user write appropriate code to deal with the exceptional situation if it arises.

The **try** statement [↳](#)

A **try statement** gives the programmer an opportunity to recover from and/or salvage an exceptional situation that may arise while executing a block of code. A try statement consists of a **try block** , zero or more **catch blocks** , and an optional **finally block** . Its syntax is as follows:

```
try {  
    //code that might throw exceptions  
  
} catch(<ExceptionClass1> e1){  
    //code to execute if code in try throws exception 1  
  
}  
catch(<ExceptionClass2> e2){  
    //code to execute if code in try throws exception 2  
  
}  
catch(<ExceptionClassN> en){  
    //code to execute if code in try throws exception N  
  
}  
finally{  
    //code to execute after the try block and catch bloc  
    k finish execution  
  
}
```

Note that curly braces for the **try** , **catch** , and **finally** blocks are required even if there is a single statement in these blocks (compare that to **if** , **while** , **do/while** and **for** blocks where curly braces are not required if there is only

one statement in the block). Furthermore, a **try block** must follow with at least one **catch block** or the **finally block**. A try block that follows with neither a catch block nor a finally block will not compile.

The **try** statement is the counterpart of the throw statement because putting a piece of code within a try block signifies that the programmer wants do something in case that piece of code throws an exception. In other words, a try block lets the programmer deal with an exceptional situation as opposed to the throw statement, which lets the programmer avoid dealing with it.

While a try block contains code for normal operation of the program, a **catch block** is the location where the programmer tries to recover from an exceptional situation that arises in the try block. If the code in the try block throws an exception, the normal flow of execution in that try block is aborted (i.e. no further code in the try block is executed) and the control goes to the catch block. Here, the programmer can take alternate approach to finish the processing of the method. For example, the programmer may decide to just show a popup message to the user about the exception and move on to the next statement after the try statement.

A catch block is associated with a **catch clause**, which specifies the exception that the catch block is meant to handle. For example, a catch block with the catch clause as `catch(IllegalArgumentException e)` is meant to handle `IllegalArgumentException` and its subclasses. Thus, this catch block will be executed only if the code in the try block throws an `IllegalArgumentException` or its subclass exception. You can specify any valid exception class (including `java.lang.Throwable`) in the catch clause.

A **finally block** is the location where the programmer tries to salvage the situation or control the damage so to say, without attempting to recover from an exception. For example, a program that tries to copy a file may want to close any open files irrespective of whether the copy operation is successful or not. The programmer can do this in the finally block. You may think of a finally block as the step where a car mechanic reassembles the parts back irrespective of whether he was able to fix the car or not!

Here is a method that calls the `computeSimpleInterest` method shown

above within a try statement:

```
public void doInterest(){

    try{
        double interest = computeSimpleInterest(1000.0,
10.0, -1);
        System.out.println("Computed interest "+interest
);
    }catch(Exception e){
        System.out.println("Problem in computing interes
t:"+e.getMessage());
        e.printStackTrace();
    }finally{
        System.out.println("all done");
    }

}
```

In the above code, the call to `computeSimpleInterest` throws an `IllegalArgumentException` because `t` is negative. Thus, the `println` statement after the method call is not executed. The exception is caught in the catch block because its catch clause, i.e., `catch(IllegalArgumentException)` matches with the exception that is thrown by the try block and the control goes to the first statement in this catch block. It prints the exception's message and the stack trace on the console. Finally, the control goes to the code in the finally block, where it prints "`all done`". If you omit the catch block, the control will go directly to the finally block after the invocation of `computeSimpleInterest` method. After the execution of the code in the finally block, the caller of `doInterest` method will receive the same `IllegalArgumentException`.

Note that a finally block, if present, always executes irrespective of what happens in the try block or the catch block. Even if the try block throws an exception and there is no catch block to catch that exception, the JVM will execute the finally block. It will throw the exception to the caller only after the finally block finishes. The only case where the finally block is not executed is if the code in the try or the catch block forces the JVM to shut down using a call to

`System.exit()` method.

The following is a complete program that illustrates how an exception alters the normal program flow:

```
public class TestClass{

    public static void main(String[] args){
        TestClass tc = new TestClass();
        tc.doInterest();
    }

    public double computeSimpleInterest(double p, double r, double t){
        if( t<0) throw new IllegalArgumentException("time is less than 0");
        return p*r*t/100;
    }

    public void doInterest(){
        try{
            double interest = computeSimpleInterest(1000.0, 10.0, -1);
            System.out.println("Computed interest "+interest);
        }catch(IllegalArgumentException iae){
            System.out.println("Problem in computing interest:"+iae.getMessage());
            iae.printStackTrace();
        }finally{
            System.out.println("all done");
        }
    }
}
```

It generates the following output on the console :

```
Problem in computing interest:time is less than 0
java.lang.IllegalArgumentException: time is less than
0
        at TestClass.computeSimpleInterest(TestClass.j
ava:8)
        at TestClass.doInterest(TestClass.java:14)
        at TestClass.main(TestClass.java:4)
all done
```

You should observe the following points in the above code:

1. The `return` statement in `computeSimpleInterest` is not executed because the previous statement throws an exception.
2. The `println` statement in the try block of `doInterest` is not executed because the call to `computeSimpleInterest` ends with an exception instead of a return value. Control goes to the `catch` block directly after the method call.
3. The `catch` block prints the details captured in the exception object. It shows the sequence of the method invocations in reverse order from the point where the `IllegalArgumentException` object was created. You should try removing the catch block and see the output.
4. Once the `catch` block is finished, the control goes to the `finally` block.
5. The `doInterest` method returns after the execution of the `finally` block.
6. There is no throws clause in `computeSimpleInterest` method even though it throws an exception. The reason will be clear in the next section where I talk about checked and unchecked exceptions.

15.2 Differentiate among checked, unchecked exceptions, and Errors

15.2.1 Checked and Unchecked exceptions [🔗](#)

As discussed earlier, exceptions thrown by a method are part of the contract

between the method and the user of that method. If the user is not aware of the exceptions that a method might throw, she will be blindsided during run time because her code would not be prepared to handle that exception. The **throws** clause of a method is meant to list all exceptions that the method might throw. If an exception is listed in the throws clause, the user of the method will know that she needs to somehow handle that situation.

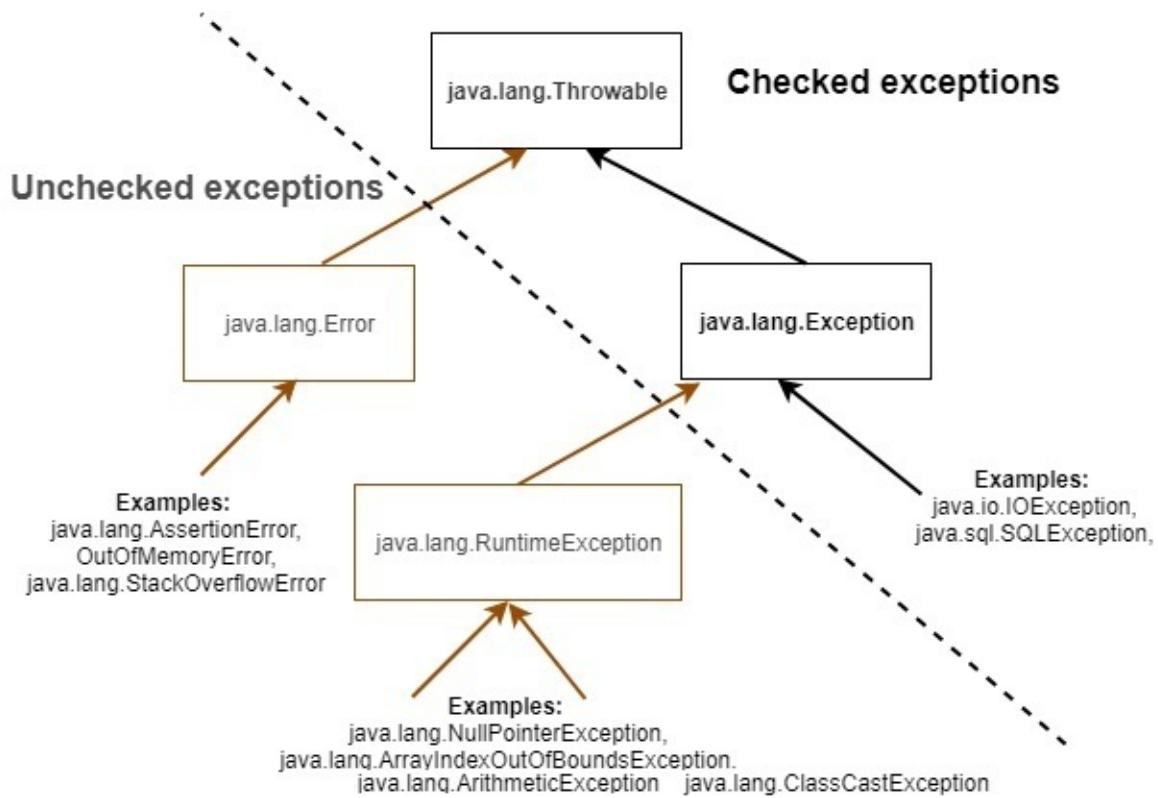
But what if a developer forgets to list an exception in the throws clause of a method? In that case, we are back to the same problem of blindsiding the user of that method at run time. This is where the compiler plays an important role. While compiling a method, the compiler checks for the possibility of any exception that might get thrown by the method to the caller. If that exception is not listed in the throws clause of that method, it refuses to compile the method.

This sounds like a good solution, but the problem here is that the compiler does not check for all kinds of exceptions thrown by a piece of code. It checks for only a certain kind of exceptions called "**checked exceptions**" and forces you to list the only those exceptions that belong to this category of exceptions in the throws clause.

Exceptions that do not belong to the category of checked exceptions are called "**unchecked exceptions**". They are called unchecked because the compiler doesn't care whether a piece of code throws such an exception or not. Listing unchecked exceptions in the throws clause is optional.

Finding out whether an exception is a checked exception or not is easy. Java language designers have postulated that any exception that extends `java.lang.Throwable` but does not extend `java.lang.RuntimeException` or `java.lang.Error` is a checked exception. The rest (i.e. `java.lang.Error`, `java.lang.RuntimeException`, and their subclasses) are unchecked.

The following figure illustrates this grouping of exceptions.



Note that Java has a deep rooted exception class hierarchy, which means there are several classes, subclasses, and subclasses of subclasses in the tree of exceptions. So just because an exception is a `RuntimeException`, does not mean that that exception directly extends `RuntimeException`. It could even be a grand child of `RuntimeException`. For example, `ArrayIndexOutOfBoundsException` actually extends `IndexOutOfBoundsException`, which in turn extends `RuntimeException`. Similarly, just because an exception is a checked exception does not mean that it directly extends `Exception`. It may extend any subclass of `Exception` (except `RuntimeException`, of course).

Rationale behind checked and unchecked exceptions [🔗](#)

Recall our discussion on exceptional situations where I talked about two kinds of exceptional situations - ones that a developer knows about and ones that a developer doesn't expect to occur at all. While a developer may want to provide an alternate path of execution in case of a situation that is known to occur but

there is no point in providing an alternate path of execution for a situation that is never expected to occur.

For example, if a piece of code tries to write to a file, the developer may want to take a different approach if he is not able to write to the file system. But if the data array that it is trying to write is null, there is nothing much he can do. Such unexpected situations occur mostly due to badly written code. In other words, if a code gets itself into an unexpected situation, it is most likely because of a programming error, i.e., a bug in the code. Such issues should be fixed during development itself. But if they do occur in production, they should rather be handled at a much higher level than at the component level.

Unchecked exceptions are for such **unexpected** situations. Java language designers believe that unchecked exceptions need not be declared in a method declaration because there is nothing to gain by forcing the caller to catch them. Only checked exceptions need to be declared because the caller of the method may have a plan to recover from them. There are two kinds of unchecked exceptions - exceptions that extend `java.lang.RuntimeException` (aka **runtime exceptions**) and exceptions that extend `java.lang.Error` (aka **errors**).

Runtime exceptions and errors [🔗](#)

Characterizing a situation as expected or unexpected is a design decision that depends on the business purpose of the method. One method may expect to receive a null argument and may work well if it gets a null argument, while another may not expect a null argument and may end up throwing a `NullPointerException` when it tries to access that null. In the second case, passing a null to that method would be considered a bug in the code, which must be identified and fixed during testing. Such exceptions that signify the presence of a bug in the code are categorized as **runtime exceptions**.

Here, the word runtime in `RuntimeException` does not imply that only exceptions that extend `RuntimeException` can be thrown at run time. All exceptions are thrown only when the program is executed, i.e., at run time. It refers to the fact that the developer comes to know of the occurrence of the situation that results in a `RuntimeException` only when the program is run, i.e.,

during run time. Had the programmer anticipated the occurrence of that situation during compile time, he would have fixed the code, and in which case the exception would not have been thrown upon running the program. For the same reason, runtime exceptions are usually not thrown explicitly using the `throw` statement. Indeed, why would you write the code to throw an exception if you don't even expect that situation to occur. The JVM throws runtime exceptions on its own when it encounters an unexpected situation. For example, if you try to access a null reference, the JVM will throw a `NullPointerException`, or if you try to access an array beyond its size, the JVM will throw an `ArrayIndexOutOfBoundsException`. It is possible to recover from runtime exceptions but ideally, since they indicate bugs in the code, you should not attempt to catch them and recover from them. A well written program should not cause the JVM to throw runtime exceptions.

The case of **errors** is similar to **runtime exceptions**. The difference is that errors are reserved for situations where the operation of the JVM itself is in jeopardy. For example, a badly written code may consume so much memory that there is no free memory left. Once that happens, the JVM may end up throwing `OutOfMemoryError`. Similarly, a bad recursion may cause `StackOverflowError` from which no recovery is possible. Errors signify serious issues in the interaction between the code and the JVM and are thrown exclusively by the JVM. It is never a good idea to throw them explicitly or to try to recover from them because the code will likely not work as expected anyway once the JVM starts throwing Errors.

Although the exam does not focus on the reason for categorizing exception between checked and unchecked exceptions, it is actually a very important topic to understand for a professional developer. You should also compare Java's exception handling with C#'s.

Identifying exceptions as checked or unchecked

Java follows a convention in naming exception classes. This convention is sometimes helpful in determining the kind of exception you are dealing with. The name of any class that extends `Error` ends with `Error` and the name of any class that extends `Exception` ends with `Exception`. For example,

`OutOfMemoryError` and `StackOverflowError` are Errors while `IOException`, `SecurityException`, `IndexOutOfBoundsException` are Exceptions.

However, there is no way to distinguish between unchecked exceptions that extend `RuntimeException` and checked exceptions just by looking at their names. It is therefore, important to memorize the names of a few important runtime exception classes, namely - `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmaticException`, `ClassCastException`, and `SecurityException`.

15.2.2 Commonly used exception classes [↑](#)

The Java standard library contains a huge number of exception classes but the exam expects you to know about only a few of them, which I will cover now. These exception classes are important because you will encounter them quite often while working with Java.

You may have read about the classification of exceptions based on whether they are thrown by the programmer or by the JVM. It is true that there are a few Exceptions that are thrown by the JVM on its own. However, there is no Exception that is thrown exclusively by the JVM. In fact, many methods of the Java standard library classes throw the same exceptions explicitly using the `throw` statement.

I have omitted the package name from the classes below for brevity but note that all of the following exception classes belong to the `java.lang` package.

Common Exceptions that are usually thrown by the JVM [↑](#)

1. `ArithmaticException` extends `RuntimeException`

The JVM throws this exception when you try to divide a number by zero.

Example :

```
public class X {
```

```
static int k = 0;
public static void main(String[] args){
    k = 10/0; //ArithmetricException
}

}
```

2. ClassCastException extends RuntimeException

The JVM throws this exception when you try to cast a reference variable to a type that fails the IS-A test. Don't worry I will discuss this in detail in the next chapter.

Example :

```
Object s = "asdf";
```

```
StringBuffer sb = (StringBuffer) s; //ClassCastException because s refers to a String and not a StringBuffer.
```

3. IndexOutOfBoundsException extends RuntimeException

This exception is a common superclass of exceptions that are thrown where an invalid index is used to access a value that supports indexed access.

For example, the JVM throws its subclass

ArrayIndexOutOfBoundsException when you attempt to access an array with an invalid index value such as a negative value or a value that is greater than the length (minus one, of course) of the array. Methods of String class throw another of its subclass

StringIndexOutOfBoundsException when you try to access a character at an invalid index.

Example :

```
int[] ia = new int[]{ 1, 2, 3}; // ia is of length 3.
```

```
System.out.println(ia[-1]); //ArrayIndexOutOfBoundsException
```

```
System.out.println(ia[3]); //ArrayIndexOutOfBoundsException
```

```
System.out.println("0123".charAt(4)); //StringIndexOutOfBoundsException
```

4. NullPointerException extends RuntimeException

The JVM throws this exception when you attempt to call a method or access a field using a reference variable that is pointing to null.

Example :

```
String s = null;
System.out.println(s.length()); //NullPointerException because s is null.
```

Common Errors usually thrown by the JVM ↗

1. ExceptionInInitializerError extends Error

The JVM throws this Error when any exception is thrown while initializing a static variable or a static block.

Example :

```
public class X {
    static int k = 0;
    static{
```

```

        k = 10/0; //throws ArithmeticException but i
t is wrapped into an ExceptionInInitializationError

    }

}

```

2. OutOfMemoryError extends Error

The JVM throws this Error when it runs out of memory. This usually happens when a program creates too many objects.

Example:

```

public static void main (String[] args) {
    StringBuilder sb = new StringBuilder("a long st
ring");
    for(int i=0; i<Integer.MAX_VALUE; i++){
        sb.append(sb.toString());
    }
}

```

3. StackOverflowError extends Error

The JVM throws this Error when the thread executing the method runs out of stack space. This usually happens when a method calls itself recursively and there is no boundary condition to stop the recursion.

Example :

```

public void m1(int k){
    m1(k++); //StackOverflowError

}

```

Exceptions thrown by Application Programmer ↗

All instances of **Exception** and its subclasses (except **RuntimeException**s) are generally thrown by the application programmer using the **throw**

statement. In some cases, application programmer may throw **RuntimeException**s as well.

1. **IllegalArgumentException** extends **RuntimeException**

This exception is thrown when a method receives an argument that the programmer has determined is not legal.

Example:

```
public void processData(byte[] data, int datatype)
{
    if(datatype != 1 && datatype != 2)
        throw new IllegalArgumentException("Invalid datatype "+datatype);
    else System.out.println("Data Processed.");
}
```

2. **NoClassDefFoundError** extends **Error**

Although it is an Error but it is not thrown by the JVM. It is thrown by a class loader (which is just another class in Java standard library) when it is not able to find the definition of a class that it is trying to load. Beginners get this error often while trying to run their program. For example, if your class has a package statement but you have not put the class file in its proper directory structure, the system class loader will not be able to find that class when you try to run it and will throw this error.

3. **NumberFormatException** extends **IllegalArgumentException**

This exception is thrown when a method that converts a **String** to a number receives a **String** that it cannot convert.

Example:

```
int i = Integer.parseInt("asdf");//a NumberFormatException will be thrown by the parseInt method
```

4. **SecurityException** extends **RuntimeException**

This exception is thrown if the Security Manager refuses to permit the requested operation due to restrictions placed by the JVM. For example, when a Java program runs in a sandbox (such as an applet) and tries to use prohibited APIs such as File I/O, the security manager throws this exception. Since this exception is explicitly thrown using the new keyword by a security manager class, it can be considered to be thrown by the application programmer.

15.3 Create and invoke a method that throws an exception

15.3.1 Creating a method that throws an exception [🔗](#)

Now that you know about the throws clause and the types of exceptions, let us look at the rules for creating a method that throws an exception. Actually, there is just one rule - If there is a possibility of a **checked exception** getting thrown out of a method, then that exception or its superclass exception must be declared in the **throws clause** of the method. The following are examples of a few valid methods that illustrate this rule:

1. **void foo(int x) throws Exception**

```
{  
    if(x == 2) throw new Exception(); //throws Exception only if x==2  
  
    else return;  
}
```

It doesn't matter whether the code throws an exception every time it runs or only some times. If there is a path of execution in which an exception will be thrown, the method must list that exception in the throws clause.

```
2. void foo() { //no throws clause necessary  
  
    if(someCondition) throw new RuntimeException()  
;  
    else throw new Error();  
}
```

`RuntimeException` and `Error` are **unchecked exceptions** and are therefore, exempt from being declared in the throws clause. Declaring them in the throws clause is valid though.

```
3. void foo() throws Exception
```

```
{  
    if(someCondition) throw new java.io.IOException(); //throwing a sub-exception  
  
    else return;  
}
```

It is ok to declare a superclass exception in the throws clause and throw a subclass exception in the method. But the reverse is not acceptable:

```
void foo() throws java.io.IOException  
  
{ //will not compile  
  
    if(someCondition) throw new Exception(); //throwing a super-exception  
  
    else return;
```

```
}
```

Remember that a throws clause is a commitment that you give to the user of this method that this method may throw only those exceptions that are listed in the throws clause. If you commit that you may only throw an **IOException** but then throw an **Exception** instead, then the caller will have a problem because the caller is only prepared to handle an **IOException** and not an **Exception**. Throwing an **Exception** will break the caller's code.

Declaring a broader exception (i.e. a superclass exception) and throwing a narrower exception (i.e. a subclass exception) is ok because if the caller is already prepared to handle the broader exception it can handle the narrower one without changing any code. For example, an **IOException** is an **Exception** and by throwing **IOException**, you are not breaking your commitment if you have committed that you may throw an **Exception**.

4. void foo() throws java.io.IOException, java.sql.SQLException

//can throw a common superclass exception as well.

```
{
    if(someCondition) throw new java.io.IOException();
    else throw new java.sql.SQLException();
}
```

If a method throws multiple exceptions, you can either list them individually or list a common superclass. The logic behind this is the same as before. As long as the method is true to what it has committed in its declaration, it is good!

5. void foo() throws Exception, java.io.IOException

```
{//specifying IOException is redundant because IOException is a subclass of Exception}
```

```
    if(someCondition) throw new java.io.IOException();
    else throw new java.sql.SQLException();
}
```

It is ok for a method to declare a superclass exception as well as a subclass exception even though adding the subclass exception to the list when a superclass is present is redundant.

6. **void foo() throws Exception**

```
{ //declaring Exception in the throws clause even though it is not thrown by the method body
```

```
    System.out.println("hello");
}
```

A method can declare any exception in its throws clause irrespective of whether that method actually throws that exception or not. It is sometimes useful to "future-proof" a method by declaring Exception in the throws clause if you believe that the method's implementation may change later. By declaring Exception in the throws clause, the users of the method will not have to change their code if the method actually starts throwing Exception or any of Exception subclasses later.

7. **void foo1() {**

```
    try{
        if(someCondition) throw new Exception(); // will be caught by the catch block
    }
    else return;
```

```
    }catch(Exception e){  
        }  
    }
```

The requirement to list an exception in the throws clause is applicable only when an exception is thrown out of the method to the caller. If the code inside a method throws an exception but that exception is caught within the method itself, there is no need to declare it in the throws clause of the method.

Remember that an exception can only be caught by a catch block and not by a finally block. Therefore, the throws clause is necessary in the following method:

```
void foo2() throws Exception  
{  
    try{  
        if(someCondition) throw new Exception();  
        else return;  
    }finally {  
        System.out.println("in finally"); //will be  
        executed but the exception is not caught here  
    }  
}
```

8. void foo() throws Throwable

```
{  
    if(true) throw new Exception();  
    else return;  
}
```

As you know, **Throwable** is the root of all exception classes, and therefore, if you declare **Throwable** in the throws clause, you can throw

any exception in the method.

Although not important for the exam, deciding which exception to throw and which to declare is an important matter.

Declaring exceptions ↴

As shown in point number 8 above, declaring a broad exception class in the throws clause is an easy way to get rid of any compilation errors with the method if the method throws different kinds of exceptions from different parts of its code. However, this is considered a bad practice because this burdens the user of the method with dealing with a broad range of exceptions.

On the other hand, listing specific exception classes individually restricts the future modifiability of a method because throwing new exceptions later will break other people's code.

It is recommended to be balanced in your approach towards listing exceptions in the throws clause. You should try to be only as specific as is possible without compromising the modifiability of the method. For example, I/O related methods may encounter different kinds of issues while reading a file. It may not be possible to identify all such issues while writing the method. New issues may be discovered later and you may have to modify your method to accommodate those. Therefore, it is better to declare a common superclass `IOException` in the throws clause instead of individual subclasses such as `FileNotFoundException` or `EOFException`.

Throwing exceptions ↴

You should always throw the most specific exception possible. For example, if you don't find the file while trying to open it, you should throw a `FileNotFoundException` instead of an `IOException` or `Exception`. By throwing the most specific exception, you give more information to the caller about the problem. This helps the caller in determining the most suitable resolution of the problem.

15.3.2 Throwing exceptions from initializers and constructors ↴

Throwing exceptions from static initializers ↴

The JVM executes static initializers automatically when it loads a class. Although the JVM loads a class due some action taken by the application code such as creating an object of the class or invoking a static method of the class, there is no direct invocation of a static initializer from the application. Therefore, if a static initializer ends up throwing an exception, there is no way for the application to handle that exception and to recover from it. For this reason, a static initializer is not allowed to throw any checked exception. If the compiler sees a possibility of a checked exception getting thrown out of a static initializer, it will generate an error. For example, the following code will not compile:

```
public class TestClass{  
    static int i = 5;  
    static{  
        if(i == 0) throw new Exception();  
    }  
}
```

The error generated by the compiler message says, "Error: unreported exception `java.lang.Exception`; must be caught or declared to be thrown", which is a bit misleading because there is no

where to declare the exception!

Throwing exceptions from instance initializers [↳](#)

Instance initializers are always executed when the application code tries to create an instance of the class. Thus, it is possible for the application code to catch an exception thrown by an instance initializer and for the same reason, instance initializers are allowed to throw checked exceptions. However, an instance initializer has the same problem as a static initializer - there is no way to specify a throws clause for an instance initializer. Recall that an instance initializer is executed no matter which constructor of the class is invoked. In that sense, an instance initializer is kind of a part of each constructor of the class. Therefore, an exception thrown from an instance initializer can be thought of as an exception thrown by every constructor of the class. Thus, if we declare an exception thrown from an instance initializer in the throws clause of each constructor of the class, we should be good. That is exactly what Java mandates. Here is an example:

```
public class TestClass{
    int i = 5;

    {
        if(i == 0) throw new Exception();
    }

    TestClass() throws Exception{
    }

    TestClass(int x) throws Exception{
    }
}
```

In the above code, since the instance initializer of **TestClass** throws **Exception**, each of its two constructors must declare **Exception** in their

throws clauses. Now, can you guess whether the following code will compile or not?

```
public class TestClass{  
    int i = 5;  
  
    {  
        if(i == 0) throw new Exception();  
    }  
  
}
```

Observe that **TestClass** does not define any constructor explicitly. Therefore, the compiler will provide the default no-args constructor for this class. However, the default constructor does not have any throws clause. Therefore, the exception thrown by the instance initializer is not getting declared by that constructor. Hence, the above code will not compile.

Throwing exceptions from constructors [↳](#)

Just like a method, a constructor is allowed to throw any exception as long as it declares that exception in its throws clause. However, there is one important difference between a method and a constructor. Recall that the first line of every constructor is always a call to a constructor of its super class or to another constructor of the same class. Thus, if a constructor decides to throw an exception, it has an impact on the subclass because that exception will be propagated to the subclass constructor as well. Therefore, if a subclass constructor invokes a superclass constructor that throws an exception, that subclass constructor must also declare that exception in its throws clause. Here is an example:

```
class Fruit{  
    Fruit() throws Exception{  
        if(Math.random()>0.5) throws Exception; //throws  
        an exception randomly  
  
    }  
}
```

```
Fruit(int calories){  
}  
}  
}
```

The `Fruit` class above has two constructors - one with a throws clause and one without. Now let's see some subclasses of `Fruit`:

```
class Apple extends Fruit{ //will NOT compile.  
}  
}
```

Notice that since `Apple` doesn't define any constructor explicitly, the compiler will provide the default no-args constructor to this class. Furthermore, the compiler will insert a call to `super()`; in the default constructor. This default constructor will cause the `Apple` class to fail compilation because it has neither a throws clause nor a wrapping try/catch block that could catch the exception thrown by the call to `super()`; , which is nothing but a call to `Fruit`'s no-args constructor. To make it compile, you must provide the no-args constructor with an appropriate throws clause yourself:

```
class Apple extends Fruit{  
    Apple() throws Exception{  
    }  
  
    Apple(int calories) { //no throws clause needed  
        super(calories);  
    }  
}
```

Observe that the int constructor of `Apple` does not require any throws clause because it invokes `Fruit`'s int constructor, which does not throw any exceptions. You could also change `Apple`'s no-args constructor to invoke `Apple`'s int constructor using `this(100)`; to avoid having a throws clause:

```

class Apple extends Fruit{
    Apple() {
        this(100);
    }

    Apple(int calories) { //no throws clause needed

        super(calories);
    }
}

```

A subclass constructor is free to throw any new exception along with the exceptions listed in the throws clause of a super class constructor.

Note that it is not possible for a subclass constructor to catch an exception thrown by the superclass constructor because to do that you would have to put the call to `super(...)` in a try block, in which case the call to `super(...)` will not be at the first line of the constructor!

15.3.3 Invoking a method that throws an exception [🔗](#)

The rules for invoking a method that throws an exception are similar to the ones for creating a method that throws an exception. As far as the compiler is concerned, there is no difference between using a `throw` statement to throw an exception and invoking a method that throws an exception to throw an exception. In both the cases, the compiler forces you to either declare the exception in your `throws` clause or handle the exception using a `try/catch` block. Of course, as discussed before, the compiler is only concerned with **checked exceptions**. The following example illustrates this point:

```

public class TestClass{

    public static void foo() throws Exception{

```

```
        if(true) throw new Exception();
        else return;
    }

    public static void main(String[] args) throws Exception {
        foo();
    }

}
```

In the above code, the method `foo()` throws an `Exception` using the `throw` statement. Since `Exception` is a checked exception, the compiler forces it to be declared in the `throws` clause. On the other hand, instead of throwing an exception explicitly using the `throw` statement, `main` invokes `foo`. But the compiler knows that invoking `foo` may result in an `Exception` being thrown (because it is mentioned in the `throws` clause of `foo`) and so the compiler forces `main` to declare `Exception` in its `throws` clause as well. The other option for `main` is, of course, to handle the exception itself:

```
public static void main(String[] args) { //no throws
clause necessary

try{
    foo();
}catch(Exception e){
    e.printStackTrace();
}
}
```

It is ok to use a catch block with a broader exception class to catch a narrower exception. For example, if you are calling a method that declares `Exception` in its `throws` clause, you can use `catch(Throwable e)` to handle that exception:

```
public static void foo() throws Exception{
    //some code that throws an exception
}
```

```
}

public static void main(String[] args) {
    try{
        foo();
    }catch(Throwable e)
{

    e.printStackTrace();
}
}
```

The reverse, i.e., using a catch block with a narrower exception to catch a broader exception is not acceptable. Thus, the following will not compile:

```
public static void main(String[] args) {
    try{
        foo();
    }catch(java.io.IOException e)

{ //catch clause is too narrow to catch Exception

    e.printStackTrace();
}
}
```

You can think of a catch clause as a basket of a particular size. You can use a bigger basket to catch a smaller exception, but you cannot use a smaller basket to catch a bigger exception. Note that I am using the word "bigger" here in the sense that a bigger exception has a bigger tree of subclasses than a smaller exception. For example, **Throwable** has a lot more subclasses than **Exception** because the tree of **Throwable** includes the tree of **Exception** as well as the tree of **Error**. Similarly, **Exception** is bigger than **IOException** because the tree of **Exception** includes the tree of **IOException** along with a lot of other exception subclasses.

If a method doesn't want to catch the exception then it must declare that exception (or its superclass) in its **throws** clause. This is no different from the rule that you have seen before while creating a method that throws an exception. For example, assuming that the method **foo** throws **Exception**, here are valid throws clauses for a method that invokes **foo**:

```
//declare the same exception class  
  
public static void bar() throws Exception  
  
{  
    foo();  
}  
  
//declare a super class of the exception class thrown by foo  
  
public static void bar() throws Throwable  
  
{  
    foo();  
}
```

Of course, **bar** is not limited to throwing just the exceptions declared in the throws clause of **foo**. It can add its own exceptions to the throws clause irrespective of whether the code inside the method throws them or not.

To catch or to throw ↗

The decision to catch an exception or to let it propagate to the caller depends on whether you can resolve the problem that resulted in the exception being thrown or not. Consider the following code for a method that computes simple interest:

```
public static double computeInterest(double p, double r, int t) throws Exception{
    if(t<0) throw new Exception("t must be > 0");
    else return p*r*t;
}
```

and the following code that uses the above method:

```
public static void main(String[] args){
    double interest = 0.0;
    try{
        computeInterest(100, 0.1, -1);
    }catch(Exception e){}
    System.out.println(interest);
}
```

Upon execution, the main method prints interest as **0.0** even though the **computeInterest** method did not really compute interest at all. It threw an exception because t was less than 0. However, as a user of the program, you won't know that there was actually a failure during the computation of interest.

While the **computeInterest** method did its job of telling the **main** method of a problem in computation by throwing an exception, the **main** method swepted this problem silently under the rug by using an empty **catch** block. This is called "**swallowing the exception**" and is a bad practice.

The purpose of a **catch** block is to resolve the problem and not to cover up the problem. By covering up the problem, the program keeps running but starts producing illogical results. Ideally, **main** should not have caught the exception but declared the exception in its **throws** clause because it is in no position to resolve the problem. It would have been appropriate for a program with GUI to catch the exception and ask the user to input valid arguments.

Sometimes, it becomes necessary to catch an exception even though no resolution is possible at that point. The right approach in such a case is to log the exception to the console so that the program can be easily debugged later by inspecting the logs.

```

public static void main(String[] args){
    double interest = 0.0;
    try{
        computeInterest(100, 0.1, -1);
    }catch(Exception e){
        e.printStackTrace();
        //or System.out.println(e);

    }
    System.out.println(interest);
}

```

15.3.4 Using multiple catch blocks [↳](#)

The Java standard library has a huge number of exception classes. On top of it, Java allows you to create your own exception classes as well. The purpose of having so many exception classes instead of having a few generic exception classes is to capture as many details of an exceptional situation as possible. These details help the caller of a method resolve the problem. For example, instead of throwing a general `IOException` when a file is not found, you should throw a more specific `FileNotFoundException` because it describes the problem more accurately to the caller. If a method expects to encounter different issues, it should throw a different exception for each issue. The caller can use multiple catch blocks to catch these exceptions and take action based on which exception is thrown by a method. Here is how -

```

import java.io.*;
public class TestClass{

    //assume that CharConversionException, FileNotFoundException,
    //and EOFException extend IOException

```

```

void foo(int x) throws IOException{
    if(x == 0) throw new CharConversionException();
    else if(x == 1) throw new FileNotFoundException(
);
    else throw new EOFException();
}

public static void main(String[] args){
    TestClass tc = new TestClass();
    try{
        tc.foo(2);
    }catch(EOFException eofe){
        System.out.println("End of file reached");
    }
    catch(CharConversionException cce){
        System.out.println("Some problem with file sy
stem");
    }
    catch(FileNotFoundException fnfe){
        System.out.println("No such file found");
    }
    catch(IOException ioe){
        System.out.println("Unknown I/O Exception");
    }
}
}

```

In the above code, if the call to `foo` throws an exception, the control will check each catch clause one by one to see if it is able to catch the exception. The control will enter the first catch block that is able to handle the exception. So for example, if the call to `foo` throws `FileNotFoundException`, the first catch block that is able to handle this exception is the third one, i.e., the `catch(FileNotFoundException fnfe)` block. The last catch clause, i.e., `catch(IOException ioe)` is also able to handle a `FileNotFoundException` but the control will not bother with it because it has already found a matching catch block before it reaches the `catch(IOException)` clause.

Unreachable catch blocks [↳](#)

The above program makes for an interesting problem. What if you move the `catch(IOException ioe)` block before the `catch(FileNotFoundException fnfe)` block? Well, the code will fail to compile. The `catch(IOException ioe)` clause will always satisfy a `FileNotFoundException` and so the control will never get to enter `catch(FileNotFoundException fnfe)` block. Therefore, the compiler will deem the `catch(FileNotFoundException fnfe)` block as unreachable. It is like putting a bigger basket above a smaller one. The smaller basket will never be able to catch anything because of the bigger one sitting above it!

A similar issue will occur if you change `foo`'s throws clause in the above program from `throws IOException` to `throws CharConversionException, FileNotFoundException, EOFException`. In this case, the compiler will realize that `foo` cannot throw an `IOException` (because it is not listed in `foo`'s throws clause anymore) and all three exceptions that it can actually throw are being caught already by the three catch blocks and therefore, the `catch(IOException ioe)` block will never be executed. Ideally, the compiler should refuse to compile the code but the JLS allows this and only requires a compiler to generate a warning.

Wait, there is more. What if you change the `catch(IOException ioe)` clause in the `main` method to `catch(Exception e)`? You know that `foo` doesn't throw `Exception`. So, what do you think the compiler will do? Well, actually, the compiler won't have an issue with it. Remember that one of the subclasses of `Exception` is `RuntimeException`, which is an unchecked exception. Any method is free to throw a `RuntimeException` without declaring it in its throws clause. Therefore, the compiler needs to consider the possibility that `foo` may throw a `RuntimeException` and the `catch(Exception e)` block will get executed if that happens. Thus, the compiler has no choice but to accept the code.

Nested try statements [↳](#)

It is possible to nest `try` statements inside other `try`, `catch`, or `finally` blocks. The rules for such `try` statements are exactly the same as those for

regular `try` statements. But remember that a `catch` block associated with a nested `try` statement is cannot catch an exception thrown by the outer `try` block. Here is an example:

```
void foo() throws EOFException {  
    try{  
        if(true) throw new FileNotFoundException();  
        else throw new EOFException();  
    }catch(FileNotFoundException fnfe){  
        try{  
            throw new EOFException();  
        }catch(EOFException eofe){  
            eofe.printStackTrace();  
        }  
    }  
}
```

In the above code, the `catch(EOFException eofe)` block is associated with the `try` statement that is nested inside the `catch(FileNotFoundException fnfe)` block. Therefore, it cannot catch an `EOFException` thrown from the outer `try` block. In fact, if the outer `try` block throws `EOFException`, the control will not even enter the `catch(FileNotFoundException)` block because `catch(FileNotFoundException fnfe)` clause does not satisfy `EOFException`. Thus, the `EOFException` thrown from the outer `try` block will remain unhandled, which means it must be declared in the `throws` clause of the method.

Throwing an exception from catch or finally [🔗](#)

It is possible to rethrow the same exception (or throw a new one) from the `catch` block. You may want to do this if you want to do something upon receiving an exception before letting it propagate up the call chain. Of course, if it is a checked exception you would have to declare it in the `throws` clause of the method. Here is an example:

```
void bar() throws EOFException {  
    try{  
        foo();  
    }catch(EOFException eofe){  
        //do something here and then rethrow the same exception again.  
  
        throw eofe;  
    }  
}
```

The same effect can be achieved with a **finally** block:

```
void bar() throws EOFException {  
    try{  
        foo();  
    }finally{  
        //do something here  
  
    }  
}
```

Since there is no catch block to catch the **EOFException** thrown by **foo** , the exception will automatically be thrown out of the **bar()** method to the caller of **bar()** , but only after the code in the **finally** block is executed.

Rethrowing an exception explicitly from a **catch** block is helpful only if you have multiple **catch** blocks and you want to do a different thing in every **catch** block.

It is also possible to throw an entirely new exception from the catch and finally blocks. Here is an example:

```
void bar() throws Exception {  
    try{
```

```
    foo();
}catch(EOFException eofe){
    throw new Exception();
}
}
```

The `EOFException` thrown by `foo` will be caught by `catch(EOFException eofe)` block. Since the catch block throws `Exception`, `bar` will end up throwing `Exception` to its caller. Thus, `bar` must list `Exception` in its `throws` clause.

If you throw an exception from the finally block, then the exception thrown from the try block or the catch block is ignored, and the exception thrown from the finally block is what gets thrown out of the method. The following example illustrates this point:

```
void bar() throws IOException {
    try{
        foo();
    }catch(EOFException eofe){
        throw new Exception();
    }
    finally{
        throw new IOException();
    }
}
```

Even though the catch block throws `Exception`, the JVM doesn't throw it to the caller of `bar`. It waits until the finally block finishes execution. However, the finally block throws a new `IOException`. So the JVM ignores the `Exception` that it was about to throw to the caller and throws the `IOException` instead. As far as the compiler is concerned, it realizes that `bar` can only throw an `IOException` and not `Exception` to the caller and therefore, it is ok with listing just `IOException` in `bar`'s `throws` clause.

15.4 Exercise

1. Create a method named `countVowels` that takes an array of characters as input and returns the number of vowels in the array. Furthermore, the method should throw a checked exception if the array contains the letter '`x`' .
2. Invoke the `countVowels` method from main in a loop and print its return value for each command line argument. Observe what happens in the following situations: there is no command line argument, there are multiple arguments, there are multiple arguments but the first argument contains an '`x`' . (Use String's `toCharArray` method to get an array of characters from the string.)
3. Ensure that your main method prints the number of vowels in other command line arguments even if one argument contains an '`x`' .
4. Pass null to the `countVowels` method and observe the output.
5. Modify `countVowels` method to throw an `IllegalArgumentException` if it is passed a `null` .
6. Modify `countVowels` method to return `-1` , if the input array is `null` and `0` , if the input array length is less than `10` . Do not use an `if` statement.

Chapter 16 Understanding Modules

- Describe how a modular project is compiled and run
- Declare modules and enable access between modules
- Describe the Modular JDK

16.1 Module Basics

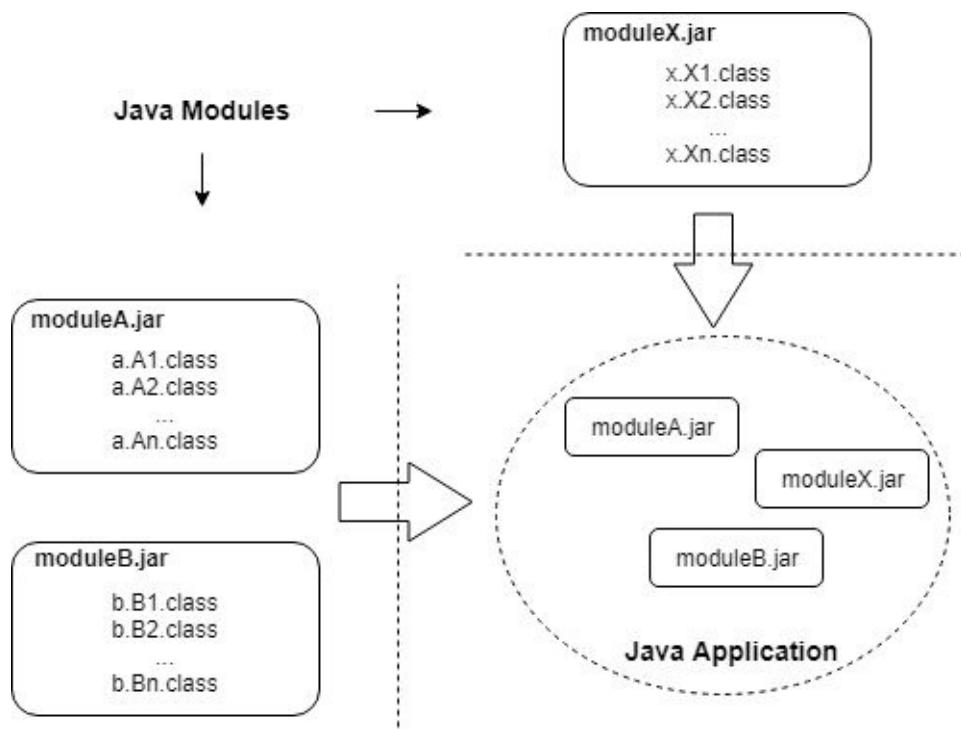
16.1.1 What are modules?

In the Kickstarter for Beginners chapter, you saw how to package individual classes of an application into a jar file and how to run an application using that jar file. The jar file approach has been a great way to package multiple files into a single file and to deliver applications as well as class libraries to users.

However, Jar files are not without problems. Packaging classes into a jar files without a well thought out plan gives rise to unwieldy applications that are difficult to update and/or reuse. Jar files also make it difficult to use just a part of an application without having the whole set of jar files. If you are a Java beginner or haven't been involved in professional level Java development much, you may find it hard to believe that the jar file approach also led the Java community to something called "Jar Hell". Jar Hell is a term coined to describe hard to debug behavior of applications due to the presence of multiple versions of a class library on the classpath and/or due to the use of different versions of a library used by different parts of the applications. For example, what if an application uses Spring and Hibernate and these libraries require a different versions of the same jdbc driver or a logging library?

Although the JDK did not provide any out of the box solution to such issues, various Java community projects such as Ant, Maven, and Gradle have attempted to provide some way of managing these issues.

With Java 9, the Java language designers came up with a new approach called the Java Module System, also known as Project Jigsaw, to packaging Java classes that attempts to solve at least some of the problems mentioned above. It envisions a Java application (or a library) to be composed not of classes or jar files but of modules, where each module of an application clearly specifies what other modules it depends on and what features it provides to other modules. A module, in turn, is composed of classes. A module, therefore, is a much coarser grained component of an application than a class. In other words, classes are packaged in a jar file to form a module, and modules are mixed and matched at run time to form an application. The following diagram illustrates how classes and modules come together to make an application.



The above diagram shows three jar files named moduleA.jar, moduleB.jar, and moduleX.jar. These module jar files are used together to create an application. Note that an application is just a loosely coupled collection of modules and is not packaged as a jar file itself. You will soon see that an application is created by specifying the required modules from different teams, groups, or companies on the command line at run time.

It is not like professional application teams were not already grouping independent parts of an application into separate jar files. Indeed, even before

the advent of modules, Java applications were composed of jars collected from various teams. But the process of grouping the classes and, more importantly, documenting what a jar file requires and provides, varied from team to team and from build tool to build tool. Java 9 takes the best of the practices prevalent in the industry and formalizes them into the Module System.

Besides formally specifying a structural design for applications, Java 9 also includes new tools and enhances existing tools to help developers package their code as modules.

Package vs Module [🔗](#)

You may be wondering at this point where packages fit into this picture. After all, we have been using packages to organize our Java code since ages. Well, packages still play the same role. They are still used to organize our code but at a finer level.

Conceptually, a package creates a namespace where all of the code required to implement a particular functionality resides. It is a means to reduce the size of the code contained in a class file by separating the logic into multiple closely related classes. This helps in reading, understanding, and navigating the code. But this is from the perspective of the developer of the functionality and not from the perspective of the users of that functionality. The users of that functionality are not interested in how the code is organized. They are interested in knowing what it takes to use that functionality. Users simply want to use the functionality as a black box. Unfortunately, packages do not provide this information. Developers have been relying on informal methods such as naming jar files according to their own conventions and adding release notes to deliver this information.

This is where modules come in. A module provides a way to deliver functionality in a single artifact that includes not just the code but also the information on the services that the code provides and any dependencies that it may have.

Thus, if you are a developer of some functionality, you would still organize your code into packages. But you would then also package all of your classes (and

any other resources such as image files, xml files, or property files) that are used to implement this functionality into a single deliverable with a description of your functionality and its dependencies in a prescribed format in the form of a module. In other words, modules are not an alternative to packages but are on top of packages.

The Java module system is a fairly large topic with dedicated books written to cover it. However, the OCP Java SE 11 Part 1 exam focuses only on the basics of the Java module system.

There has been a fair bit of criticism of the Java Module system in the industry. Even though Java Modules is an advanced topic, Oracle has included it in the Part 1 Exam of the OCP certification, which tells me that they are quite serious about pushing it. For this reason, I will focus mostly on how to work with it from the exam perspective instead of getting into the analysis of its merits and demerits. However, if you are going to face technical interviews, you should go through online discussions and articles that talk about the problems associated with the Java Module system.

16.1.2 Declaring a module

Since a module is a logical unit of functionality from the users' perspective, it is best to first identify the functionality that we want to deliver as a module. For example, let us say we are developing an application for finance and one of its functions is to compute simple interest. We want to deliver this simple interest calculator as a module.

Naming a module

A module name follows the same rules and conventions that you saw for naming packages and classes earlier. A valid package name would therefore, be a valid module name. In short, the name must be a valid Java identifier (so, special

characters such as dash and slash are out but underscore and dot are in) and it should follow the reverse domain name pattern to avoid name clash. Ideally, the name should be descriptive enough to tell the user the purpose and/or functionality of the module instead of being too generic such as `tools` or `utils`. So, `com.abc.finance.calculators.simpleinterest` would be a good name for our module. However, for the purpose of this chapter, I am going to name the module as just `simpleinterest` to reduce clutter and to make it easy to try out the code that I am going to present here. Let us also decide that the FQCN of our class will be `simpleinterest.SimpleInterestCalculator` instead of `com.abc.finance.calculators.simpleinterest.SimpleInterestCalculator` for the same reason. Note that there is no relation between the module name and the package name(s) of the class(es) contained in that module. However, since both follow the reverse domain name convention, they may very well be the same.

The module descriptor [↑](#)

Remember I talked about the lack of any formal way to describe a package? This is where a module differs from a package. Module descriptor is the formal way to describe a module. Every module must have a module descriptor that specifies the name of the module, what it provides, and what it requires. A module descriptor is nothing but a file by the name `module-info.java`. Yes, the extension of the file is `java`. It is a Java file that will be compiled to generate a class file by the name `module-info.class`. When we talk of a module, it is this descriptor that we generally refer to. Indeed, the module itself is a black box and the module descriptor is all the user should worry about. In our case, the contents of the module descriptor will be as follows:

```
//in file module-info.java
```

```
module simpleinterest{  
}
```

There isn't much in this module descriptor. All it says is that this is a module named `simpleinterest`. But it is a valid module descriptor nonetheless and it illustrates the basic syntax for defining a module. You have the module

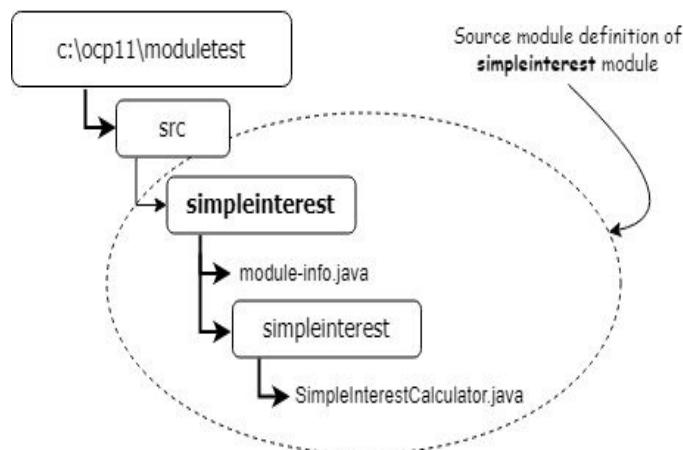
keyword and the module name, followed by the opening and closing braces. We will be adding more information in this descriptor as we go along. At this time, however, it is a good start!

The `module-info.java` file cannot be empty.

16.1.3 Directory structure of a module [↳](#)

The directory structures for the sources and for the compiled classes of a module are almost exactly the same as that of a package. The only difference is that while a package may reside in any folder, a module resides in a folder by the same name as the module. This rule applies to the source code as well as the class files. If the module directory contains the source code, it is called "source module definition" and if the module directory contains compiled classes, it is called "module definition".

Thus, in our example, the source code and compiled classes of our module must reside in folders named `simpleinterest`. The following image illustrates the directory structure that I will use to develop and execute the code for our module:



My base directory is `c:\ocp11\moduletest`. This is the directory where I am going to develop and run all the code presented in this chapter. If you are using *nix, you might want to create a `moduletest` directory under

/home/<username>. (Remember to flip \ to / in directory paths if you are using *nix.)

Under the base directory, I have a `src` directory, where I will keep the source code for all of my modules. Since this particular module is named `simpleinterest`, I have a directory by the same name under `src`. The directory tree rooted at `simpleinterest` is therefore, the source module definition of the `simpleinterest` module. Furthermore, since our `SimpleInterestCalculator` class belongs to the `simpleinterest` package, I have kept `SimpleInterestCalculator.java` under `simpleinterest\simpleinterest`.

Note that it is not necessary to follow the package structure for organizing Java source files. You may keep all your Java source file(s) directly under the module root folder as well. However, organizing the source code as per their packages has a couple of advantages. Besides making the source code easy to navigate, it is understood well by the Java compiler. The compiler can automatically locate and compile the code present in a source module definition.

The contents of `SimpleInterestCalculator.java` are as follows:

```
package simpleinterest;
public class SimpleInterestCalculator{
    public double calculate(double principle, double rate, double time){
        return principle*rate*time;
    }
    public static void main(String[] args){
        System.out.println(new SimpleInterestCalculator()
            .calculate(100, .05, 2));
    }
}
```

16.2 Describe how a modular project is compiled and run

16.2.1 Compiling a module ↗

Now that our directory structure and the code base for the module is ready, let us compile it. Open a command/shell prompt, `cd` to the `c:\ocp11\moduletest` directory and run the following command:

```
c:\ocp11\moduletest>javac -d out --module-source-path src --module simpleinterest
```

The above command uses three command line switches: `-d`, `--module-source-path`, and `--module`.

You have used the `-d` switch before to direct the output of the Java compiler to a particular directory. It works the same way here. So, `-d out` will cause the compiler to produce its output in the `out` directory. The compiler will create the `out` directory if it does not already exist.

The `--module-source-path` switch tells the compiler the location of the source module definition. Since we have kept our source code in a directory under the `src` directory, that is what we have specified here as well. Observe the double dash in this switch.

Since the name of the module is `simpleinterest`, the compiler will create a `simpleinterest` directory under `out`. The `out\simpleinterest` directory is the `simpleinterest` module's "root directory" and the tree rooted under `simpleinterest` is the "module definition" of the `simpleinterest` module.

The compiler will save the class files of this module inside `out\simpleinterest` as per the package driven directory structure of the classes of this module. For example, since `SimpleInterestCalculator` class is in `simpleinterest` package, the compiler will put `SimpleInterestCalculator.class` under `out\simpleinterest\simpleinterest` directory.

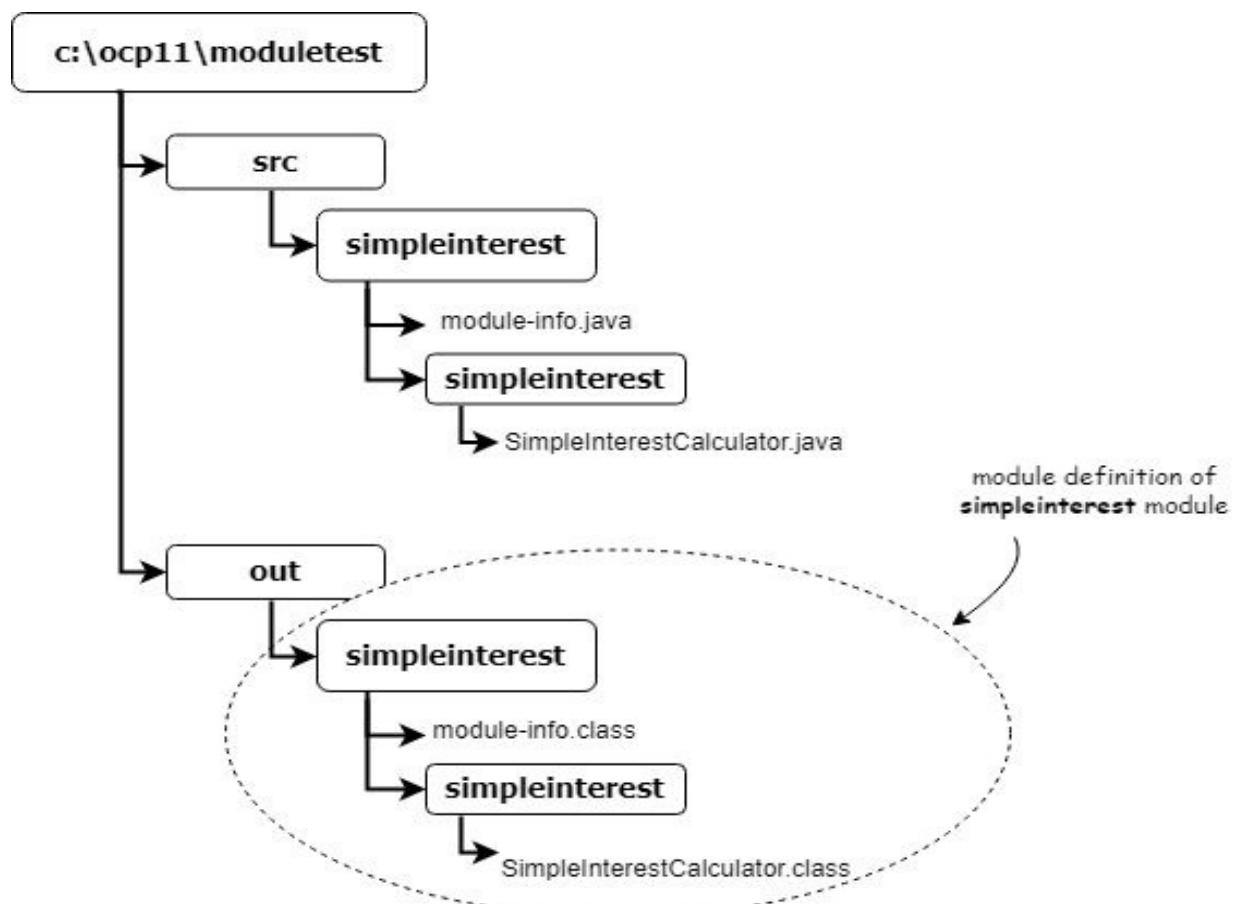
Note that I am using relative paths instead of absolute paths (i.e. `out` instead of `c:\ocp11\moduletest\out`). Relative paths are relative to the current directory, and since our current directory is `c:\ocp11\moduletest`, `out`

resolves to `c:\ocp11\moduletest\out`. You may use absolute paths and relative paths interchangeably in these commands but it is better to use relative paths because they will work even if your base directory something different from `c:\ocp11\moduletest`.

The `--module` switch specifies the name of the module that we want to compile. The compiler tries to locate this module in the paths specified in `module-source-path`. In this case, we want to compile the `simpleinterest` module. Again, observe the double dash.

Based on the above options, the compiler will look for `module-info.java` under `src\simpleinterest` directory and will compile all the Java files under all the directories that are present inside `src\simpleinterest` directory.

Upon successful execution of the above command, you should see the directory structure as shown below:



Observe the directory structure generated by the compiler under `out\simpleinterest`. It is the same as what you get after compiling non-modular Java source files.

I suggest that you play around with the above command by changing the particulars of this set up. For example, rename the module source directory to something else or move the Java source file to another location within the module source directory and observe the error message generated by the compiler.

Compiling individual module source files [🔗](#)

It is possible to compile module source files without using the `--module` option by listing the files (either individually or using a wildcard) that you want to compile. For example, the following old style command will produce the same result as produced by the command used above:

```
c:\ocp11\moduletest>javac -d out\simpleinterest  
src\simpleinterest\module-info.java  
src\simpleinterest\SimpleInterestCalculator.java
```

Observe that you have to specify the module root directory component `simpleinterest` in the output path explicitly. Without this, the compiler will save the output under `out` instead of `out\simpleinterest`. Listing individual files for compilation is not a preferred way to compile a module and is seldom used. You will not be tested on this in the exam either. For the purpose of the exam, you are expected to know how to compile a module using the `--module` switch only.

16.2.2 Running a module [🔗](#)

Running a module means executing the main method of a class contained in that module. Thus, the class that we want to execute must have the standard main method. The following command shows how to run our `simpleinterest`

module that we just compiled.

```
c:\ocp11\moduletest>java --module-path out --module simpleinterest/simpleinterest.SimpleInterestCalculator
```

This command uses two switches: `--module-path` and `--module`.

The `module-path` switch is used to tell Java the location of the module definition. This is the directory where the module's root directory is located or, if the module is packaged as a jar file, module-path is the directory where that jar is located. In our case, we have specified `out` in `module-path` because our module's root directory, i.e., `simpleinterest`, is located in `out`.

The `module` switch is used to tell Java the name of the module that we want to execute. Observe the value specified for `--module`. It is in the format `<module name>/<main class name>` because, as of now, our module is present in an exploded format and does not contain any information about the main class. So, we need to tell the JVM about the class that we want to run as well. Once we package our module into a jar, we won't need to specify the main class on the command line.

Packaging a module

The process of packaging classes of a module into a jar file is the same as the one we saw for packaging classes into a regular jar. The only thing special about a module jar is that the name of the jar is, by convention, the same as the name of the module (with `.jar` extension, of course). Inside the jar file, class files must still exist in their package driven directory structure just like before.

Observe that a module also has a `module-info.class` in its root folder. This class must be in the root folder of the module's jar as well. Since the structure of the files in a modular jar file mimics the directory structure of the compiled module, a modular jar is also considered as the "module definition" of a module.

Here is the command that will package our module into a jar file:

```
c:\ocp11\moduletest>jar --create --file simpleinterest.jar --main-class simpleinterest.SimpleInterestCalculator -C
```

```
out\simpleinterest .
```

This should create `simpleinterest.jar` in the `moduletest` directory. Once

The `--create` switch tells the jar tool that we want to create something and the `--file` and the subsequent value specify the jar file that we want to create. The `--main-class` switch makes the jar tool add a `Main-Class` entry in the resulting jar file's manifest.

The `-C` switch needs some explanation. We want the jar file to mimic the directory structure of the module, which means the structure inside the jar file should be the same as the structure inside the `out\simpleinterest` directory. But we are executing the command from the `moduletest` directory. There are two extra levels under `moduletest` (`out` and `simpleinterest`) that we don't want reflected in the jar file. The `-C` option is to make the jar tool change its working directory before adding files to the jar. We want it to step down to `out\simpleinterest` directory and then include the files from there.

The dot at the end of the command means that we want to include everything in its current directory (which is now `out\simpleinterest` due to the use of the `-C` switch) to be included in the jar file.

Running this module is now a piece of cake:

```
c:\ocp11\moduletest>java --module-path .
--module com.abc.finance.calculators.simple
interest
```

Observe that, since we added the Main-Class attribute in module jar's manifest, we are able to "run" the module.

You need not worry too much about the various switches used in the jar command. You won't be tested on how to create a jar file in the exam. However, executing a module packaged in a jar file is on the exam. So, it would be good if you understand how a modular jar file is created.

Understanding module-path [🔗](#)

Another interesting thing in the above command is the value that we have specified for **module-path**. We have specified only a dot for the **module-path** because our module jar is in the current directory. A module-path contains all the locations where you want the JVM to search for module definitions. While trying to load a module, the JVM looks for that module in all the locations specified in the **module-path**. It expects to find exactly one of the two things - a directory with the same name as the name of the module or a jar file containing **module-info.class** for that module. The name of the jar file is not important because the JVM takes the name of the module from the **module-info.class** contained in the jar file. The same module should not be present more than once on the module-path, otherwise the JVM will complain.

In the command used above, while searching for the module named **simpleinterest**, the JVM will look for a jar file that contains **module-info.class** with the module name as **simpleinterest** at the root or a directory by the name of **simpleinterest** containing appropriate **module-info.class**.

Compare this with the **-classpath** switch, which requires all jar files that you want on the class path to be listed explicitly instead of requiring just the directory that contains the jar file(s).

16.3 Declare modules and enable access between modules

16.3.1 Enabling access between modules [🔗](#)

Let us improve the design of our **simpleinterest** module a little bit. It is a good design practice to define functionality in the form of an interface and let the actual implementation implement that interface. This ensures that the user of the functionality is able to replace one implementation with another without

much impact on their code. Java modules allow us to take this technique even further by defining functionality separately in a module of its own. Separating the interface and the implementation into separate modules allows us to build an application by mixing and matching modules without the need to bundle classes that are not required for the application.

In our case, we will create a `calculators.InterestCalculator` interface in `calculators` module. We will then have our `SimpleInterestCalculator` class implement this interface. Here is the interface definition:

```
//in file calculators\calculators\InterestCalculator.java
```

```
package calculators;
public interface InterestCalculator{
    public double calculate(double principle, double rate, double time);
}
```

and the following is the module descriptor for the new module:

```
//in file calculators\module-info.java
```

```
module calculators{
    exports calculators;
}
```

The only thing new in this module definition is the `exports` clause. A module is an insular unit of packages. Members of a module are normally not accessible to code belonging to other modules. To make a package accessible to other modules, the package must be "exported" explicitly in the module descriptor. Making something accessible only through an explicit `exports` clause ensures that code from other modules does not access code that is internal to this module inadvertently. Note that I am talking about packages instead of classes because only packages can be exported and not individual classes. When you export a package, all of the public types contained in that package are made accessible to

other modules. In module parlance, accessing a module is called "reading" the module. So, the **exports** clause in the above code enables classes in other modules to "read" the **InterestCalculator** interface. This is also known as **module readability**.

The directory structure of this module is as per the source module definition format explained earlier.

The `c:\ocp11\moduletest\src\calculators` directory is the source directory for this module. It contains `module-info.java` and `calculators\InterestCalculator.java` files. Let us run the following command to compile this module:

```
c:\ocp11\moduletest>javac -d out --module-source-path  
src --module calculators
```

This should create appropriate output in the `out` directory.

Let us now move our attention to the `simpleinterest` module. Since we want `SimpleInterestCalculator` to implement `InterestCalculator` interface, we need to first make our intention clear by stating that our `simpleinterest` module "requires" access to the `calculators` module by modifying `simpleinterest`'s module descriptor as follows:

```
module simpleinterest{  
    requires calculators;  
}
```

The **requires** clause is the counterpart of the **exports** clause. The purpose of having a **requires** clause is to make the dependencies of a module explicitly clear to the users. The users of a module need not go through the import statements of each class of a module to know which other modules does this module depend on. All they need is to check the module-info.

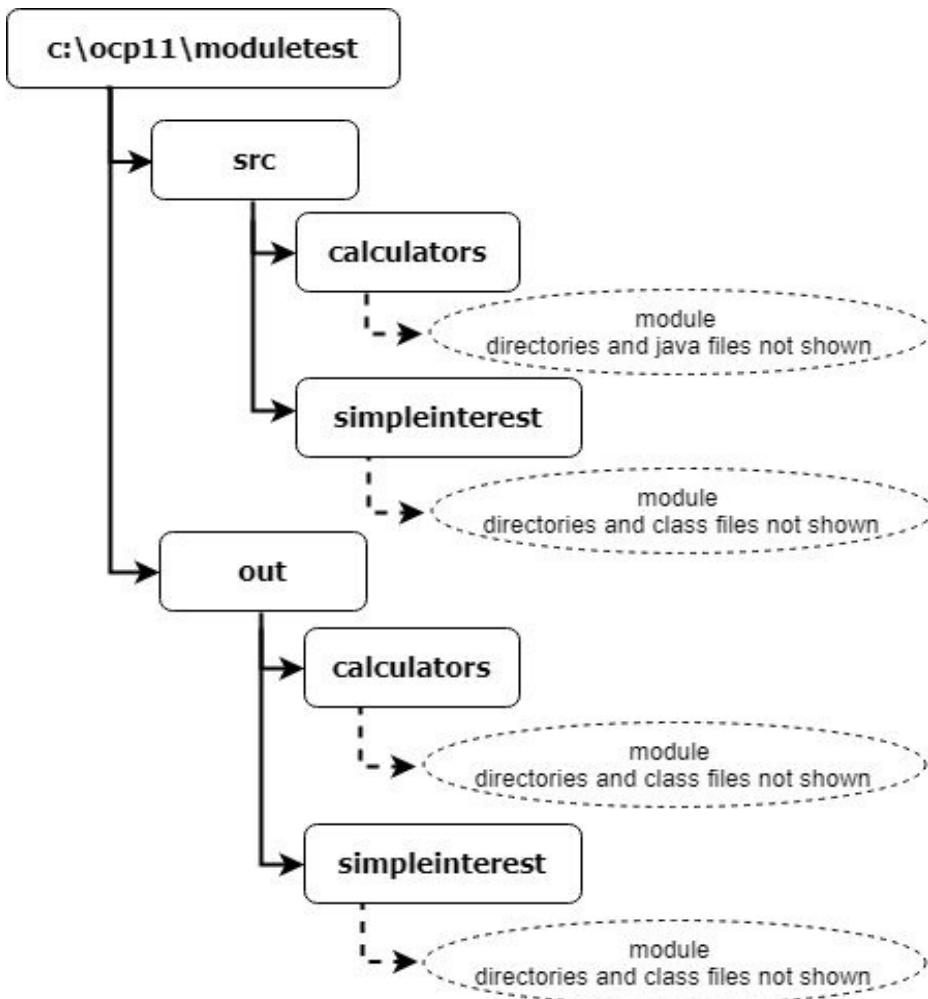
We can now modify `SimpleInterestCalculator.java` as follows:

```
package simpleinterest;
import calculators.InterestCalculator;
public class SimpleInterestCalculator implements InterestCalculator{
    public double calculate(double principle, double rate, double time){
        return principle*rate*time;
    }
    public static void main(String[] args){
        InterestCalculator ic = new SimpleInterestCalculator();
        System.out.println(ic.calculate(100, .05, 2));
    }
}
```

The following command can be used to compile this module:

```
c:\ocp11\moduletest>javac -d out --module-source-path src --module simpleinterest
```

After the compilation of this module, our directory structure looks like this:



We can now execute the `simpleinterest` module as we did before using the following command:

```
c:\ocp11\moduletest>java --module-path out --module simpleinterest/simpleinterest.SimpleInterestCalculator
```

16.3.2 Qualified exports [🔗](#)

The `exports` clause opens up a package to access from all other modules. Once a package is exported, you cannot stop any other module from accessing it. It is similar to making a member of a class `public`, and just like the `public` modifier, the `exports` clause makes a module less encapsulated.

The Java module systems allows you to fine tune access to a module only to

specific modules using a variation of the exports clause. It is called the **exports-to** clause or the qualified exports clause. The following is how it is used:

```
module <modulename>{
    exports <packagename> to <modulename(s)>;
}
```

For example, in the following module-info, the `com.abc.datatransfer` package is being made accessible only to `db` and `service` modules:

```
module datatransfer{
    exports com.abc.datatransfer to db, service;
}
```

The modules for which a package is made readable are called "friendly" modules. Java standard library uses this feature heavily for its internal modules. For example, the `java.base` module of the JDK has several packages which are supposed to be used only by a selected few internal modules. It uses qualified exports to achieve the same. Here is a partial code snippet from `java.base` module's module-info:

```
module java.base {
    ...
    exports jdk.internal.logger to
        java.logging;
    exports jdk.internal.perf to
        java.desktop,
        java.management,
        jdk.jvmstat;
    ...
}
```

There is no particular order in which unqualified and qualified exports have to appear in a module info. Conventionally, however, qualified exports are listed after all the unqualified exports.

Using qualified exports provides protection from creating inadvertent

dependencies on packages that are not meant to be used by everyone. Therefore, it is a good idea to use qualified exports as much as possible. Unqualified exports should be used only when you are absolutely sure that it is ok for the package to be used by any other module.

16.3.3 Transitive dependencies [↳](#)

A module **A** may read another module **B** by using a `requires` clause in its module-info. But what if the module **B** reads yet another module **C**? Does that mean module **A** reads the module **C** as well? Normally, no. Module dependencies are not transitive and if module **A** specifies that it requires only module **B**, it cannot read module **C** just because **B** reads **C**.

Now, let's see how this pans out in the following situation. What if a class in module **A** needs to invoke a method of a class in module **B**, whose return type is defined in module **C**? Something like this:

```
module ui{
    requires hr;
}
module hr{
    requires valueobjects;
    exports hrservice;
}

//code appearing in a class in ui module
```

```
HRService hrService = new HRService(); //HRService is defined in hr module
```

```
Employee e = hrService.getEmployee(employeeId); //Employee is defined in valueObjects module
```

Since the `ui` module does not have a `requires valuesobjects;` clause, the `ui` module cannot access the `Employee` class from the `valuesobjects` module. Thus, the above code will fail to compile.

The obvious solution is to add a `requires valuesobjects;` clause in the `ui` module. But this solution has a serious problem. What if the `hr` module has several requires clauses? How would the `ui` module know which of the `hr` module's requires clauses should be added to its module-info? Only multiple compilation failures can make this information known to the `ui` module!

This is where the `requires transitive` clause comes into picture. Since the `hr` module knows that whichever module requires `hrservice` package, will require classes in `valueobjects` module as well, it documents this information using the `requires transitive` clause, like this:

```
module hr{
    requires transitive valueobjects;
    exports hrservice;
}
```

This basically says that if you require the `hr` module, then you also require the `valueobjects` module. But more than merely documenting this fact, the `requires transitive` clause also eliminates the need for adding an extra `requires valuesobjects;` clause in the `ui` module. The `requires hr;` clause in the `ui` module automatically makes all the modules transitively required by `hr` module, readable to the `ui` module. This is called "implied readability".

It is not clear whether Qualified Exports and requires transitive are on the OCP Java 11 Part 1 exam or not. Some candidates have reported seeing options that use these clauses, so, it is a good idea to understand what they do.

16.4 Advanced module compilation and execution [↳](#)

Seeing all the new module related command line switches feels overwhelming at first. But once you understand the underlying theme behind these options, you will realize that they are not much different from the `-classpath` (and `-sourcepath`) options that you have used so far. The `module-path` and `module-source-path` switches are to a modular project what `classpath` and `sourcepath` switches are to a regular project. Earlier, you used to specify just the FQCN of the main class while launching an application, now, you specify the module name using the `--module` option. That is basically it.

I have observed that many students get confused due to the fact that `classpath` and `module-path` are not mutually exclusive. It is possible (and sometimes necessary) to use both the switches simultaneously in a command while compiling as well as executing a module. They are used together in projects that make use of modular as well as non-modular libraries. However, a discussion on this usage is not in scope for the OCP Java 11 Part 1 exam.

Another cause of confusion is the presence of multiple switches that mean and do the same things. For example, `--module-path` is the same as `-p` and `--module` is the same as `-m`.

In this section, I will summarize all the options that you need to use while compiling and executing a module.

16.4.1 Compiling multiple modules at once [↳](#)

In the previous example, we compiled the `calculators` and `simpleinterest` modules separately one after another. We could have easily compiled both of them together at the same time by listing both of them in the `--module` switch:

```
c:\ocp11\moduletest>javac -d out --module-source-path  
src --module calculators,simpleinterest
```

Observe that the module names are separated using a comma. In fact, Java will

compile the **calculators** module even without us specifying it explicitly in the **--module** switch because **javac** will notice that **simpleinterest** requires **calculators** and so, it will compile **calculators** first. However, this is possible only when the source code of the module is organized correctly in the package driven directory structure. If the source files are organized differently, **javac** will not be able to locate them on its own.

Using a module without source code

While developing a module, you may be required to use third party modules whose source code is not available. For example, what if the **calculators** module was developed by another team and you just had a **calculators.jar** for it?

Well, we can use the **--module-path** switch for telling **javac** the location of a module just like we used it while executing a module. Let's say we put **calculators.jar** in **c:\ocp11\moduletest\modulejars** directory. We can now use the following command to compile the **simpleinterest** module:

```
c:\ocp11\moduletest>javac --module-path modulejars -d out --module-source-path src --module simpleinterest
```

Observe that we didn't specify the actual jar name in the module path (although we could have done that as well). We specified just the directory that contains it. We can specify as many directories as we need separated by the path separator (; on windows and : on *nix). The difference between specifying a directory and specifying a jar is that when you specify a directory, all the jars as well as exploded module definitions present in that directory become available to the compiler (or to the jvm, in the case of execution), while in the other case, only the jar that you specify on the module path becomes available. It is alright to specify a directory on the module-path while learning but it is always better to specify the jars while developing a professional application. The command to run **simpleinterest** with explicitly listed jar files in the module-path looks like this:

```
c:\ocp11\moduletest>java --module-path  
    modulejars\simpleinterest.jar;modulejars\calculato  
rs.jar --module simpleinterest
```

16.4.2 Module jar vs regular jar [🔗](#)

As you saw before, a modular jar is no different from a regular non-modular jar from in terms of its structure. In fact, it is possible to use a modular jar just like a non-modular jar. If you are wondering whether you can run the main class of a modular jar file the old way, then yes, you can. For example, assuming that you have packaged the two modules into two jars as explained in the previous section and put them in `c:\ocp11\moduletest` directory, you can execute our `SimpleInterestCalculator` class with the following command:

```
c:\ocp11\moduletest>java -classpath  
calculators.jar;simpleinterest.jar  
simpleinterest.SimpleInterestCalculator
```

However, when you run a class using the `-classpath` or `-jar` option, you lose all the benefits of modules. Which means, the JVM will not enforce the access rules specified in module descriptors. A class from any jar can access a class from another jar. This option is helpful when a team has not moved to the modular structure for their project but wants to use a modular jar produced by another team.

Similarly, if you are running a modular project, it is possible to use a non-modular jar as a modular jar by simply putting it on the `module-path` instead of the `classpath`. Once you put a non-modular jar on the module-path, it is deemed to be an "automatic module". A discussion on automatic modules is beyond the scope of this exam and so, I will not talk about it any further in this book.

16.4.3 Summary of command line switches used for compilation



For compiling a Java class and/or module, you need to remember the following five command line options:

1. **--module-source-path** : This switch is used to specify the location of the module source files. It should point to the directory that contains source module definition. In other words it should point to the parent directory of the directory where `module-info.java` of the module is stored. For example, if your module name is `moduleA`, then the `module-info.java` for this module would be in `moduleA` directory and if `moduleA` directory exists in `src` directory, then `src` should be specified in the `--module-source-path` option, i.e., `--module-source-path src`

If `moduleA` depends on another module named `moduleB`, and if `moduleB` directory exists in `src2` directory, you can add this directory in `--module-source-path` as well, i.e., `--module-source-path src;src2`.

2. **-d** : This switch is **required** when you compile a module. It is used to specify the output directory. This is the directory where `javac` will generate the module and package driven directory structure and the class files for the sources. For example, if you specify `out` as the output directory, `javac` will create a directory under `out` with the same name as the name of the module and will create class files with appropriate package driven directory structure under that directory.

3. **--module** or **-m** : This switch is used when you want to compile all the source files of a particular module. This option is helpful when you want to compile all the files at once without listing any of the source files of a module individually in the command.

For example, if you have two Java files in `moduleA`, stored under `moduleA\A1.java` and `moduleA\A2.java`, you can compile both of them at the same time using the command: `javac --module-source-path src -d out --module moduleA`

Javac will find out all the Java source files under `moduleA` and compile all of them. It will create the class files under the output directory specified in `-d` option, i.e., `out`. Thus, the `out` directory will now have two class files - `moduleA/a/A1.class` and `moduleA/a/A2.class`.

4. **--module-path** or **-p** : This option specifies the location(s) of any other module upon which the module to be compiled depends. You can specify the module directories or jar files containing the module classes here. For example, if you want to compile `moduleA` and if it depends on some other module named `abc.util`, then you can add this to your javac command like this:

```
javac --module-source-path src --module-path  
thirdpartymodules/abc.util.jar -d out --module moduleA
```

5. **-classpath** : This option is used for compilation of non-modular code. If you are compiling regular non-modular code but that code depends on some classes, then you can put those classes/jars on the classpath using **-classpath** option.

Note: This option is not helpful for compilation of modular code because classes in modular code cannot read classes present on classpath. Modular code can only see other modular code. That is why, non-modular classes have to be converted into "automatic modules" and specified on **--module-path** as explained in the note below.

The following is not required for the part 1 but is good to know anyway because it is on the part 2 exam.

If your module depends on a non-modular third party jar, you need to do two things -

1. *Put that third party jar in `--module-path` .*

Putting a non-modular jar in `--module-path` causes that jar to be loaded as an "automatic module". The name of this module is assumed to be the same as the name of the jar minus any version numbers. For example, if you put `mysql-driver-6.0.jar` in `--module-path` , it will be loaded as an automatic

module with name mysql.driver . Name derivation is explained in detail in java.lang.module.ModuleFinder JavaDoc but for the exam, just remember that hyphens are converted into dots and the version number and extension part is removed.

It is also possible for a non-modular jar to specify its module name using Automatic-Module-Name: <module name> entry to the jar's MANIFEST.MF .

2. *Add a requires <module-name>; clause in module-info of your module.*

16.4.4 Summary of command line switches used for execution

You need to know about the following three command line switches for running a class that is contained in a module:

1. **--module-path** or **-p** : This switch tells the location of the module definition. If the module is packaged in a jar file, then you can either specify the path to the jar, or specify the path to the directory where the jar file stored (relative to the current directory or an absolute path). For example, **--module-path c:\javatest\modulejars**

You can also specify the location where the compiled module exists in the exploded form. For example, if your module is named **abc.math.utils** and this module is stored in **c:\javatest\output** , then you can use: **--module-path c:/javatest/output** . Remember that **c:\javatest\output** directory must contain **abc.math.utils** directory and the module files (including **module-info.class**) must be present in their appropriate directory structure under **abc.math.utils** directory.

You can specify as many module locations separated by path separator (; on windows and : on *nix) as required.

2. **--module** or **-m** : This switch is used to specify the module that you want to run. For example, if you want to run `abc.utils.Main` class of `abc.math.utils` module, you should write `--module abc.math.utils/abc.utils.Main`

If a module jar's manifest contains the `Main-Class` property, you can omit the main class from the command. For example, you can write, `--module abc.math.utils` instead of `--module abc.math.utils/abc.utils.Main`.

3. **-classpath** or **--class-path** or **-cp** : This switch is used while executing non-modular code. It is also used while executing a modular code to specify non-modular jars that are required by the project.

You are not required to know the following information for the part 1 exam. It is good to know anyway because it is required for the part 2 exam.

It is possible to treat modular code as non-modular by ignoring module options altogether. For example, if you want to run the same class using the older classpath option, you can do it like this:

`java -classpath mathutils.jar abc.utils.Main`

Remember that modular code cannot access code present on the `-classpath` but "automatic modules" are an exception to this rule. When a non-modular jar is put on `--module-path`, it becomes an "automatic module" but it can still access all the modular as well as non-modular code. In other words, a class from an automatic module can access classes present on `--module-path` as well as on `-classpath` without having any "requires" clause (remember that there is no `module-info` in automatic modules).

Thus, if your modular jar A depends on a non-modular jar B, you have to put that non-modular jar B on `--module-path`. You must also add appropriate requires

clause in your module A's module-info otherwise compilation of your module will not succeed. Furthermore, if the non-modular jar B depends on another non-modular jar C, then the non-modular jar C may be put on the classpath or on the module-path.

16.5 Describe the modular JDK

16.5.1 Modular JDK [↳](#)

In the previous sections of this chapter, I showed you how to create and use modules. It was important to discuss the basic concepts of modules first because Java 9 has modularized all the classes and packages of the JDK by following the same concepts. Although there are still a few more topics related to modules left that one needs to understand before they can fully appreciate the amount of restructuring done in JDK 9, those topics are not on the Java 11 part 1 certification exam. So, I will try to explain the modular JDK without relying too much on such topics.

If you look at the size of the Java runtime library, it is hard to believe that Java was originally designed to run on even the smallest of devices. Now, twenty five years later, Java has gone far beyond its goal. It has been enormously useful in building large mission critical applications as well as small desktop applications. One of the reasons for its success is the constant evolution of the features, tools, and libraries that Java comes bundled with.

Another key promise of Java was WORA, "Write Once Run Anywhere". Java came with a single set of API that you could use to write an application that would work the same on all platforms. Java achieved this feat by developing a run time environment that hid the differences in the underlying hardware and provided a common set of features.

Over the years, the types and capabilities of devices differed vastly and it became impossible to hide over the differences in hardware capabilities of so many platforms any longer. But if you have a Java runtime for a device and if

that runtime is not able to run a Java application developed for another device, then that would amount to abandoning the WORA promise. Java did finally abandon WORA when it came up with multiple versions of the runtime such as J2SE, J2EE, and J2ME. J2ME was small footprint version of the Java runtime that could run on initial generation of mobile phones. It was vastly different from J2SE and J2EE in terms of capabilities. However, the core Java library still remained the same.

By Java 8, the standard Java library, with over four thousand five hundred classes, had become a big monolithic pile of interdependent classes and packages that was impossible to split into smaller independent pieces. Even if an application used a very small part of the runtime library, it still had to be bundled with the whole library. You might have heard about a jar file named `rt.jar` that every Java installation comes with. This 60 MB jar file contains the complete Java SE library. If you ever wanted to distribute your Java application comprising just a single class, you would have to include this huge jar file along with it.

Modular JDK is an effort to solve this problem. It divided the JDK into several loosely coupled modules that can be combined at compile time, build time, as well as run time, into a variety of configurations as per the needs of the applications. For example, a large service side application can use the full JDK but a micro-service can be packaged with only the modules that are absolutely essential. It also allows applications that are bundled along with the runtime, to be shipped with smaller profiles, thus, increasing the reach of Java applications to more devices and platforms.

16.5.2 Organization of the modular JDK [↳](#)

The packages, or API packages, as they are called in technical jargon, contained in the Java platform are distributed into various modules. These modules can be categorized into two broad categories - standard modules and non-standard modules. The modules that are governed by the Java Community Process (JCP) are called Standard modules. Their names start with `java` . For example, `java.base` , `java.sql` , and `java.logging` are standard modules. All other modules that are specific to a JDK are called non-standard modules. Their

names start with `jdk`. For example, `jdk.jdeps`, `jdk.rmic`, and `jdkavadoc` are non-standard modules.

Similarly, standard packages have names starting with `java` or `javax` such as `java.lang`, `java.sql`, `javax.crypto` and `javax.net`, and the JDK specific non-standard packages, generally, have names starting with `jdk` such as `jdk.internal.perf`, `jdk.internal.logger`, and `jdk.internal.util`. Depending on the provider of the JDK, it may have packages reflecting the company name as well such as `sun.net` or `sun.utils.resource`.

It is important to know that a standard module may contain non-standard packages but a non-standard module does not contain any standard package.

The standard and non-standard modules are then combined to make a variety of Java configurations such as the **full Java SE Platform**, the **full JRE**, and the **full JDK**.

For the purpose of the exam, we need to focus on the Java SE platform.

The Java SE Platform

The Java Platform, Standard Edition (Java SE) is the **core Java platform** for general-purpose computing. It is composed only of the standard modules (i.e. whose names start with `java`) and is required to be supported by all Java implementations. You do not need to remember all the modules available in this platform but some of the most commonly used modules in this platform are `java.base`, `java.desktop`, `java.logging`, `java.sql`, `java.prefs`, `java.desktop`, and `java.net.http`. All of the programs that you have written or will write for preparing for the Java certification exam will require just this platform.

Note that this does not imply that the Java SE platform includes all of the standard modules. For example, `java.cardio` is a standard module (it is governed by the JCP) but is not a part of the Java SE Platform. Neither does it imply that this platform itself will not make use of classes from non-standard packages or modules. As I mentioned before, a standard module may contain

non-standard packages. All it means is that none of the modules included in the Java SE Platform "export" any non-standard package. In other words, if your module "requires" only the modules contained in the Java SE platform, it will depend only on the standard Java packages and will be portable to all Implementations of the Java SE Platform.

The `java.base` Module

This module defines the **foundational APIs** of the Java SE platform. By foundational, we mean that it lies at the core of all Java platforms. This module doesn't require any other module but every other module depends on this module. Again, you need not remember the complete list of packages contained in this module but here are a few important packages that you should be aware of - `java.lang` , `java.lang.annotation` , `java.io` , `java.nio` , `java.net` , `java.util` , `java.util.concurrent` , `java.security` , and `java.time` .

All of the Java API classes and packages that we have used in this book belong to this module. Every class that we have written in this book depends on a class from this module. Of course, `java.lang.Object` belongs to this module, after all! But did you notice that we never wrote `requires java.base;` in the module declarations of the modules that we developed in the previous section? The reason is that since everything in Java requires packages exported from this module, the Java compiler does not require it to be specified explicitly. In other words, much like the `java.lang` package is not required to be imported in a class explicitly, the `java.base` module is not required to be specified in a `requires` clause of a module explicitly. It is always assumed to be "required".

Note that you may see the use of phrases such as "core APIs" or "core packages" in problem statements or options of the questions in the exam. You should mark such options as incorrect. On the other hand, the phrase "foundational APIs" is used by the official API description and is, therefore, correct. So, for the purpose of the exam, `java.base` defines all the "foundational APIs" and not all the "core APIs" of the Java platform.

There is a "core Java platform" though. It is defined by Java Platform, Standard Edition (Java SE) APIs, which are defined in multiple standard modules (and not just in the `java.base` module).

16.5.3 Benefits of the modular JDK

The following is a list of benefits of a modular JDK as described by Java Enhancement Proposal (JEP) 200. I am quoting important points from this document because the exam has questions containing statements from it. If you have time, you should go through this document, otherwise, the following list should suffice.

1. It makes the Java platform more easily scalable down to small devices. Since Java classes and packages are distributed into modules, it is possible to create a configuration with a small footprint by packaging just the modules that are required. For example, if an application does not require any GUI, there is no need to package the `java.desktop` module with it.
2. It improves security and maintainability. Before modules, all members of all classes, irrespective of their access modifiers, were accessible through reflection API. Now, by encapsulating packages into modules, the JVM is able to restrict reflective access only to packages that explicitly allow it. The use of explicit exports and requires clauses prevent inadvertent dependence on classes that are not meant to be used outside a module.
3. It improves application performance by not requiring unnecessary classes to be packaged and/or loaded.
4. It is backward compatible. It is not required for an application to be structured as modular to be run on the modular JDK. A non-modular application will still run just like before on the modular JDK using the `-classpath` option.
5. Customization is possible. It is possible to make the packages that are not exported explicitly in the module descriptor accessible through the use of appropriate command line options. For example, if a module does not export a package, you can still use the `--add-exports` option to make it accessible to other modules. Of course, this option breaks the encapsulation

mechanism and is undesirable. It should be used only in exceptional circumstances.

As I mentioned earlier, Java modules is a fairly large topic. There are several important concepts such as module graphs, dependency analysis, and using non-modular jars that I have not talked about. The reason is that these concepts require a lot of time to discuss, while the OCP Java 11 Certification Part 1 exam merely scratches the surface of this topic. You will, at the most, get one question on it, which you will be able to answer based on the information given in this chapter. For this reason, it is best to ignore it at this time but you will be required to study this topic thoroughly for the Part 2 exam.

16.6 Exercise

1. Create a module named `compoundinterest` containing a class named `CompoundInterestCalculator` similar to the `simpleinterest` module described in this chapter.
2. Compile the source code of `compoundinterest` module with and without using the `--module` switch. Use `-d` option to direct the output to your output directory.
3. Run the `CompoundInterestCalculator` class with and without using the `--module-path` switch.
4. Enhance your module by making the `CompoundInterestCalculator` class implement the `InterestCalculator` interface of the `calculators` module.
5. Package `compoundinterest` module into a jar and run this module using the `--module-path` switch.
6. Create a module `finance` with a class `TestClass`. It should have a main method that computes simple interest as well as compound interest. Make appropriate modifications to `simpleinterest` and `compoundinterest` modules for this purpose.

Chapter 17 Understanding Java Technology and environment

- Describe Java Technology and the Java development
- Identify key features of the Java language

17.1 Java Technology and key features of the Java language

After going through the previous chapters, you already know all you need to know about this objective. I will just summarize the important points here.

Key points on Java Technology and the Java Development

1. **Platform Independence** - Java code is compiled into **Java bytecode**, which is interpreted by a virtual machine called the Java Virtual Machine (JVM). JVM is available for multitude of platforms (CPU+OS architectures). This means that the bytecode can run on all those platforms without any change. Thus, you do not need a Java compiler for every platform on which you want to run a Java class. The a class file produced on one platform will run without on any other platform as it is, if there is a JVM for that platform. If there is no JVM for a particular platform (for example, Android or iPhone), you cannot run a Java program on that platform.
2. **Java Installation** - Java is not a part of the Operating System. It is an application itself and is installed separately on top of the Operating System. A separate installable is available for most main stream desktop OSs such as Windows, MacOS, and Linux. As of now, the Java platform is not available for Android and iOS.

3. **JRE vs JDK** - The Java Runtime Environment (JRE) includes just the class libraries and executables that are required to run a Java program while the Java Development Kit (JDK) includes the tools such as the Java compiler and the Java debugger that are required to develop a Java program in addition to a JRE. Thus, it is not necessary to install the JDK if you just want to run a Java program but installation of the JDK requires the installation of the JRE.
4. **Development tools** - Java Development Kit (JDK) comes bundled with several applications. Some are pretty much required such as the compiler (javac) and some are useful while development such as debugger (), class inspector (javap), and, documentation generator (javadoc). It also comes with the JVM (java) for most common platforms.
5. **IDE** - An Integrated Development Environment (IDE) is an additional third party tool that makes Java development easier. It contains several tools on top of the tools available in the JDK such as code editor, syntax highlighter, and source organizer. However, an IDE is not essential for developing a Java program. Netbeans, Eclipse, and IntelliJ are some commonly used IDEs.
6. **Java Platform configurations** - Due to modularization of the JDK, it is now possible to split the Java runtime as well as library into configurations of varying features and sizes. While the core Java features are bundled in a readymade configuration named as the Java SE platform, a minimal JRE image for your application can also be created using just the modules that are required by your application.

Key Features of Java -

1. **Object-Oriented** - Java has features such as classes, objects, and access control, that allow you to do object-oriented development. It eliminates some non-OO features such as standalone functions. The following are some of the "object-oriented features" of Java:
 1. **Encapsulation** ensures that classes can be designed so that only certain fields and methods of an object are accessible from other objects. Java allows precise access control by marking data members

as public/protected/private (or default), which promotes encapsulation.

2. Java allows a class to extend at most one class but allows a class to implement more than one interfaces. Java supports **multiple inheritance of type** but does not support **multiple inheritance of state and implementation**.
 3. **Polymorphism** ensures that at run time the method to be executed depends on the actual object referred to by a reference. If a subclass overrides a method of a base class and if the object referred to by a variable is of type subclass, then the subclass's version of the method is used even if the declared type of the variable is of base class. This is also called dynamic binding. Java supports dynamic binding and polymorphism.
2. **Huge standard library** - Java Runtime Environment includes a huge set of readymade classes are useful for a wide range of applications such as networking, file and database I/O, text processing, data structures, and so on. It includes implementation for several networking protocols such as HTTP and protocols for communicating with a database server. It does not include a database engine though.
 3. **Less Complex** - Java eliminates a lot of complicated programming constructs to make it the code less prone to errors. For example, Java does not have pointers, multiple class inheritance, operator overloading, goto, and pragmas.
 4. **Garbage collection** - Java frees the developers from actively coding for garbage collection. It performs checks in the background that identify unused object and cleans them up. Garbage collection is done on low priority and so, it does not affect performance.
 5. **Secure** - A Java application can be run with a security manager. This security manager can be customized to allow precisely only those operations that you want to allow for an application. Third party Java applications downloaded from the internet can be run within a sandbox. This limits the operations that a program can do on the host machine.
 6. **Multithreading** - Java makes developing multithreaded applications a lot

easier than other languages. It offers lower level language constructs such as monitors as well as higher level multithreading API such as Fork/Join framework that helps you develop multithreaded and concurrent programs. However, Java does not parallelize code execution automatically on its own.

7. **High Performance** - Java used to be criticized a lot at one point in the past for its poor performance. However, that was a long time ago. Java interpreters are now highly optimized and deliver almost similar performance as a native application. One of the most important features of Java interpreters that contributes to the high performance is their ability to monitor and optimize code blocks that are executed frequently in an application.

Here are a few links worth going through for a more detailed discussion.

- Platform Independence - <http://stackoverflow.com/questions/2748910/how-is-java-platform-independent-when-it-needs-a-jvm-to-run>
and <http://functionspace.org/topic/1689/Why-is-Java--called-a-platform-independent-language->
- Object-Oriented Concepts
- <http://docs.oracle.com/javase/tutorial/java/concepts/>
- Encapsulation - [http://en.wikipedia.org/wiki/Encapsulation_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming))

ENTHUWARE END USER LICENSE AGREEMENT

Go to www.enthuware.com/eula to access Enthuware ebook EULA.