

OCP Java SE 11 Programmer - Part 1 Exam Fundamentals
1Z0-815

Hanumant Deshmukh

Saturday 12th October, 2019
Build 1.9

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 455001
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 makes 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.

Lead Author and Publisher: Hanumant Deshmukh

Technical Editor: Liu Yang

Technical Validators: Aakash Jangid, Bill Bruening

Technical Proofreaders: Carol Davis, Robert Nyquist

Copy Editor: Lisa Wright

Book Designers: Fatimah Arif

Proofreader: Ben Racca

Typesetter: Lillian Musambi

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.

Saturday 12th October, 2019 Build 1.9

To my alma mater,

Indian Institute of Technology, Varanasi

Acknowledgements

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 Lockhart for designing the cover.

This book also wouldn't be possible without many people at Enthware, 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 Enthware.com and CodeRanch.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, Rajashekar 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 Raghuvveer 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
2. Fedor Lvovich Dobrotvorskii

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. 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 big believer in freedom of speech and expression and works on promoting it in his spare time.

You may reach him at support@enthuware.com



Contents At A Glance

Introduction	i
1 Kickstarter for Beginners	1



Contents

Introduction	i
0.1 Who is this book for?	iii
0.2 How is this book different from others?	iii
0.3 How is this book organized?	iv
0.4 Will this be asked in the exam?	vi
0.5 General tips for the exam	vii
0.6 Official Exam Details and Exam Objectives	viii
0.7 Feedback, Errata, and Reviews	xi
1 Kickstarter for Beginners	1
1.1 Key points in OOP	2
1.1.1 A matter of perspective	2
1.1.2 API	3
1.1.3 Type, class, enum, and interface	4
1.2 Why is something so	5
1.3 Declaration and Definition	8
1.4 Object and Reference	8
1.5 static and instance	12
1.6 Stack and Heap	13
1.7 Conventions	19
1.7.1 What is a Convention?	19
1.7.2 Conventions in Java	19
1.8 Compilation and Execution	20
1.8.1 Compilation and Execution	20
1.8.2 Running a single file source code program	23
1.8.3 Packaging classes into Jar	24
1.8.4 Compilation error vs exception at run time	26
1.9 Nomenclature	27
1.10 Java Identifiers	28



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 OCPJP-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 OCPJP-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](https://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 Enthware'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 -

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 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 Enthware, 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/peexam{}_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

1. Describe Java Technology and the Java development environment
2. Identify key features of the Java language

Working With Java Primitive Data Types and String APIs

1. Declare and initialize variables (including casting and promoting primitive data types)
2. Identify the scope of variables
3. Use local variable type inference
4. Create and manipulate Strings
5. Manipulate data using the `StringBuilder` class and its methods

Working with Java Arrays

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

Creating and Using Methods

1. Create methods and constructors with arguments and return values
2. Create and invoke overloaded methods
3. Apply the `static` keyword to methods and fields

Reusing Implementations Through Inheritance

1. Create and use subclasses and superclasses
2. Create and extend abstract classes
3. Enable polymorphism by overriding methods
4. Utilize polymorphism to cast and call methods, differentiating object type versus reference type
5. Distinguish overloading, overriding, and hiding

Handling Exceptions

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

Creating a Simple Java Program

1. Create an executable Java program with a main class
2. Compile and run a Java program from the command line

3. Create and import packages

Using Operators and Decision Constructs

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

Describing Objects and Classes

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

Applying Encapsulation

1. Apply access modifiers
2. Apply encapsulation principles to a class

Programming Abstractly Through Interfaces

1. Create and implement interfaces
2. Distinguish class inheritance from interface inheritance including abstract classes
3. Declare and use List and ArrayList instances
4. Understanding lambda Expressions

Understanding Modules

1. Describe the Modular JDK
2. Declare modules and enable access between modules
3. Describe how a modular project is compiled and run

0.7 Feedback, Errata, 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 <https://enthuware.com/forum> .

All confirmed errors are listed here: <https://enthuware.com/815/errata.php> .

Since this book is published on “print on demand” basis, an updated version, with all enhancements and fixes, is published every few weeks. you may check whether have the latest version by comparing build number mentioned title page of your copy shown in book description on <https://enthuware.com/815/amazon.php> . If you are using the Kindle ebook version and have an older version, you can get the most recent version by requesting Amazon support. They will send the updated version on your device upon your request.

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 <https://enthuware.com/815/amazon.php> 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



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 repeat over and over throughout this book. 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

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 here");
        }
    }
}
```

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 permit 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

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.

<http://stackoverflow.com/questions/11715485/what-is-the-difference-between-declaration-and-definition-in-java>

<http://www.coderanch.com/t/409232/java/java/Declaration-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

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

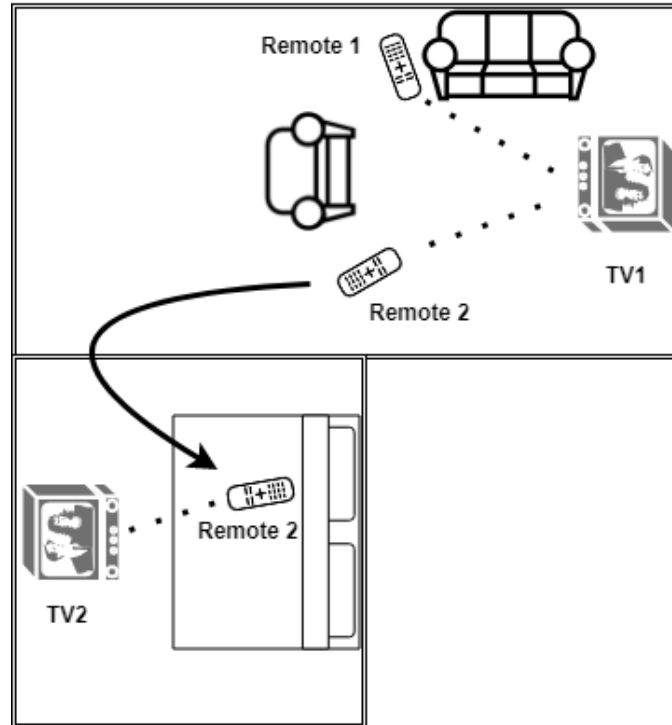
as a cookie cutter. Just as you create cookies out of dough using a cookie cutter, you create objects out of memory space using a class.

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 that object. The following picture illustrates the situation:



Relationship between an object and a reference

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.

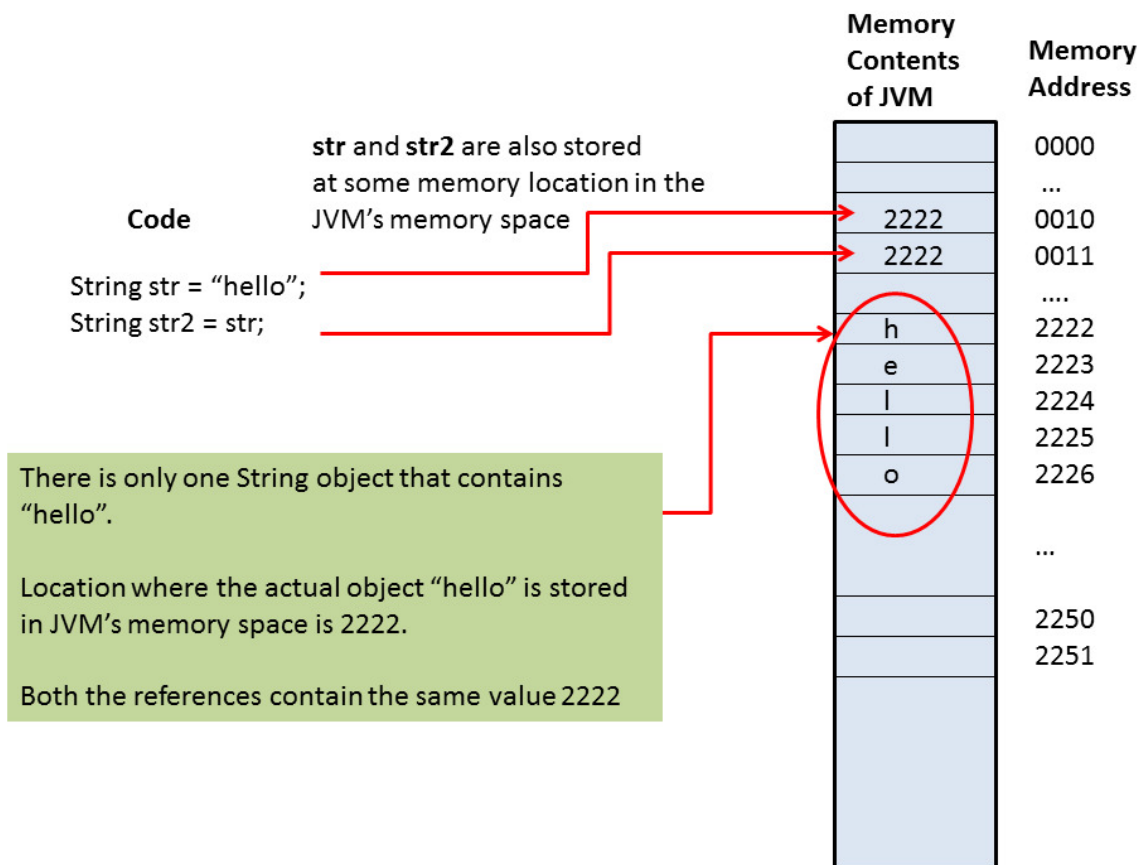
A program's memory can be thought of as a long array of bytes starting with 0 to `NNNN`, where `NNNN` is the location of last byte of the array. Let's say, within this memory, the object "hello" resides at the memory location `2222`. Therefore, the variable `str` actually contains just `2222`. It doesn't 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.



Object and Reference

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

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

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();
}
}

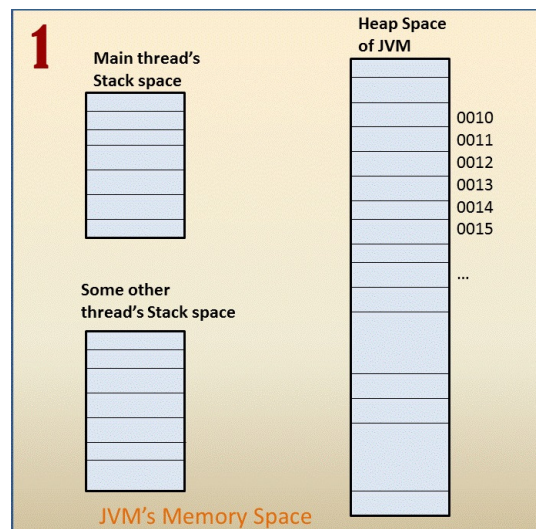
```

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.

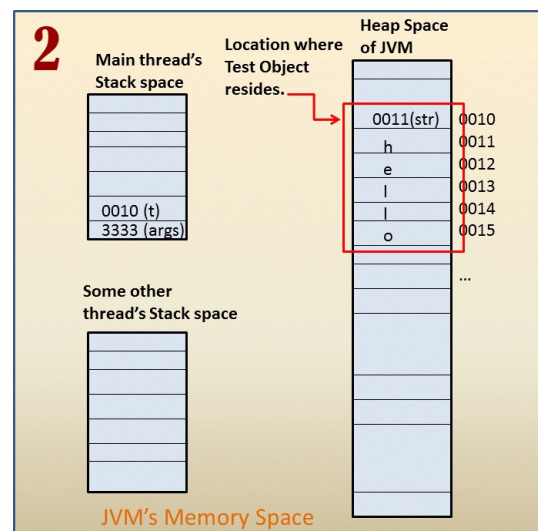
In Java, each thread is given a fixed amount of stack space. In the above example, when you execute the program, a main thread is created with a fixed amount of stack space. All this space is initially empty. This is represented by the following figure.



Step 1 - Stack and heap are empty

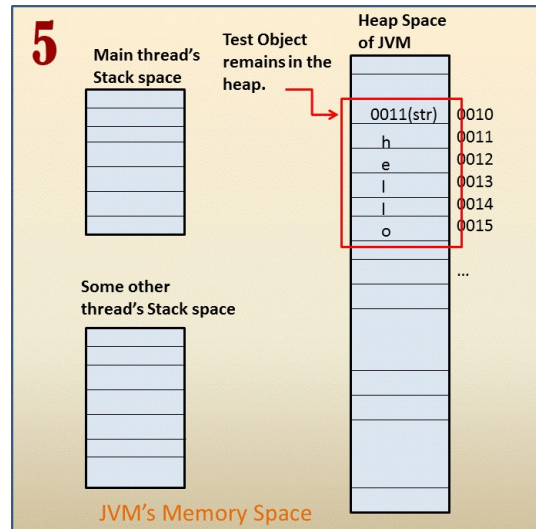
When this thread invokes the main method, all the temporary variables created by this method

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 a reference variables, they contain the address of the location where actual objects referred to by `args` and `t` respectively reside). Therefore, as the following figure shows, the stack fills up by the amount of space required by these reference variables.



Step 2 - Stack has two variables and heap has one Test object

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 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. This is represented by the following figure.



Step 5 stack is empty again but the heap still has Test object

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 as they please. JVM goes a great length to organize the heap space. Organization of heap space is an advanced topic and is very important when you do performance analysis of an application. But it is not relevant for certification exams and so I will not be discussing 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.

Note

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 when 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:

1. 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.)
2. 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.
3. 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**".
4. 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.
5. Only temporary variables i.e. variable 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.
6. When a method is invoked by a thread, it uses the thread's stack space to keep its temporary variables.

7. 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.

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.
 - (a) Class names start with an uppercase letter. For example, `ReadOnlyArrayList` is a good name but `Readonlyarraylist` is not.
 - (b) 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.
 - (c) 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.ReadOnlyArrayList`.

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: 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.Account
```

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:

```
c:
├── javatest
│   ├── Account.java
│   └── accounting
│       └── Account.class
```

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

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

Note

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.

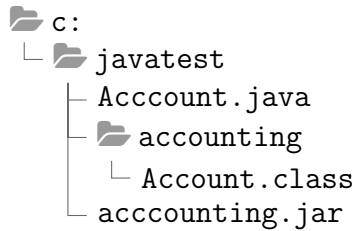
Creating a jar file that maintains the package structure of class files is quite easy. Let us say you have the directory structure shown below:

```
c:  
└─ javatest  
   └─ Account.java  
      └─ accounting  
         └─ Account.class
```

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
Main-Class: accounting.Account
Created-By: 11.0.2 (Oracle Corporation)
```

You can actually go ahead and create `mymanifest.txt` file with just one line `Main-Class: accounting.Account` in `c:\javatest` directory (make sure there is a new line at the end of the file) 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
```

Note

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

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 parentheses, braces, 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

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).

Note

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.

This is the end of sample. If you liked it, please purchase the full version from <http://enthuware.com/815/amazon.php>