



JAVA

ORACLE CERTIFIED FOUNDATIONS ASSOCIATE



OCFA

JAVA FOUNDATIONS EXAM 1Z0-811



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



Hanumant Deshmukh

EXAM STUDY GUIDE

eN
ENTHUWARE®

Java Foundations Exam Fundamentals

Exam Code 1Z0-811

Hanumant Deshmukh

Tuesday 28th May, 2024
Build 1.4

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 © 2020 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: Anil Kumar, 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.

Tuesday 28th May, 2024 Build 1.4

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 Enthuware, including Paul A Prem, who have been developing mock exams for the past fifteen years. Their experience helped fine tune several topics in this book.

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

I am also thankful to countless Enthuware.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, Rajashekhar Kommu, Kaushik Das, Gopal Krishna Gavara, Dinesh Chinalachiagari, Jignesh Malavia, Michael Tsuji, Hok Yee Wong, Ketan Patel, Anil Malladi, Bob Silver, Jim Swiderski, Krishna Mangallampalli, Shishiraj Kollengreth, Michael Knapp, Rajesh Velamala, Aamer Adam, and Raghuveer 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.

About the Author

Hanumant Deshmukh is a certified professional Java architect, author, and director of a software consultancy firm. 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 twenty 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 and his Masters in Financial Analysis from ICFAI. 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

CONTENTS

I believe you have already gotten 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. Not many people know about it but Oracle has an **entry level** Java certification named **Java Foundations Certified - Junior Associate (JFCJA)** with exam code **1Z0-811**, which is geared towards high schoolers, college goers, and Java beginners. Oracle has renamed it to **Java Certified Foundations Associate (JCFA)**. This exam hasn't been very popular because of the presence of the **Oracle Certified Associate Java Programmer** certification aka **OCAJP** with exam code **1Z0-808**, which is also meant for entry level Java professionals who want to make a career in Java programming. The OCAJP exam is also a prerequisite for getting the more advanced **Oracle Certified Professional Java Programmer (OCPJP 1Z0-809)** certification. Since the **JCFA** exam does not make one eligible to take the OCPJP exam, there was little benefit in taking this exam.

Oracle has changed the **Java certification paths** since Java 11 by eliminating the Associate level Java programming certification altogether. One just needs to pass one exam, 1Z0-819 or 1Z0-829, to get the Oracle Certified Professional Java 11 or 17 Developer certification. This is mostly a good thing because now one has to pay the price of just one exam to get the OCP certification. However, this has made the exam quite heavy with lots of advanced topics such as Concurrency, NIO, JDBC, Modules, Localization, and Annotations. It will be very difficult for an entry level candidate to master these advanced topics. Thus, appearing for the **1Z0-819** or **1Z0-829** exams directly will be a risky gamble for entry level Java programmers.

This makes the **Oracle Certified Foundations Associate (OCFA)**, Java certification very attractive now. If you are a high schooler or a Java beginner, the **1Z0-811** exam is the best way

to prove that you have learnt the basics of Java programming. This exam costs a lot less (only \$95) than the OCPJP exam (\$245). By preparing for the OCFA certification, you will get to learn the fundamentals and you will also get a verifiable certification to show on your resume, which will help in your job hunt. You can then proceed to prepare for the more advanced OCPJP certification.

The only issue with the OCFA certification is that it is still stuck on the old Java 8 version. It is possible that Oracle may update it for a new version of Java.

I have designed this book to help one master all the topics required to pass this exam in about two months with two hours of study time every day. Of course, if you are already an experienced Java programmer, you may be over with it in a few weeks. Since the exam is currently based on Java 8, I have kept the content close to Java 8. However, if a particular topic has been affected by the new developments in Java, I have mentioned the relevant changes as well.

About the mock exams

Mock exams are an essential preparation tool for achieving a good score on the exam. Even if you know all the topics mentioned in the official exam objectives or if you have a few years of Java development experience under your belt, the exam may still trip you up if you haven't practised answering multiple choice questions in a limited amount of time. 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 have never imagined. A bad quality mock exam will easily consume 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 <https://enthuware.com>. It is priced quite reasonably (only 9.99 USD) and has stood the test of time.

The combination of this book and Enthuware mock exams should be enough for you score above 90%.

0.1 Who is this book for?

This book is for Oracle Certified Foundations Associate Java certification exam (Exam Code 1Z0-811) takers who know a little bit about computers and programming in general. You don't need any prior Java knowledge. Some programming experience will be helpful but is not required. Many people who have not programmed earlier find out while reading their first programming book that they have an intuitive understanding of logic. They take to programming like fish to water. So, don't be afraid of jumping straight into Java programming even if you are a complete beginner. Since this book is focused on Java, I will not teach fundamentals of programming or algorithms in this book as such, but I will go through enough basics that you will be able to write simple Java programs in no time.

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.

1. Do you know what Operating System, RAM, and CPU are?
2. Do you know what a command line and File Explorer are?
3. Can you create and locate a file or a folder on your machine using File Explorer?
4. Can you execute or launch a program or a software by double clicking on the executable on your machine?
5. Can you open a text editor such as Notepad and create text files on your machine?

It would be good if know all of the above. If not, I would suggest you to quickly go through a general computer book for beginners first, and then come back to this book. I recommend **Michael Miller's "Computer Basics Absolute Beginner's Guide"** for this purpose. 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?

Twenty years ago, when I first got into Java programming, there weren't that many books written on Java. But with Java now being the most popular programming language, there are literally thousands of books, from beginners to advanced, written on Java. However, while teaching a batch of student for the **Oracle Certified Foundations Associate (OCFA)** exam, I couldn't find one that was focused on this exam. I checked out several Java beginners books but most of them seemed way out of touch with current state of Java development. One of the beginners books spent quite a few number of pages on Applets, something that has been completely out of favor for more than fifteen years. Some spent too many pages on the history of Java, something that is interesting to know as a story but adds little value to the knowledge base of the reader. But more importantly, besides perpetuating technically imprecise clichés, most of them missed explaining fundamental aspects of the Java language and Java development. My goal in this book is to make sure you learn

the basics of Java really well and, of course, to prepare you well for the OCFA exam. Thus, 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 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. **No wasting time** - My goal in this book is to teach you the basics of the Java language and not to teach the history of Java language or the past and the future of the Java language! I will not be wasting your time on stuff that is not important from the perspective of the exam.
5. **No Assumptions** - I do not assume that you have any prior knowledge of Java or any other programming language. I believe that you don't have to understand the programming

philosophy of other languages such as C/C++ before learning Java just because Java came a long time after them. Comparisons with other languages are helpful to those who already know other programming language(s) but it is less confusing for a new programmer to learn Java without worrying about the features of other languages.

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.

Please note that colors are visible only in the eBook/Kindle version. The paperback version is printed in black and white with colors displayed as shades of gray.

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 General tips for the exam

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

1. **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. Some people believe that picking the same option, for example option 2, for all such questions is a good idea.

There is **no credit for partial answer** either. You must select all correct options to get the credit for answering that question correctly.

2. **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.
3. **Mark and Move** - You have to answer a fixed number of questions (75) in a fixed amount of time (150 minutes). That means, you have about two minutes to answer each question. So, one approach could be that you move to the next question after every 2 minutes. However,

this is not a very good approach. The exam has easy as well as tough questions. Some topics (such as loops) are, by nature, very hard and require a lot more time to answer. Furthermore, you never know which type of question will you get at the beginning of the exam. If you get a few time consuming questions at the start, you will start feeling the time pressure and that will affect your ability to answer on the subsequent questions. It is very easy to answer a simple question incorrectly when you are stressed. A better approach is to take a quick look at the question and judge whether you can answer it quickly with confidence. If not, just mark the question (there is a checkbox for marking a question on the top right corner) and move to the next question. This way, you will be able to answer all easy questions without pressure. The goal here is to make sure that you don't trip on the easy questions. Once you are done with the last question. Go to the Review screen and start solving the "marked" questions. You will now have enough time to solve the tough and time consuming questions because you never took full 2 minutes on the easy questions. Even if you are not able to answer a few questions at the end, you know that they were tough questions and there was a high chance of answering them incorrectly anyway.

4. **Read all options** - Sometimes, one feels so, confident about an option that they select it believing it to be correct and move on to the next question. However, the questions and the options in this certification exam can sometimes be tricky. The options are designed to confuse you and therefore, it is absolutely critical to read all options even if you feel you have found the right option. It is better to take a moment to rule out the rest of the options.
5. **Must/Cannot/Never/Always** - You should be very careful about selecting an option that has such strong words because there is almost always an exception to a rule. If you see two options where one says you "cannot" do X, while the other says, you "may" do X, and if you have no idea about the topic, select the second option because there is a a good chance that a workaround to do X exists.
6. **Do not fight the question writer** - Questions are developed by humans and it is possible that you may find a question to be ambiguous or incorrect. You may also feel that more than one options are correct. In such situation, pick the option that you feel is the best. Do not go with the option that you feel is correct in an obscure situation. Go with the most common scenario.
7. **Ignore minor/trivial errors** - Although all information given in the question is important, you may encounter a question with an obvious typo or mistake. It is best to ignore such errors because the exam doesn't test you on such typos. The exam is tough but not unfair. For example, if you see System.out.println misspelled as Sytem.out.println, ignore that mistake.
8. **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.

9. **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.

Do not fret over the missing stuff. Assume that all such things such as the source code file name, the directory, classpath/path, are valid. Just focus on the code that is given and assume that everything else is irrelevant and is not required to arrive at the answer.

10. **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 your studies!

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

0.5 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-foundations/pexam_1Z0-811 during your preparation.

Exam Details

Duration: 120 Minutes

Number of Questions: 60

Passing Score: 65%

Format: Multiple Choice

Exam Price: USD 95 (varies by country)

Exam Objectives

What Is Java?

1. Describe the features of Java
2. Describe the real-world applications of Java

Basic Java Elements

1. Identify the conventions to be followed in a Java program
2. Use Java reserved words
3. Use single-line and multi-line comments in Java programs
4. Import other Java packages to make them accessible in your code
5. Describe the `java.lang` package

Working with Java Operators

1. Use basic arithmetic operators to manipulate data including +, -, *, /, and %
2. Use the increment and decrement operators
3. Use relational operators including ==, !=, >, >=, <, and <=
4. Use arithmetic assignment operators
5. Use conditional operators including &&, ||, and ?
6. Describe the operator precedence and use of parenthesis

Working with the Random and Math Classes

1. Use the Random class
2. Use the Math class

Using Looping Statements

1. Describe looping statements
2. Use a for loop including an enhanced for loop

3. Use a while loop
4. Use a do- while loop
5. Compare and contrast the for, while, and do-while loops
6. Develop code that uses break and continue statements

Arrays and ArrayLists

1. Use a one-dimensional array
2. Create and manipulate an ArrayList
3. Traverse the elements of an ArrayList by using iterators and loops including the enhanced for loop
4. Compare an array and an ArrayList

Java Methods

1. Describe and create a method
2. Create and use accessor and mutator methods
3. Create overloaded methods
4. Describe a static method and demonstrate its use within a program

Java Basics

1. Describe the Java Development Kit (JDK) and the Java Runtime Environment (JRE)
2. Describe the components of object-oriented programming
3. Describe the components of a basic Java program
4. Compile and execute a Java program

Working with Java Data Types

1. Declare and initialize variables including a variable using final
2. Cast a value from one data type to another including automatic and manual promotion
3. Declare and initialize a String variable

Working with the String Class

1. Develop code that uses methods from the String class
2. Format Strings using escape sequences including %d, %n, and %s

Using Decision Statements

1. Use the decision making statement (if-then and if-then-else)
2. Use the switch statement
3. Compare how `==` differs between primitives and objects
4. Compare two String objects by using the `compareTo` and `equals` methods

Debugging and Exception Handling

1. Identify syntax and logic errors
2. Use exception handling
3. Handle common exceptions thrown
4. Use try and catch blocks

Classes and Constructors

1. Create a new class including a main method
2. Use the private modifier
3. Describe the relationship between an object and its members
4. Describe the difference between a class variable, an instance variable, and a local variable
5. Develop code that creates an object's default constructor and modifies the object's fields
6. Use constructors with and without parameters
7. Develop code that overloads constructors

0.6 Feedback, Errata, and Reviews

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/811/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 you have the latest version by comparing build number mentioned title page of your copy shown in book description on <https://enthuware.com/811/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/811/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

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 Programming and Programming Languages

We receive "instructions" all the time. When we are kids, we receive instructions from our parents. When we are in school, we receive instructions from our teachers. When we grow up, we receive instructions from our team leads, managers, and CEOs, customers, employers, and so on. One obvious fact about receiving instructions, which is very important in this discussion, is that we receive instructions in the language that we understand. Indeed, why would anyone give us instructions in a language that we don't understand? Most of the time, we follow the instructions and we do as we are told. But sometimes, we don't follow them. Either because we don't understand them, or because we are not able to do them. Well, sometimes we just don't want to do them but let's leave that case for now :)

In the computer world, the term used to describe the act of providing instructions is "programming" or "coding" and the set of instructions is called a "program" or the "code". Thus, when parents instruct children on how to behave, you could say that parents are "programming" the children, or when a teacher gives instructions on how to answer a test, you could say that the teacher is "programming" the students. If the language of instruction in the above situations is English, then English is the "programming language"!

The relation between a computer and a programmer is very similar. A programmer instructs a computer to do something in a language that the computer understands. In other words, a programmer programs a computer using a "programming language", which the computer understands. If the instructions given by the programmer are understood by the computer and are within the capabilities of the computer, then the computer will "execute" them and the outcome will be as the programmer expected. If you have ever followed a recipe from a cookbook, you know what I am talking about. You are being programmed by the author to produce the dish. From your perspective, you are simply executing the instructions given in the recipe. You trust the author to give you the right instructions and, upon executing those instructions exactly as given, you can expect the dish as the outcome.

But the fact remains that computers are not humans. They are just electronic machines with a very limited set of things that they can do. Furthermore, the only "language" that computers understand is "machine language", the language that has only two alphabets - 0 and 1, and which very few humans can converse in easily. So, how are most people going to interact with a machine whose language they can't speak? This situation is no different from the one where you go to a restaurant in a foreign country and you don't speak the language that the waiter understands. That's where a translator (also known as an interpreter) comes into the picture. A translator understands two languages - your language and the waiter's language. He translates your instructions into the language that the waiter understands. Similarly, to help those who can't speak machine language, the people who understand machine language create "translators", which convert instructions written in a more human understandable language into instructions in the machine language. The technical terms for such translators are "compilers" and "interpreters". There are subtle differences between a compiler and an interpreter but you don't need to worry about that right now.

The languages that most programmers are familiar with and are able to write the instructions in are called "high level languages". You must have heard of Java, C/C++, Python, and JavaScript. They are all high level languages. They are called high level because they use words from a human language (English) and are a lot easier to read and write for humans. But they cannot be understood

by a machine. The instructions written in a high level language have to be translated into a "low level language", which the computer can understand.

The following figure summarizes the above discussion. You have a programmer, who writes instructions in a high level language, a translator, which converts the instructions in a machine language, and a computer that executes the instructions and produces the output expected by the programmer.

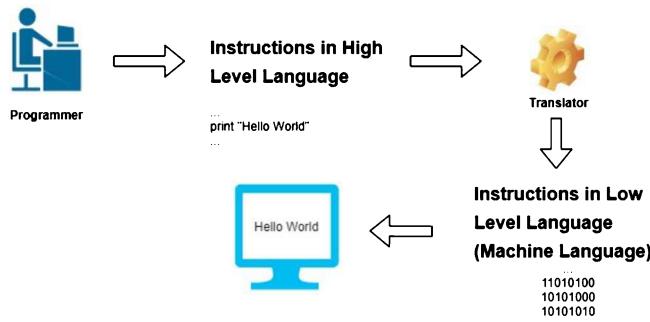


Figure 1.1.1: Steps in writing and executing a program

The above diagram illustrates the basic steps in programming and execution. However, this diagram doesn't exactly match what we observe in real life. Specifically, the steps where the programmer writes a program, and translates it using a translator, do not happen outside the computer. We see everything happening in the computer itself and that causes confusion in students who are just beginning to learn programming.

A long long time ago, in 1950s, and 60s, these steps used to occur outside a computer. At that time a computer was just a simple microprocessor chip stuck on circuit board. There was no screen or even a keyboard. A programmer would write the code directly in the machine language by punching holes on a piece of paper cards and then feed those cards through a special device attached to the microprocessor. Later on, microprocessors became a little more powerful and started supporting monitors and keyboards. The programmers would then feed the code into the computer using keyboards. This is when the microprocessors based devices started to look like the computers that you see today. So, while a microprocessor is still the heart of a computer, you need other peripheral devices such as memory bank, a monitor and a keyboard to harness the power of the microprocessor. If you open up your phone, laptop, or computer cover, you will see a board on which a lot of chips as well as devices are stuck. In the old times, this board and the chips used to be a lot larger in size but the basic architecture of the computer has remained the same.

Very soon, the execution power of computers increased due to advancement in electronics and people started using computers to do a variety of things ranging from word processing to data management. The set of instructions required to program a computer to do these things increased exponentially and it was practically impossible to write so much of machine language code and feed it to the computer manually. This led programmers to write helper programs that would accept instructions in a high level language and convert them into the machine language. These helper programs are called "compilers" and "interpreters". Similarly, they wrote helper programs that would take the machine level code and feed it to the microprocessor. The program that manages

the microprocessor and the devices connected to the microprocessor is called "Operating System". Windows, Android, Linux, MacOS, are all operating systems.

So, instead of dealing with the microprocessor directly, programmers take help of these helper programs to interact with the microprocessor. Both - compilers/interpreters, and the Operating System, reside in the memory of the computer itself. In other words, the basic steps shown in the above diagram still happen, but they performed by the programs running inside the computer.

1.2 Applications

In its early days, programming involved writing all the instructions for achieving a particular task in a file and submitting that file to the operating system for execution. As tasks became more and more varied and complicated, programs became too huge to maintain in a single file. Programmers started organizing their code into multiple files for ease of management and also so that they could reuse parts of it for performing other tasks. For example, if a program that printed account details were split into two separate parts where one part would just do the account management and the second part would just do the printing, then the printing part could be reused in another program that printed salary details. Such reusable parts are packaged separately and are called "libraries". A program, therefore, is composed of new code as well existing code present in a library.

With the advent of powerful operating systems such as Unix and Windows, programmers started writing programs for people who had no idea about programming but who could use the power of computers for performing their routine tasks such as writing documents and letters, maintaining their daily expenses, tracking their investments, and so on. Such ready to use programs are called "applications" or "apps", in the world of smartphones and portable devices. Microsoft Word, PowerPoint, and Google Chrome browser are examples of commonly used applications. You need to have these applications "installed" on your device before you can use them. Once installed, you can ask the operating system to execute that application, usually by clicking on an icon. You can also execute it by specifying the name of the application on the command prompt (or shell prompt on Mac/Linux).

Applications that run on desktop or laptop computers are called Desktop applications, while those that run on mobile devices are called Mobile apps. TicToc, Contacts, and Messaging are examples of mobile apps. A lot of applications these days can also be accessed by opening websites on the browser. Such applications are called "web applications". For example, GMail is a web application that you use by browsing to gmail.com.

Applications can be huge or they can be small. For example, an inventory management application might involve a large number of views with various options while a calculator application might show just one view. An application such as Volume Manager may even run as long as the computer remains switched on, completely in the background without showing any window to the user or it may run for a moment upon the user's request, do its job, and then terminate. It all depends on what an application is meant to achieve.

From a programmer's perspective, all applications are just programs written in one programming language or another.

1.3 Application Programming Interfaces ↗

In the previous section I explained how programs are usually split into multiple parts so that some of the parts can be reused in other programs. The parts themselves may not necessarily look like applications with which humans can interact directly but they are programs nonetheless and they implement some functionality that can be used by other programs. For example, a program that interacts with the printer can be used by any application that requires something to be printed. If you ever attached a printer to your computer, you must have seen your computer trying to find a "driver" that can talk to the printer. This "driver" is nothing but an independent program that knows how to interact with that printer. Once the driver is installed, any application on that computer is able to use that printer automatically. How does that work? How is an application that knows nothing about a new printer able to use that printer to print?

Well, when you try to ship a package through the postal service do you know or care how exactly does the postal company deliver the letter to the destination? You simply wrap your package, put on the destination address label, fill out the required forms, and give it to the company representative at the counter. The rest is taken care of by the company. Basically, the company provides you with a simple "interface" to interact with them. Thus, shipping is a simple to use service but there are thousands of big and small mail order businesses that use the shipping service.

The same thing happens in software. A program that provides a service exposes an interface for other programs to use that service. This interface is called an **Application Programming Interface (API)**. If you are trying to develop an application that requires printing, your application would use the API provided by the driver to interact with the printer.

Just like a mail order business that uses third party companies for advertising, packaging, and shipping its products, you would focus on implementing the business logic of what you actually want your application do to and try to use third party services for tasks that are not unique to your application. Instead of writing everything from scratch, you would use the software libraries provided by someone else for such tasks. Some common services that applications usually require are interacting with files, interacting with the screen to show the user interface, interacting with mouse/keyboard, and communicating with other websites. All these services are implemented by someone other than the application developer. The application simply makes use of the APIs provided by the service providers to use their services. You may wonder at this point how exactly is an application developer supposed to know about the API provided by a software library. Simple. Documentation! If a library provider wants other people to use their library, they have to provide some kind of documentation explaining how exactly other people can use their library.

The reason I am talking about APIs is that a large part of gaining expertise in Java programming is gaining expertise in using the APIs provided by the Java platform. As you will soon see, Java comes with a huge library of reusable components, which makes developing Java applications easy as well as quick. But to harness the power of Java libraries, you must be mentally prepared to read a lot of documentation.

Exam Objectives

1. Describe the features of Java
2. Describe the real-world applications of Java

2.1 Understanding Java ↗

You may have heard the story of the five blind men who tried to describe an elephant after observing it by touching. Since none of them had a complete view of the elephant, all of them described the elephant differently based on the part they touched. None of them were entirely incorrect in explaining what they observed but their understanding of the elephant was entirely incorrect. Thus, they deemed the elephant to be a completely useless animal. The situation with Java is similar. Java is not just one thing but a collection of multiple things. The word Java is also often used to mean different things depending on the context. Unless you understand the big picture, it will be difficult for you to take advantage of all the features that it provides. So, let me first explain all the different things that Java means.

Programming Language - First and foremost, Java is a programming language. Like all programming languages, it has its own set of keywords, syntax, and rules.

Java Development Kit - As explained before, code written in a programming language needs to be translated into instructions that can be understood by the computer. From that perspective, Java may refer to the tools such as compiler, debugger, and disassembler that help you write Java code. Collectively, they are called the Java Development Kit (JDK).

Java Virtual Machine - You may have heard of and seen different types of computers with different operating systems. You may be using a windows laptop, while your friend might be using an Ubuntu desktop. Your smartphone running Android or iOS is also a computer. All these machines have different architectures, different operating systems, and different features. You can't just copy an executable program such as Word from your windows machine to your MacBook and expect it to run. If you want to execute a program on any of these machines, you have to first translate the source code of that program into the machine language understood by that particular type of computer. Thus, you will need multiple translators to run your code on multiple machines and even then, due to differences in features, the resulting machine code may not work exactly the same on all the machines.

This is where the Java Virtual Machine (JVM) comes into picture. JVM is also a computer. Just like a regular computer, it has its own devices and features and it understands its own machine language. But unlike a regular computer, it is not a physical computer. In other words, it has all the virtues of a real computer but it does not exist physically, which is why it is called a "virtual" machine. A JVM is, in fact, a software program that mimics a computer and runs on top of a physical computer. Java designers identified the common features of regular physical computers and created a software program that simulates all those features. They translated this program into machine languages of all commonly used physical computers so that it can run on all those physical computers. Internally, the JVM uses the features of the physical computer on which it executes. Different types of computers require different JVMs but all such JVMs provide a single standard uniform interface.

So now, instead of worrying about different types of computers and instead of translating

your program into multiple machine languages for different computers, you just have to translate your Java code into the machine language understood by the JVM. You need to understand that instead of the computer executing your machine code directly, it will be the JVM that will be executing the machine code now. Furthermore, since a JVM is specific to a particular type of computer, it knows how to translate those instructions into the machine language understood by that particular type of computer.

Although Java designers have developed JVMs for all commonly used computers, there are still some types of computers for which there is no JVM. Most notably, there is no JVM available for Android and iOS based computers. The reason is that Android and iOS based systems do not have much in common with regular desktop computers. Therefore, it is not possible to run Java programs written for desktop computers on Android/iOS based smartphones. But the way technology is progressing for smartphones, it is entirely possible that there might be a JVM for these devices as well in future.

Java bytecode - The machine language understood by the JVM is called Java bytecode. If you want to run your program on the JVM, you must first "compile" that program into instructions written in Java byte code using the Java compiler provided by the Java Development Kit. The JVM interprets these bytecode instructions and further translates them into the machine code understood by the physical computer. Finally, the JVM uses the features provided by the Operating System to execute the machine code instructions. The reason why this machine language code is called "byte" code has to do with a bit of computer science theory, which is not important at this stage.

Java Class Library - As I mentioned before, new applications can be developed quickly if you reuse existing code instead of writing everything from scratch. Java comes with a huge library of code for common tasks that are required by Java applications. ranging from File manipulation to Graphical User Interfaces (GUI), from Networking to Databases, and from Security to Multi-threading. This library is called the Java Application Programming Interfaces (Java APIs). All the code in this standard library is organized in various "packages" package. For example, you can actually develop a complete database driven GUI application in a few lines of Java code by using the Swing package. The total size of your code turns out to be very small only because a large amount of code that already exists, is reused.

Java Runtime Environment - The Java Virtual Machine and the pre-compiled collection of the Java class library is called the Java Runtime Environment (JRE). Most desktop computers come with the a preinstalled but if a computer does not have it, one can easily download and install it for free. If a JRE is available on a particular machine, you can rest assured that your application will work the same on that machine without any change. Actually, there is a term for this - **WORA** - Write Once Run Anywhere! The JRE makes WORA possible. This is a big deal because before Java, programmers had to maintain different versions of their programs for different users depending on what type of computers they were using. They also had to test their programs on all these different platforms because there was no guarantee that even after compiling their program for a particular computer, it will behave the same. In practice, however, due to fundamental differences between different types of computers, a Java program might run a little

differently on them. Java GUI applications, in particular, do not look exactly the same way on Windows, Mac, and Linux computers.

It is important to understand that all of the above aspects of Java are independent of each other. For example, the Java language is used to develop Android applications even though there is no JVM or the JRE for Android. This is possible because Android uses the programming language aspect of Java and provides its own tools and libraries to translate Java code into the machine code that Android understands. The Java Virtual Machine, on the other hand has no dependency on the Java programming language. It is possible to write programs in several other languages such as Groovy, Jython, Kotlin, and Scala, and compile them into Java byte code. Of course, there will be different compiler programs for compiling code written in other languages, but once the code is compiled into bytecode, it does not matter which language was used to write the source code. Furthermore, some low powered consumer devices contain a JVM but do not contain the JRE. Since all these parts of Java work independently, it is possible to use whichever piece is suitable for a given purpose. This is one of the reasons why Java has been so successful.

The following figure illustrates the relative position of a JVM among various components of a computer.

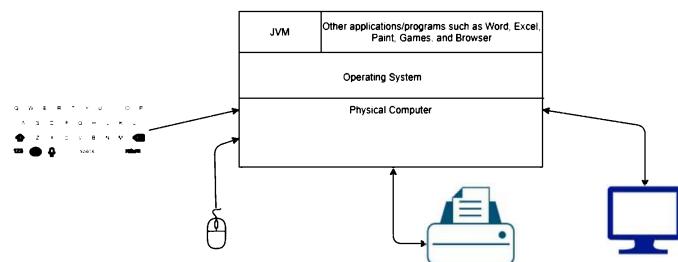
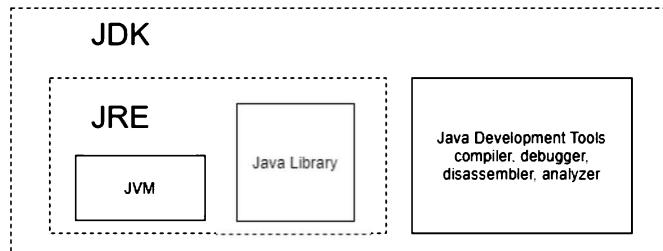


Figure 2.1.1: Relative position of the JVM, the OS, and the physical computer

In the above figure, observe that the JVM is just like any other application program running on a computer. Like all application programs, it takes help from the Operating System to use the resources of the physical computer.

The following figure explains the relation between JDK, JRE, and JVM.



JRE = JVM + Java Standard Class Library

JDK = JRE + Development Tools

Figure 2.1.2: Components of a JDK and a JRE

Observe that the JVM and the Java standard class library (Java API) are collectively known as the JRE. Executing a Java program on a machine requires JRE. While developing Java programs on a machine requires development tools such as the Java compiler, which is provided by the JDK.

While the JDK includes basic development tools, many third party organizations provide even more advanced tools such as Integrated Development Environments (IDEs), dependency analyzers, and UML designers. These tools make software development in Java easier by highlighting keywords, syntax errors, and many such features. Commonly used IDEs are Apache Netbeans, Idea IntelliJ, and Eclipse. Of course, they are all optional.

Java Platforms - A Java Platform refers to a combination of JVM, Runtime Environment, and Development tools. Java has grown a lot over the years and it now contains so many features that the Java designers felt it necessary to create different packages for different needs. So, for basic needs, you have the **Java Standard Edition (Java SE)**. This package includes the JVM, Standard Class Library, and some development tools. This package is sufficient for developing standalone Java applications. For large enterprise applications, you have **Java Enterprise Edition (Java EE or JEE)**, which contains a server JVM, the Standard Class library, the Enterprise class library, and development tools. At one time, there was a package for mobiles and other electronic devices as well. It was called **Java Micro Edition (JME)**! All of these are Java platforms.

A Java platform is standardized for all devices. So, for example, if a machine or a device supports Java SE, then you can be assured that it supports all the components of the Java SE platform. Thus, if you develop an application on the Java SE platform, it will work the same on all machines that support the Java SE platform.

An important thing to mention here is that Oracle keeps tweaking the nomenclature and the actual contents of a Java platform as per the needs of the industry. For example, Java EE standard is now managed by a separate organization called Jakarta. So, Java EE is now called **Jakarta EE**!

Java versions - Java 1.0 was released in 1996 and as of this writing (Sep 2020), version 15 is the most recent version of Java. In between, there have been regular updates to all the

aspects of Java. Java 8, on which the JFC-JA 8 Certification (1Z0-811) is based, was released in 2014. Java 8 was a monumental version because a lot of new features and path breaking changes were done in this version. A majority of companies still run their software on Java 8. I will also be focusing on Java 8 in this book but I will also mention important changes that have happened to the Java platform after Java 8 occasionally.

Java Community Process - Early on (in 1998), an independent organization of various Java users (companies as well as developers) was formed to develop standard technical specifications for various Java technologies. This organization is named Java Community Process (JCP). JCP ensures that new features and changes are made in an organized fashion. It prevents Java from fragmentation into different flavors. For example, you will not have a Microsoft Java and an Oracle Java. There is always just one flavor of Java. Any one can implement their own Java compiler, JVM, and any other Java tool but they all must adhere to the standards approved by the JCP.

If all this terminology sounds too confusing, don't worry. It is confusing. Even for senior developers. You will get the hang of it in due course of time.

2.2 Features of Java ↗

Now that you know various components of Java, it will be easier for you to understand the features of Java. In fact, several features are a direct consequence of the Java platform architecture itself. For this reason, I have categorized the features according to different perspectives. Let us check them out one by one.

From Management Perspective

1. **Platform Independent** - Java applications can be developed on one machine and run on another without the need to recompile the code.
2. **High Performance** - There was a time when Java was critiqued for its sluggish performance as compared to applications developed in C/C++. That is not so, anymore. Due to advancements in JVM, Java programs can run as fast or even faster than programs written in other languages.
3. **Secure** - Java has security features built-in. It is possible to run Java applications in a sandbox with access to only those parts of the host machine that are deemed necessary for that application, while everything else remains inaccessible. This is made possible by the Java Security Manager.
4. **Familiar** - Java belongs to the C/C++ family of languages and heavily borrows syntax and features from C/C++. This allowed existing C/C++ developers to become productive on Java quickly and paved Java's way to success.
5. **Simpler** - Java eliminated several complicated features of C/C++ such as pointer arithmetic, operator overloading, multiple class inheritance, that made applications error prone and hard to debug.

6. **Multiple delivery modes and deployment options** - Java applications can be delivered remotely to client machines in multitude of ways. For example, a rich GUI based Java application developed in Java Swing can be deployed on client machines over the internet using Java Web Start technology or a web based application developed using JSP/Servlet/Facelets can be delivered to the clients through a web browser.
7. **Java ecosystem** - Java programming community has developed tons of commercial as well as open source libraries for a variety of higher level tasks such as mail, encryption, PDF, and what not. A Java programmer is advised to first look for an existing class or library that performs the required task instead of writing one from scratch. This promotes rapid application development.
8. **Backward Compatibility** - There are many reasons for Java's success but if I were to pick one reason that has kept Java going so, strong even after 25 years, then backward compatibility would be it. An application developed on any old version of Java runs fine, or better in most cases, on newer JVMs without requiring any recompilation. Every new version of Java brought in performance enhancements and new features without breaking existing code. Two Java versions are most notable in this respect. Addition of Generics in Java 5 and Lambda Expressions in Java 8 were huge enhancements to the language but even these additions did not break existing code. This is not a trivial thing and this did not happen by chance. Java designers put in great efforts to make sure that existing applications get all the benefits of newer JVMs without requiring any change. Many new programmers may not appreciate how important this is but I can tell you that it is. Backward compatibility assures upper management that their investment in the given technology is protected. This makes the decision to use Java for developing new applications a no brainer.

From Technology Perspective

1. **Compiled** - Java code is compiled into bytecode using a compiler. The bytecode is then interpreted by the JVM. This makes Java a compiled language. Java bytecode is, of course, a low level language and is interpreted. Unlike scripting languages such as JavaScript or PHP, which are not compiled and are interpreted directly by JavaScript or PHP interpreter, compiled languages deliver better runtime performance.
2. **Variety of technological solutions under one umbrella** - The Java platform includes a variety of technologies for varying needs. For example, for rich client side application development (also known as Desktop applications), it provides Java AWT, Swing, and JavaFX. For developing web applications, Java has Java Server Pages (JSP). For server side applications, it provides Servlet, Enterprise JavaBeans (EJB), Java Persistence Architecture (JPA), and Web Services. For communication between applications over the network, Java has support for raw sockets as well as higher level API such as RMI and XML-RPC. The technologies that I have mentioned here merely scratch the surface of the Java world. There are several more technologies, which you will learn about once you get into professional Java development.
3. **Multithreaded** - A multi-threaded program is able to manage multiple tasks at the same time. Java supports multi-threading and has features that make it easier to develop multi-threaded applications.

4. **Distributed** - Distributed applications are able to take advantage of multiple machines by distributing tasks to multiple machines. You might have one or more user facing GUI based clients running on one set of machines and a server that manages data centrally on another. Java has features that allows development of such applications.
5. **Garbage Collection** - JVM reclaims unused portions of program memory by removing unused objects. This process is called Garbage Collection. JVM provides various options for customizing garbage collection.

From Programming Perspective

1. **Object-Oriented** - Java is an Object-Oriented language. We will get into the details of this later.
2. **Structured** - Java belongs to the family of languages that follow the Structured Programming method. Structured programming involves modularizing large programs into smaller multiple pieces. This increases ease of maintenance and promotes reuse. Most commonly used programming languages such as C/C++, Java, C#, Python, and PHP, belong to this category. Other programming paradigms are Procedural and Functional. Practically, however, modern programming languages such as Java routinely borrow features of other programming paradigms. For example, Java has incorporated lambda expressions from functional programming.
3. **Statically typed** - Java is a statically typed language, which means, a variable and its type must be declared at compile time. Some languages such as JavaScript and Groovy, are dynamically typed. A dynamically typed language determines the type of a variable based on the value that it contains at runtime.
4. **Strongly typed** - Java is a strongly typed language, which means that after a programmer specifies what kind object or data a variable can point to, the variable cannot point to object of any other kind. For example, if you specify that a variable named car can point to objects of type Car, you cannot make that variable point to an object of type Tree in the middle of your program! This is very different from dynamically typed languages such as JavaScript. There are advantages and disadvantages of both the approaches but a discussion on that is beyond the scope of this book.
5. **Automatic memory management** - A Java programmer cannot manage program memory programmatically. In other words, a Java programmer does not directly write code to allocate or to release memory. A program simply creates objects as and when it needs. It does not even "delete" or "remove" objects explicitly. Allocation of memory for new objects and reclamation of unused objects is handled by the JVM completely automatically.
6. **Programmatic Exception Handling** - Java makes it easier to manage exceptional situations programmatically. As you will learn later, Java has try-catch constructs that helps you control program flow in case of unforeseen circumstances. It also provides constructs for closing resources automatically after use.

7. **Ready-made class library** - Java has a huge standard class library (Java API) that contains code for all sorts of common (such as File I/O and Database I/O) as well as uncommon (such as cryptography) tasks.

Don't worry if you don't completely understand every one of these immediately. You will gain better understanding as you go through the book.

2.3 Real world Applications of Java ↗

Java was originally designed with intelligent electronic consumer devices in mind. The idea was to create a uniform platform for controlling various kinds of consumer devices that have limited memory and processing power. Although Java has been successful in that domain, it has been found even more useful in developing high performance distributed enterprise applications. Many kinds of applications have been developed in Java over the years and it would not be possible to list all of them here. Broadly, however, Java applications can be classified as follows:

Desktop Applications

Java has been used to develop standalone desktop applications with or without a Graphical User Interface (GUI). A non-GUI application can be run on a machine as a service or launched from the command line as and when needed. For example, you can have a screen capturing application running silently in the background. Wide range of GUI based applications such as simple contact management to complicated ERP applications have been developed using Java's AWT and Swing package. Enthuware's popular mock exam simulator is a good example of a standalone GUI application of medium complexity.

In early years of Java, one more type of GUI based Java applications called "Applets" were popular. Applets were small applications that were loaded in a browser window when the user opened a website containing them. They soon ran out of favor due to sluggish performance and security restrictions. Applets have been deprecated in Java 8.

Distributed Applications

Many applications provide some sort of service to the users and to provide that service lot of activities are performed at a single location and the results are displayed to several clients. For example, an airline ticket reservation system manages the tickets and reservation data in the company, but the users view and book the tickets on their computer using a GUI. Thus, this is a distributed application where there is a client component that communicates with the server component. Various pieces of an application can communicate with each other using Java RMI.

Web Applications

These days most enterprise applications such as reservation systems, banking operations, shopping sites, are accessed by the users from the browser. Users access them just like regular websites without having to install anything on their machines. Java Server Pages (JSP) was a popular choice for building the user accessible parts of such application. In recent times, however, the user interface is built using JavaScript but the server side components are still built using Java Servlet,

EJB, JPA, Web Services, and many other Java technologies.

Middleware Applications

Establishing communication between two components of a distributed application is a non-trivial task in itself. There are different approaches to manage such communication. Applications such as message brokers and protocol handlers are meant to do exactly that. They sit in the middle of two applications and help them communicate. Several applications such as Tomcat Servlet engine, JBoss Application Server, Tibco messaging service are developed in Java.

Server side Applications

Java has been used to develop purely server side application such as Database engines or data analytics platforms. It has been used to develop web servers and file servers that implement the HTTP/HTTPS and FTP protocols.

Frameworks and Libraries

A huge number of open source frameworks and libraries have been developed by Java developers. Many of these frameworks such as Struts, Spring, and Hibernate form the back bone of several enterprise applications.

Mobile Applications

Before the advent of Android and iOS, Java was used to develop applications for mobile phones using Java Micro Edition platform. Although this is now completely outdated, Java as a programming language is still used to develop applications on Android.

Java has been used to develop applications in several domains. Some of the interesting ones are Scientific applications including research and engineering, Business applications such as Tax filing, Stock trading, Banking, and ERP, Media applications such as animations and games, Consumer applications such as browser, email clients, and media players, Data analysis applications such as Big Data, Data Mining.

The list of Java applications is huge and if you google the uses of Java, you will be truly astonished to see what all people have developed using Java. In this book, you will learn to make simple Java applications that can be run from the command line. But the rules of the language that you will learn in the process will enable you to make much more advanced Java applications in the future.

2.4 What Java is not!

Nothing except the almighty is perfect and one solution never fits all. There are certain things that you can't do in Java. Some because of imperfection and some because it was never designed for them. Here are a few things that you should know in this respect.

1. **Not an executable:** The exe files that you see on Windows machines are called native executables because the operating system (Windows) can understand and execute the instructions contained in them directly. Java programs can't be run like those executable files.

When you compile Java code, you don't get an "exe" file that you can just double click and run. The result of compiling Java code produces "class" files, which are interpreted by the JVM (and not by the operating system). The JVM is the executable program here. A class file is merely the input that you give to the JVM.

Over the years, however, several approaches to "execute" Java programs have been developed. Most involve some kind of a launcher application that launches the JVM with appropriate command line options (command line options are also called "switches", by the way). This launcher application could be an exe or a simple batch file.

Another approach is to "associate" class files with "java.exe". For example, when you double click on a text file, Windows launches Notepad application with the file that you double clicked on as a parameter. Or when you double click on an image file such as jpg or png, Windows launches the MS Paint or the Photos application. Similarly, compiled Java code files can be associated with the JVM program. I will show you both the approaches later.

2. **No code level security** - Unlike native executables, compiled Java code can be easily decompiled to produce Java source code that is almost the same as the original source code that was written to generate the compiled code. When I was a newbie Java programmer, I wrote a cool Java program. Then I thought what if someone just reverse engineered it. I tried to find ways to create non-human readable exe files of my program. I soon found out that it is not always possible to do so. There are tools that can convert class files to native exes but there are so many limitations to this process that it is not worth doing. There are tools that "obfuscate" Java code, which basically make the source code very hard to read. Thus, even if someone decompiles the class files, the result will still not be understandable.
The point to take away here is that it is easy to replace a class file with a hacked one.
3. **Clunky GUI** - The Graphical User Interfaces built in Java do not compare favorably to the GUIs built using native operative system libraries. Not only that they don't look and feel as good as the native GUIs, they don't work the same on all platforms. The GUI framework of Java negates the WORA feature of Java. It makes Java a "write once test everywhere" language. For this reason, Java is used a lot more for developing server side components than for rich client application.
4. **Not on Mobiles** - As I mentioned before, there is no JRE and no JVM for Android and iOS devices as of now. So, even though smartphones have become quite powerful you still can't run Java applications on them.
5. **High learning curve** - This point is not entirely correct but comes up very often. Java designers actually designed Java to be a lot simpler than C/C++. However, by some perspective, several newer languages such as Python made things even simpler. It is definitely easier to start writing programs in Python. I personally believe that as soon as you move towards writing non-trivial commercial applications, you will hit the ceiling with such languages. The extra effort that you put in initially towards learning Java pays off handsomely later.
6. **Language shortcomings** - As they say, hindsight is always 20/20. There are certain things in the Java programming language that could have done better from the get go. For example, Java has been criticized for not having a decimal datatype or unsigned datatypes. There are

many such things but most of them are quite subjective and a detailed discussion on them is beyond the scope of this book.

7. **Can't please everyone** - Java addresses a wide range of requirements, yet, it is entirely possible that it is no good for some use cases. For example, you can't write device drivers in Java because, again, Java programs run on the JVM and not directly on the machine.

Exam Objectives

1. Describe the Java Development Kit (JDK) and the Java Runtime Environment (JRE)
2. Describe the components of object-oriented programming
3. Describe the components of a basic Java program
4. Compile and execute a Java program

3.1 Installing Java

In this section I will show you how to get all the tools on your machine and how to set up the environment on your machine to use those tools. But first, I need to introduce some basic commands that we will be using often.

3.1.1 System Check

The two most important things that you should be handy with are the **File Explorer** and the **Command Prompt**. Let's check them out.

File Explorer

Windows 10 has various ways to open the File Explorer. The easiest way is to press the Windows key and the E key at the same time (**Win+E**). The File Explorer window will show up.

By default, JREs and JDKs are installed under `C:\Program Files\Java` folder. The following screenshot shows all the JREs and the JDKs that are present on my machine. You might see a JRE in this folder as well but we are more interested in the JDK. So, if you see a `jdk1.8.*` folder, great! You already have the JDK version that we need. If you don't or if you have an older version of the JDK (such as `jdk1.7`), that is fine too because I will show you how to install the most recent JDK shortly.

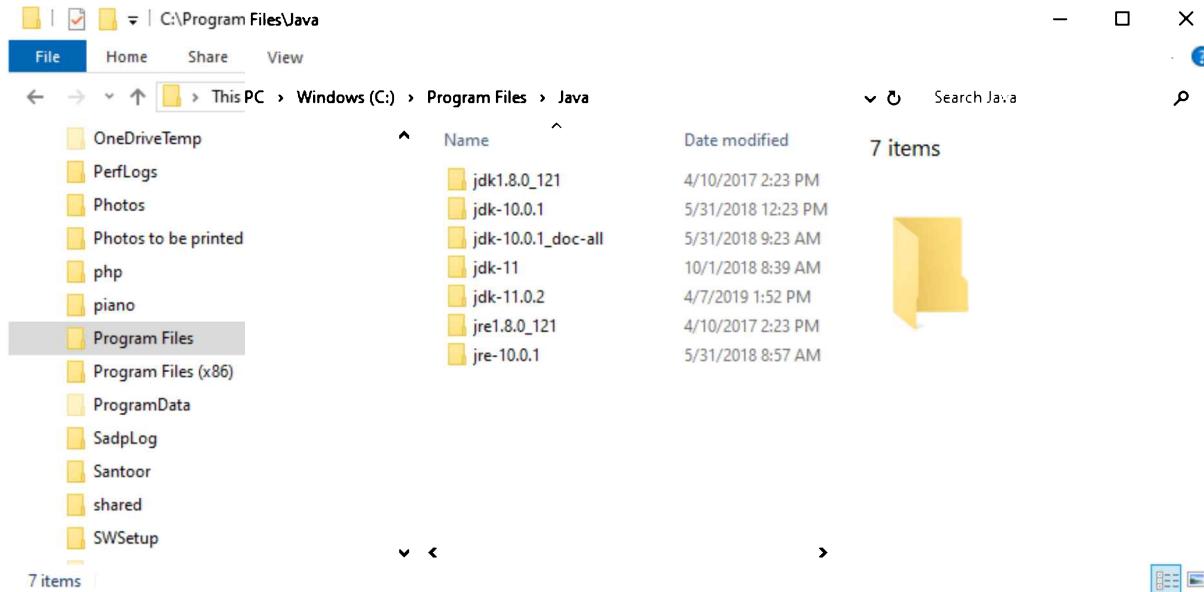


Figure 3.1.1: Looking for previous Java installations

Command Prompt

Press **Windows+R** keys to open the "Run" box. Type `cmd` in the run box and then click "Open" as shown in the following screenshot:

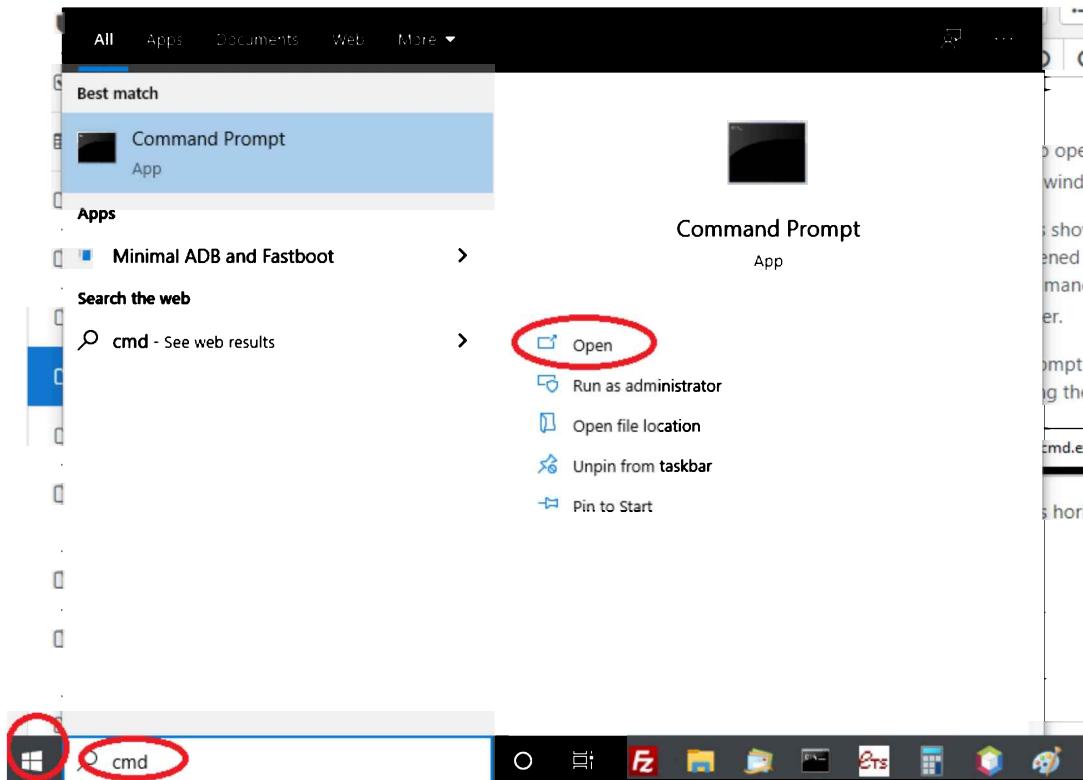


Figure 3.1.2: Opening Command Prompt/Window

You should see the following window after clicking on "Open".

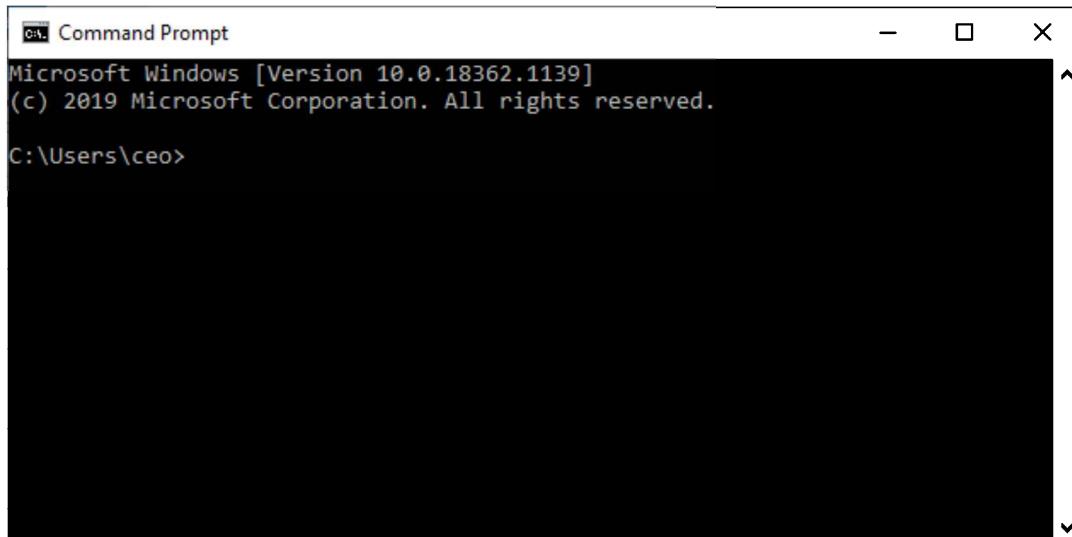
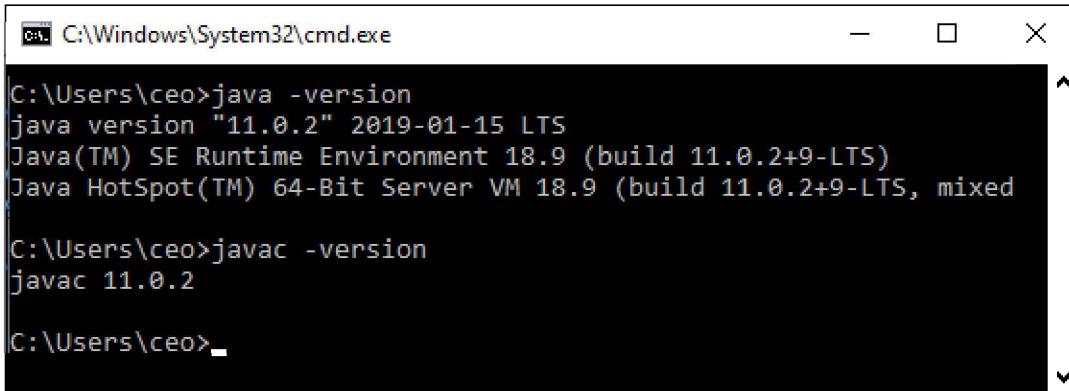


Figure 3.1.3: Command Prompt/Window

Observe the folder that is shown on the command window. In the above screenshot, it says C:\Users\ceo. This is my home folder. You should also browse to this folder from the File Explorer that you opened above. Create a new folder under this folder with name **jfcja**. You

can create it from the File Explorer or from the command prompt using the `mkdir` command (`C:\Users\mike> mkdir jfcja`). We will keep our test programs under this folder.

While your command prompt is open, let us also check whether you already have a JRE or a JDK configured on your machine. You can do this by running the `java` and `javac` commands on the command prompt as shown below.



A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The window contains the following text:

```
C:\Users\ceo>java -version
java version "11.0.2" 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed)

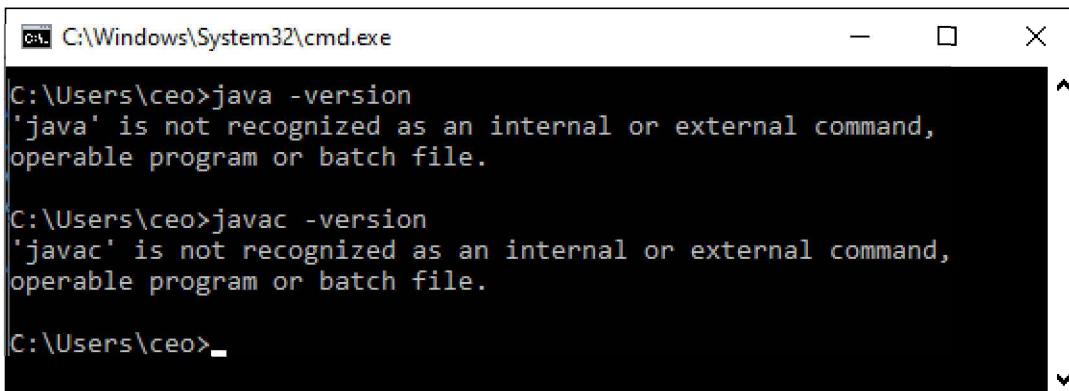
C:\Users\ceo>javac -version
javac 11.0.2

C:\Users\ceo>
```

Figure 3.1.4: Successful check for existing Java installation

The above image shows that I ran the `java` command with the `-version` option. It ran successfully and printed the version information. The output tells me that I have JRE version 11.0.2 installed and configured on my machine. Furthermore, the `javac` command also showed me the same version. This means, I have JDK version 11.0.2 configured on my machine. If you get a similar output that shows a Java version greater than or equal to 8, your machine has a JRE and a JDK installed and configured correctly. You are all set and you don't have to worry about the next section.

If your machine does not have any JRE and JDK configured, you will get the following output:



A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The window contains the following text:

```
C:\Users\ceo>java -version
'java' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\ceo>javac -version
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\ceo>
```

Figure 3.1.5: Unsuccessful check for existing Java installation

The outputs of both the commands say the same thing. Windows was not able to find `java` and `javac` programs on your computer. It means that either your machine does not have a JRE and a JDK installed, or if it is installed, Windows does not know where it is. If the `java` command shows version information but the `javac` command doesn't, that means you have a JRE but no JDK. Either way, you will need to install the right JDK.

For Mac and Linux

I have not included the steps to open the File Explorer and Command window for Linux and Mac users in this book because I am not sure if there are that many readers who require such detailed instructions. If you need instructions for your OS, please let me know and I will post the instructions online.

3.1.2 Understanding Java Builds

Before you go about installing Java, you need to be aware of a very important change in Oracle's licensing agreement. Until Java 10, Oracle's JDK was free to download as well as to use for development, demonstration, testing, and production purposes. However, since Java 11, Oracle does not allow the use of their JDK in commercial environment for free. It is still free to be used for other purposes but if you want to use it in production (meaning, the environment that is used to run your business), you need to purchase a license from Oracle. Does that mean Java is no more free? No, that's not exactly true.

Oracle has created a group called OpenJDK and has released the source code for the JDK on openjdk.java.net. Companies take that same source code and create their own version of the Java and the JDK. Each such version is called a "Java Build". Every Java build must also get a certification from OpenJDK that it satisfies all the criterial for being called a Java Standard Edition (Java SE) build. Oracle's official Java build is one of those builds and it is that particular build that is not free for commercial purpose. But many other builds, including the OpenJDK build, is free even for commercial use. So, basically, Oracle has two Java Builds - the official Oracle Java, which is not free for commercial use, and OpenJDK Java, which is free for commercial use.

Since the source code for all the builds is the same, there are all practically the same. The difference is in how soon security patches and updates are applied to a build. The difference is also in the support and help that is provided by the builder to the users of their build.

Among the free ones, the build provided by OpenJDK is considered to be the most updated and is also the build of choice for most companies.

The reason I needed to mention the above facts is that while the JFCJA exam is based on Java 8, the latest version of Java as of May 2024 is 22! The question is, if you do not already have JDK 8 installed is it still worth spending time on installing the old JDK when you can work with the latest?

Several changes and additions have been done since Java and normally, I recommend people to use the version of Java on which the exam is based. However, in this case, since this certification exam focuses on only the basics, most of the features added in recent Java versions are beyond the scope of the exam anyway. Therefore, I think it is alright if even you get the latest Java Build.

3.1.3 Downloading and Installing OpenJDK 15

Armed with the above knowledge, you can decide which Java would you like to have on your machine. I will go with the latest OpenJDK, which is available at jdk.java.net. On this site, you will see all the versions of the JDK that are available. I picked Java 15 and upon clicking on Java 15, the following is what I get.



Figure 3.1.6: Downloading OpenJDK 15

You can see four different downloads. One for each of the four commonly used operating systems. I will show you the installation process for Windows first and then for MacOS and Linux.

Step 1. Download Zip

Click on the "zip" link (on the last row, starting with Windows) to download the openjdk-15_windows-x64_bin.zip file and save it to your default downloads location. For example, if your user name is `mike`, your default download location on Windows 10 is `C:\users\mike\Downloads`. If you have changed your default downloads location, that is fine. Just note down the location where you are saving the zip file.

Step 2. Extract Zip

Open the File Explorer and go to the location where you have downloaded the zip file. Right click on the downloaded zip file and select the "Extract all..." option. You should see the following dialog:

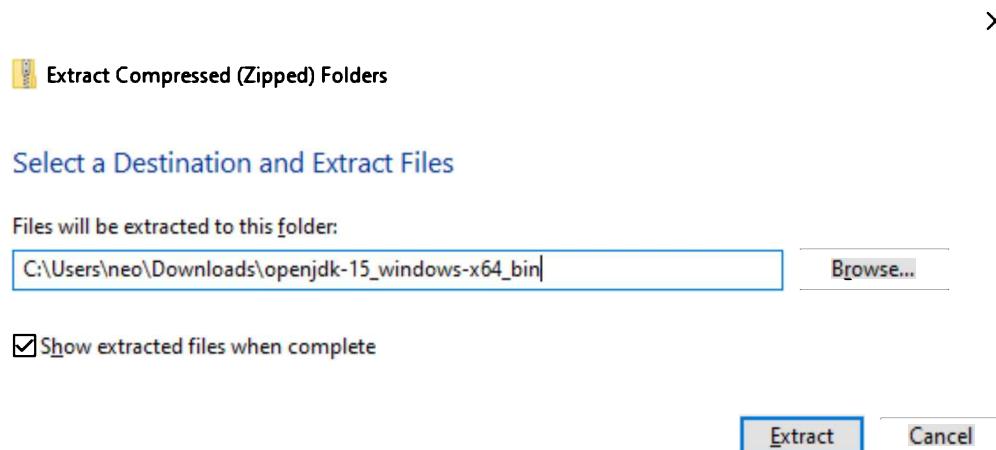


Figure 3.1.7: Extracting the contents of the jdk 15 zip file

You can extract the contents to any directory but I recommend extracting them to **C:** because that will make your life a bit easier. To do so, change the destination directory to **C:** before clicking Extract. You may be prompted for administrative permission to make changes to **C:**. If you do not have that and you cannot get help from the administrator of your machine, you may extract it to any other folder under **C:\Users\<your user name>** such as **C:\Users\mike**.

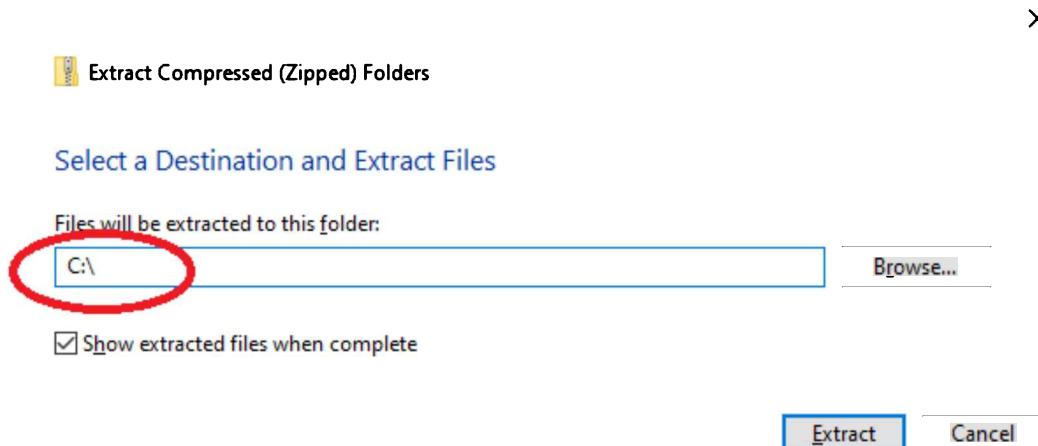


Figure 3.1.8: Extracting the contents to C:

Step 3. Verify the extract

After extracting the contents, you should have a new folder named **jdk-15** under **C:** (or under whichever directory you decided to extract the contents in). Under **jdk-15**, you will see several folders including **bin**.



Figure 3.1.9: Verify the contents of jdk installation folder

If you double click on the bin folder, you will see several executable programs. The ones we are interested in right now are `javac.exe` and `java.exe`. Javac.exe is the Java compiler program. We will use it to compile our Java code. Java.exe is the JVM program. We will use it to execute our compiled Java code.

Step 4. Setting up the PATH environment variable

Technically, your Java 15 installation is done. You can open up a command prompt and run the following command:

```
c:\jdk-15\bin\java
```

You will see the output shown in the following image:

```
C:\Windows\System32\cmd.exe
C:\Users\ceo>c:\jdk-15\bin\java -version
openjdk version "15" 2020-09-15
OpenJDK Runtime Environment (build 15+36-1562)
OpenJDK 64-Bit Server VM (build 15+36-1562, mixed mode, sharing)

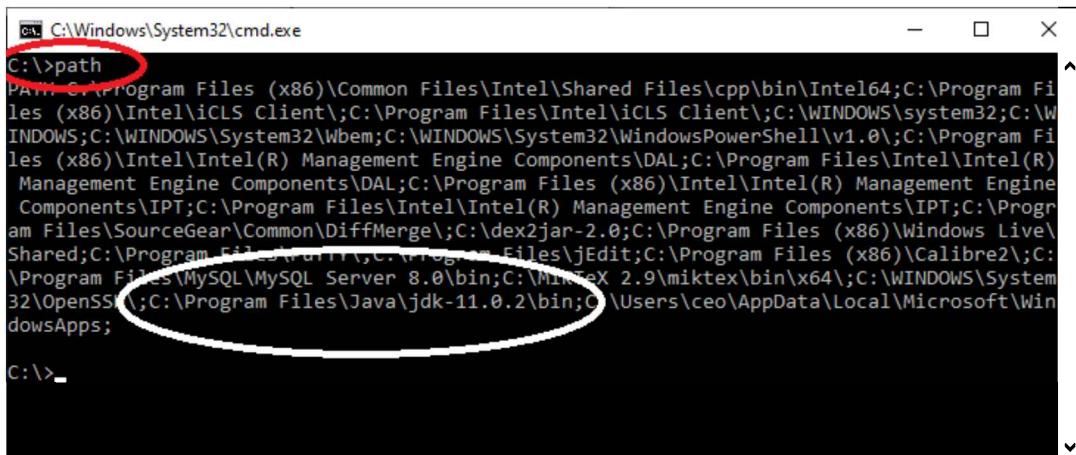
C:\Users\ceo>
```

Figure 3.1.10: Check Java version after installation

In the command that you just executed, you specified the full path (`C:\jdk-15\bin`) to the `java.exe` executable because the OS has no idea that it needs to look in the `C:\jdk-15\bin` directory to find `java.exe`. You will need to type `C:\jdk-15\bin` every time you want to compile Java code and execute it. This is too tedious. To make the OS look into the `C:\jdk-15\bin` folder when you try to execute `java`, you need to add `C:\jdk-15\bin` folder to the environment variable named **PATH**. The **PATH** environment variable tells the operating system the locations where your

executable programs may be located. Once you add a location to PATH, the OS will automatically search that location for the executable file of the program whenever you request the OS to execute a program.

Before messing with PATH, it is a good idea to check what it already has by executing the [path](#) command on the command window. The following is what I see on my machine. You should see something similar.



```
C:\>path
PATH=C:\Program Files (x86)\Common Files\Intel\Shared Files\cpp\bin\Intel64;C:\Program Files (x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCLS Client\;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\WBEM;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files\Intel\Intel(R) Management Engine Components\DAL;C:\Program Files (x86)\Intel\Intel(R) Management Engine Components\IPT;C:\Program Files\SourceGear\Common\DiffMerge\;C:\dex2jar-2.0;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files\UltraISO\;C:\Program Files\jEdit;C:\Program Files (x86)\Calibre2\;C:\Program Files\MySQL\MySQL Server 8.0\bin;C:\MikTeX 2.9\miktex\bin\x64\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files\Java\jdk-11.0.2\bin;C:\Users\ceo\AppData\Local\Microsoft\WindowsApps;
C:\>
```

Figure 3.1.11: Check existing value of PATH

The output lists all the folders present in the path. Whenever I try to execute any program on the command prompt, Windows will search for that program in all these locations one by one and execute it as soon as it finds the program. Windows will not look at the rest of the locations once it finds the program.

Observe that I have the [bin](#) folder of [jdk-11.0.2](#) in the path. So, if I try to execute just [java](#) (instead of [C:\jdk-15\bin\java](#)), Windows will pick up the [java.exe](#) executable file from [C:\program files\Java\jdk-11.0.2\bin](#). Since I am trying to work with JDK 15, I would like Windows to pick [java.exe](#) from [C:\jdk-15](#). For that to happen, I need to adjust my PATH variable. Specifically, I need to remove the existing Java 11's bin folder from PATH and add the new Java 15's bin folder to it.

Click on Windows Start/Search and type [env](#).

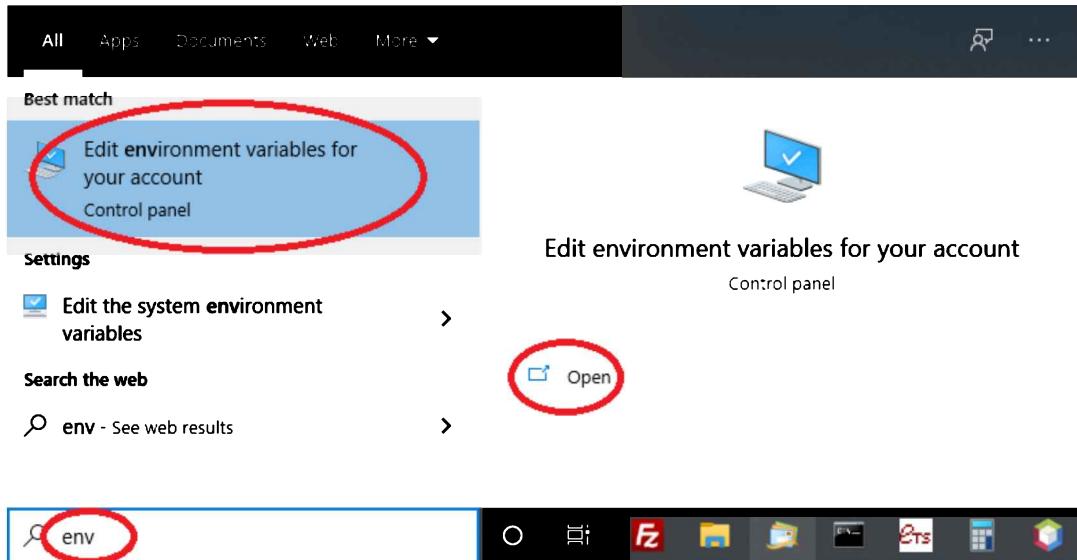


Figure 3.1.12: Edit environment variables

Observe that there are two different sets of environment variables - one for your own account and one for the system. The difference is that the variables defined for your account are visible only when you are logged in, while the variables defined in the system are visible to all the users. Thus, the actual value of an environment variable is the summation of both the values, with the system variables taking precedence over user account variables. If you did not see any JDK or JRE folder listed in the output of the path command, you can edit your own account's PATH variable. Otherwise, you need to check where exactly the path to the jdk or the jre is present and edit that PATH variable. Editing System's variable will require admin permissions but the process is the same. The following two screenshots show how to edit the existing PATH variable.

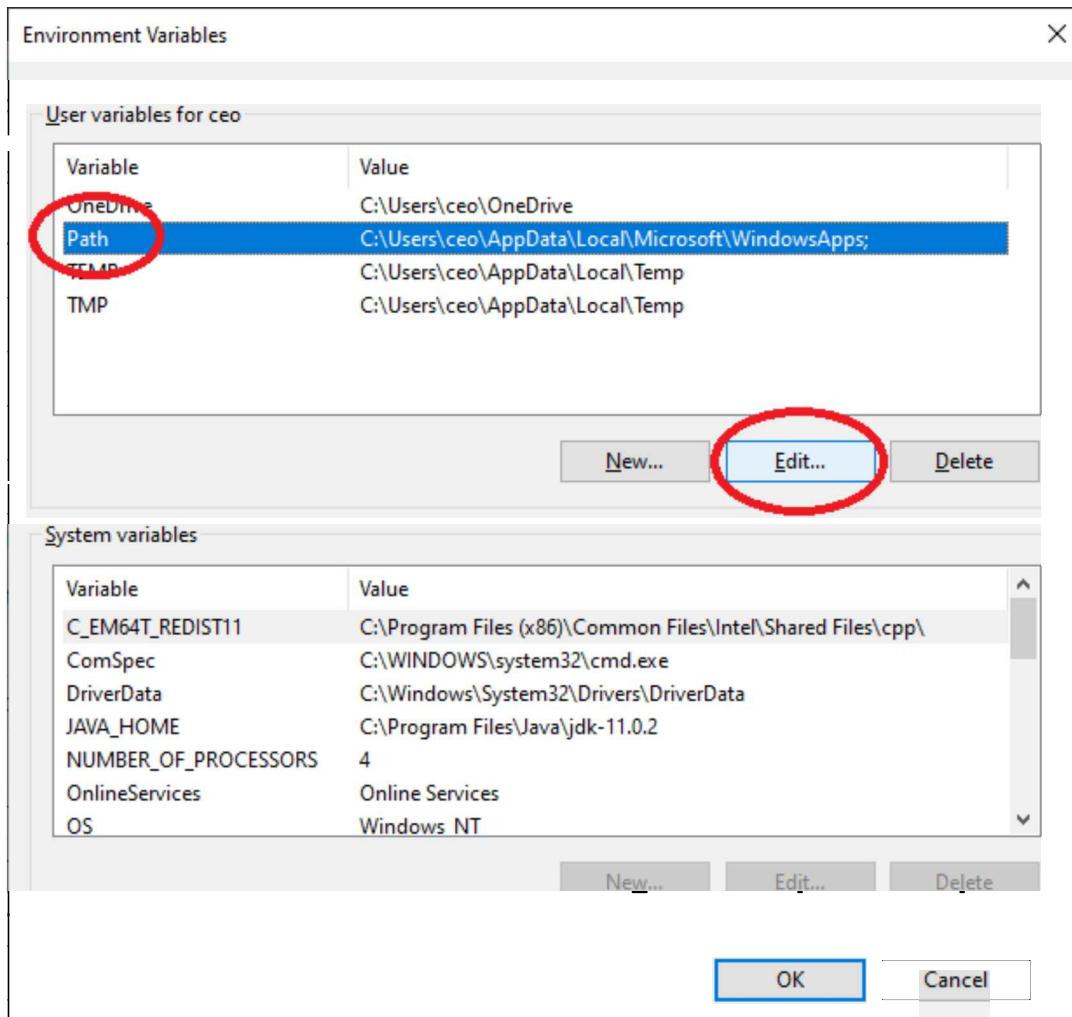


Figure 3.1.13: Edit PATH

After selecting the PATH variable and clicking on Edit, click on New to add a new folder to the path.

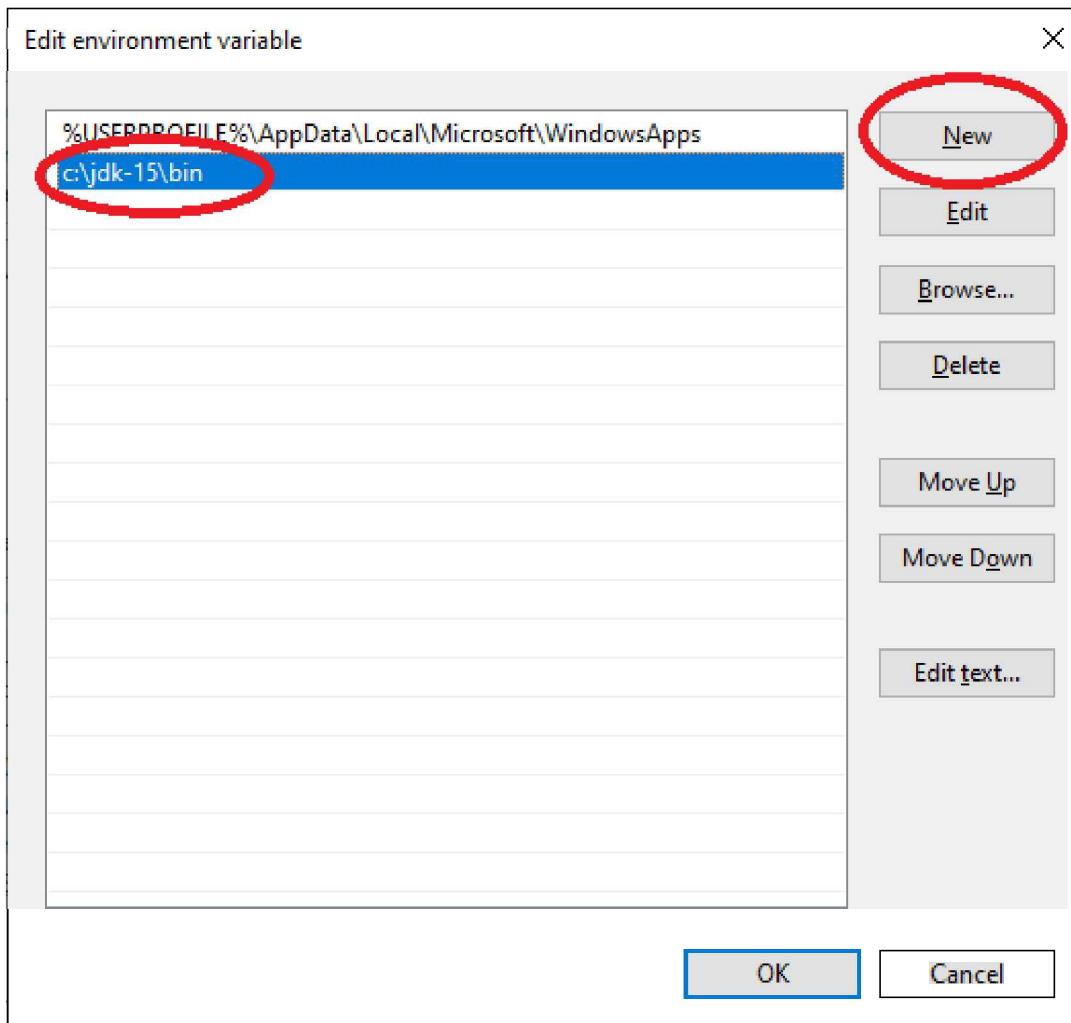


Figure 3.1.14: Add jdk's bin folder to PATH

Click OK one by one on all the open dialogs to save the changes. The change to the PATH variable will not be visible in the command windows that are already open. Close all of them, open up a new command prompt, and run the `java` and `javac` commands. You should see the following output:

```
C:\Windows\System32\cmd.exe
C:\Users\ceo>java -version
openjdk version "15" 2020-09-15
OpenJDK Runtime Environment (build 15+36-1562)
OpenJDK 64-Bit Server VM (build 15+36-1562, mixed mode, sharing)

C:\Users\ceo>javac -version
javac 15

C:\Users\ceo>
```

A screenshot of a Windows Command Prompt window titled 'C:\Windows\System32\cmd.exe'. The window shows the output of running 'java -version' and 'javac -version' commands. Both commands return the version '15'. The window has a standard Windows title bar and a black background for the command area.

Figure 3.1.15: Run java after updating PATH

The above screen shot shows that you are now able to run the correct version of Java as well as javac. This means, you are now ready to write and execute some Java code!

3.1.4 Using an Integrated Development Environment (IDE)

Code is, after all, just plain text. Decades ago, professional programmers used to write code using simple text editors. These were general purpose text editors that knew how to handle text but had no idea what that text meant or what that text was meant for. If one made a typing error in their code, they would find out only when they tried to compile the code in a separate command line window. The compiler would generate an error message and the developer would then fix the code and compile it again.

Soon, as applications grew, the size of the code became too big to be managed in this manner. This encouraged developers to use special purpose text editors that understood the programming language text that was being written. They highlighted keywords, syntax errors, and typing errors in the code. This increased the efficiency of the developers by eliminating the need to manually compile the code to find out errors.

State of the art applications nowadays contain not just code but also resource files such as images, properties, and configuration information. A project may contain thousands of files organized in hundreds of directories. A small change in one part of the code may have a cascading effect on hundreds of other code files. More over, there are multiple developers working on different parts of an application. It is practically impossible for a developer to manually manage all the code for an application using a simple text editor and the file explorer. This is where an Integrated Development Environment (IDE) comes into picture.

An IDE adds a lot more programming language specific features on top of text editors that help developers write good quality code. Besides highlighting syntax errors, they generate code based on commonly used idioms, analyze impact of a change on other parts, and compile the code transparently. They also provide support for debugging the code by executing it step by step. It would not be an exaggeration to say that professional software development is practically impossible without a good IDE.

The Java world has several free as well as commercial IDEs. Top three are - **Eclipse** (free), **Netbeans** (free), and **IDEA** (commercial). These are professional IDEs and have lots of features. But they are quite heavyweight. Meaning, they require a lot of efforts to install and have a relatively higher learning curve. But once you spend some time in learning their ways, they save a ton of time later. There are several lightweight IDEs as well, which are meant for beginners. They do not take much to install and have a low learning curve but they do not provide as many features either. **Dr Java** and **BlueJ** are two free IDEs in this category.

I know what you might be thinking now. No, I am not prepping you for downloading and installing an IDE. Gaining expertise on an IDE is extremely important for your productivity but you cannot achieve this expertise without learning the fundamentals. Under the hood, an IDE does the same things that programmers used to do manually. It just does those things automatically and transparently. Even though you may not need to do those things manually, you must still know what exactly is the IDE doing. Therefore, it is highly recommended that one should learn the basics of the language without using an IDE and move to a professional IDE later.

Another problem with using an IDE at this stage is that IDEs are not too quick to upgrade to the latest version of Java. For example, as of this writing none of the IDEs support Java 15. I am

sure, they will start supporting it soon but they don't at this moment. This is not a concern for professional application development because applications and teams do not switch to a new Java version that quickly either. In fact, even today, six years after its release, Java 8 is still the most used Java version!

So, we will start writing our Java code with a basic text editor. I prefer **Notepad++** on Windows. Unfortunately, it is not available on Mac and Linux. If you are using MacOS, you can go with **Atom** and if you are using Linux, you can use **Notepadqq**. But again, since will be writing simple and short programs, the editor doesn't matter. You can use whichever text editor you have.

3.2 JDK and JRE ↗

I explained the relation between JDK and JRE in the previous chapter. To recap, a Java Runtime Environment includes just the stuff necessary to run a Java program. This means, the JVM, the Java standard classes (aka the Java API), and its configuration and extension files. A Java Development Kit includes tools that are required to write Java programs such as the Java compiler and the debugger. A JDK contains a JRE and is therefore, much larger in size than a JRE. If you have JDK 8 installed on your computer, you will see a JRE folder inside the JDK folder.

The purpose of treating JRE and JDK as two separate entities was to ensure that a runtime environment could be preinstalled on all computers. This allowed Java application developers to ship Java applications without worrying about installing Java on the user's machine first.

Another reason for the split was related to licensing. Java application developers were allowed to bundle a JRE with their application without paying any fees to Oracle. This took care of the cases where the user's machine did not have a JRE preinstalled or had a different version of JRE.

The situation has changed drastically with Java 11. Starting with Java 11, Oracle has stopped bundling JRE with JDK. In fact, there is no standard JRE anymore. There is just the JDK. So, if you look inside the Java 11 (or higher) installation folder, you will not find a JRE folder.

However, the JDK itself has been modularized and depending on what features of the JDK your application uses, you can pick just those parts of the JDK and bundle them with your application. Licensing fees will depend on which "build" of Java you are using. For example, as explained before, Java is free while Oracle Java is not.

Note that bundling and distributing Java with an application is not a trivial matter and the above details merely present a broad overview of the issues involved. However, it is not something that you should be concerned with at this stage.

3.3 Components of a Java program ↗

Now that you have already how a Java program looks like, lets get into the components a Java program.

1. **The source code:** All the source code for a Java program resides in one or more "type" definitions, which are written in one or more source code files with `.java` extension. Don't be alarmed at the usage of the word "type" in the previous sentence. It is a technically more precise word than class and includes classes, interfaces, and enums. You will learn about interfaces and enums later but for now, we will just focus on classes.

2. **Java standard library classes:** As explained before, the Java platform comes with a huge amount of readymade components packaged in standard library classes. Any Java program can refer to those classes. You have already seen the usage of one such readymade class - `java.util.ArrayList` - in the previous section. Since these classes are already compiled, a Java program doesn't need their source code to make use of them.
3. **Third party libraries:** Just like the Java standard library classes, a Java program may make use of components developed by other companies or developers. Such components are packaged in one of "jar" files. In this case also, the source code for those components may not be available. All you need to use those classes is to put the jar files of the library into the "classpath" of your program while compilation and execution.

Besides the above, a Java program may also include resource files such as images, properties, and configurations, which we will not be worrying about at this stage.

Structure of a Java class

The exam sometimes uses the phrase "Java program" to mean a Java source code file. However, this association is imprecise and is just a remnant from the procedural programming paradigm. From the exam perspective, you should know that a Java program primarily contains the following three things:

1. Zero or one package statement
2. Zero or more import statements
3. Zero or more class definitions (type definitions, to be more precise)

The order of the above three things is important. We will dig deeper into this in the next chapter. A class definition is composed of a class declaration and the class body, like this:

```
package com.abc; <--- OPTIONAL
import com.xyz; <--- OPTIONAL
class MyClass <--- class declaration
{ //everything from the opening { to the closing } is the class body
}
```

I will go into the details of a class declaration and definition later, but for now, just remember that a class declaration can contain the `public` modifier, an `extends` clause that specifies the name of the class that it extends and an `implements` clause that specifies the names of the interfaces that it implements. For example:

```
public class MyClass extends ParentClass implements Interface1, Interface2
```

The `public` modifier, and the `extends` and `implements` clauses are all optional. Although the JFCJA exam does not expect you to know much about the `extends` and `implements` clauses, it expects you to be aware of them.

3.4 Compiling and running a basic Java program ↗

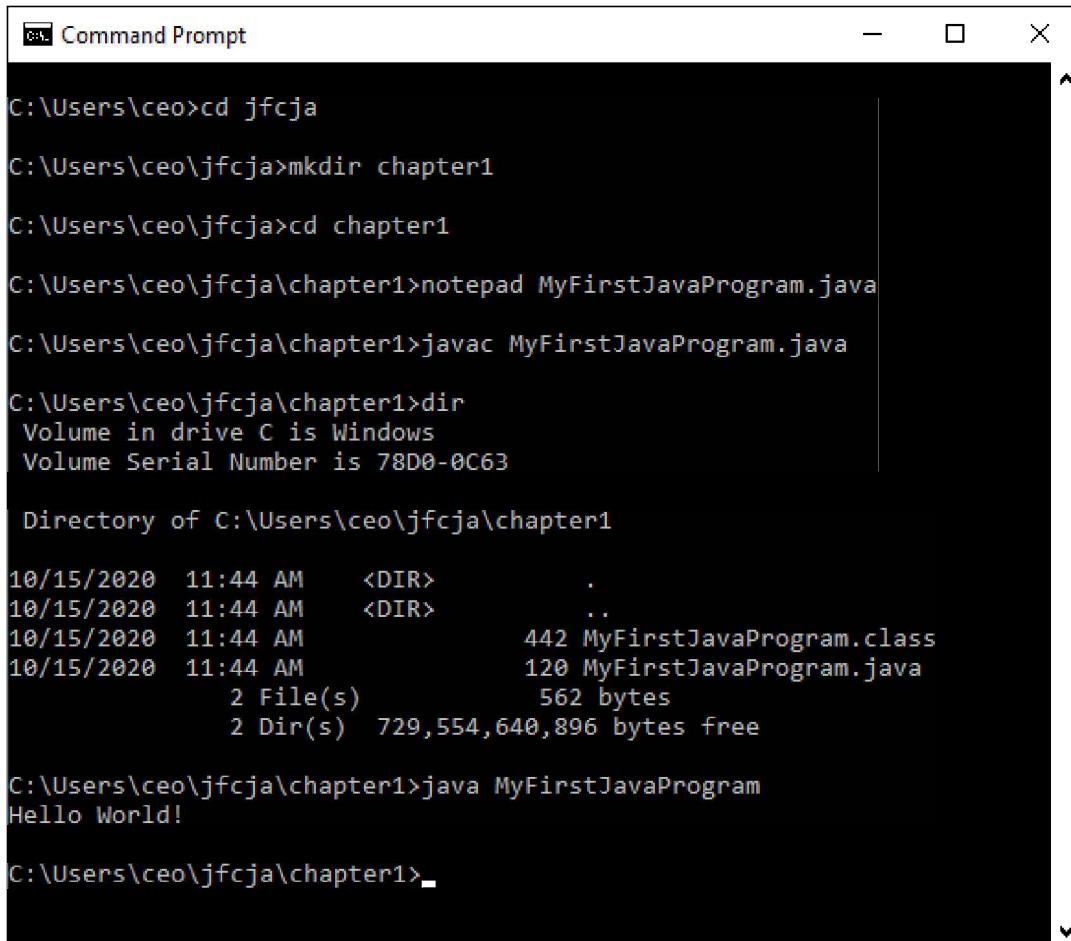
A specially written main method marks the entry point of a Java program. Thus, if the main method does not create any new object or does not call methods on other objects, the main method pretty much is the Java program. But Java being a highly object-oriented language, you can't just have a method without having a class. A method is the definition of a behavior and behavior can only be captured in a class. Thus, for creating even the most basic Java program that does nothing but print `Hello World!`, you must have a class! So, here it goes:

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

Here are the steps that you can follow to compile and run the above program.

1. **Source code location:** Open up a command prompt and cd to the folder where you are going to keep your programs while going through this book. As I suggested earlier, you may create a folder named `jfcja` under `C:\Users\<yourusername>\` to organize your all code. Under `jfcja`, create another folder called `chapter1` for keeping all the code discussed in this chapter. If you are following this structure, then you should `cd` to `C:\Users\ceo\jfcja\chapter1`
2. **Create Java file:** Create a text file named `MyFirstJavaProgram.java` in the `chapter1` folder and type up (or copy-paste, if you are reading the ebook version on your computer) the above code in this file and save it.
3. **Compile the source code:** Compile the Java file using the following command:
`javac MyFirstJavaProgram.java`
After compilation, you should see a new file named `MyFirstJavaProgram.class` created in the same folder.
4. **Run the program:** Run the program using the following command: `java MyFirstJavaProgram`
You should see `Hello world!` printed on the command line. Notice the absence of file extension `.class` while specifying the class name. To compile a Java source file, you must specify the extension of the file, i.e., `.java` though.

The following image shows the steps and the output that I got on my machine.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command history is as follows:

```
C:\Users\ceo>cd jfcja
C:\Users\ceo\jfcja>mkdir chapter1
C:\Users\ceo\jfcja>cd chapter1
C:\Users\ceo\jfcja\chapter1>notepad MyFirstJavaProgram.java
C:\Users\ceo\jfcja\chapter1>javac MyFirstJavaProgram.java
C:\Users\ceo\jfcja\chapter1>dir
Volume in drive C is Windows
Volume Serial Number is 78D0-0C63

Directory of C:\Users\ceo\jfcja\chapter1

10/15/2020  11:44 AM    <DIR>        .
10/15/2020  11:44 AM    <DIR>        ..
10/15/2020  11:44 AM           442 MyFirstJavaProgram.class
10/15/2020  11:44 AM           120 MyFirstJavaProgram.java
                2 File(s)      562 bytes
                2 Dir(s)  729,554,640,896 bytes free

C:\Users\ceo\jfcja\chapter1>java MyFirstJavaProgram
Hello World!

C:\Users\ceo\jfcja\chapter1>
```

Figure 3.4.1: Compiling and running MyFirstJavaProgram.java

If you get any other output while compiling or running this program, make sure you have installed Java as per the instructions given in the previous sections. It is important to get this right and sort out any installation issues at this stage before moving forward.

Single file Java programs

Java designers felt that the two step compilation and execution of Java programs is too tedious when you are just 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, you can also execute the `MyFirstJavaProgram.java` file directly from the command line using the following command:

`java TestClass.java` It is certainly easier to run a Java program in one step but there are certain restriction with this approach, which I will not get into right now. It is best to stick to the two step execution involving separate compilation and execution steps at this stage.

Observe that this program does not have any `package` statement or `import` statements. You will see the usage of these statements soon once we proceed towards writing more complicated programs.

3.5 Understanding the main method

3.5.1 The main method ↴

We discussed earlier that Java classes are not executables in the truest sense of the word. We cannot "execute" Java classes. The JVM is an executable. We execute the JVM. We pass the name (actually, the Fully Qualified Class Name, but more on that later) of the Java class that we want to run as an argument to the JVM. When the JVM runs, it locates the given class and looks for a specific method in that class. If it finds that method, it passes control to that method and this method then becomes the in-charge from there onward. If the JVM doesn't find that specific method, it errors out. In common parlance, we call it as executing or running a Java class or a Java program.

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

Examples of a valid main method:

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

Examples of an invalid main method:

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

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

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

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

Exam Tip

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

3.5.2 Command line arguments

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

```
java TestClass a b c
```

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

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

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

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

corresponding output generated by the program. The first line of the output shows the number of arguments received by the main method.

Command line used	Output	Inference
<code>java TestClass</code>		If no argument is specified, args contains a String array of length 0. Observe that a NullPointerException is not raised for <code>args.length</code> . That means args is not null . In this case args refers to a String array of length 0.
<code>java TestClass a</code>	<code>args[0] = "a"</code>	The first argument is stored at index 0. The first argument is NOT the name of the class.
<code>java TestClass a b c</code>	<code>args[0] = "a" args[1] = "b" args[2] = "c"</code>	Arguments can be separated by one or more than one white characters. All such separator characters are stripped away.
<code>java TestClass "a " "b"</code>	<code>args[0] = "a " args[1] = "b"</code>	If you put quotes around the value that you want to pass, everything inside the quotes is considered one parameter. Quotes are not considered as part of the argument. Observe that the first argument is "a" i.e. a String containing 'a' followed by a space.
<code>java TestClass "\""</code>	<code>args[0] = ""</code>	To pass a quote character as an argument, you have to escape it using a backslash.

Table 3.5.1: Arguments to main()

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

3.5.3 The end of main ↲

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

A simple Java program such as the one we used earlier to print arguments may have all its code in the main method. Once the main method ends, there is nothing else to do and so, the program ends. While you can write all your application code within the main method, Java applications are

usually composed of multiple classes. At run time, an application consists of instances of several classes that interact with each other by calling methods on each other. A Java application may also perform multiple activities in parallel, so, even if an activity implemented by one method ends, another activity implemented by some other method may still be going on. The code in main itself is just an activity. The end of the main method implies the end of only that activity. It doesn't mean the end of all the activities that may be going on in an application.

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

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

3.6 Object-Oriented Programming

3.6.1 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.

3.6.2 API

I talked about **Application Programming Interfaces** from a broader perspective in the first chapter. Let's dig into it a bit more because it connects 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. As discussed before, 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 **JFCJA** exam requires that you know about some of the widely used 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.

3.6.3 Principles of OOP

Object-Oriented Programming is a programming paradigm where a software application is visualized and designed as a bunch of objects interacting with each other. OOP is inspired by how real life objects behave and interact. If you look around, everything that you see is some kind of an object. Your TV, remote, chair, pen, computer, book, fan are all objects that exhibit certain properties and also provide ways for other objects to interact with them. In that sense, the whole world around you, including yourself, is just a bunch of objects interacting with each other.

Similarly, when developing a software program for solving particular problem in the OOP paradigm, you identify objects in that domain and the ways in which they interact with each other. In Java, these objects are modeled using classes and their behavior is modeled as methods of those classes. The interaction between the objects causes changes to the state of the objects, which is represented using the data fields of the objects.

Let me give you a quick example. Let's say you are developing an application for managing the inventory of a car dealership. This application is supposed to maintain a list of cars. Here, you can observe two type of objects at the outset - Dealership and Car. Furthermore, you know that there is only one dealership but the dealership has many cars. Thus, there should be just one Dealership object and multiple Car objects in the system.

Dealership should have a way to add, remove, and display the cars that are currently present in the inventory. These operations can be modeled as methods of Dealership. The data that the Dealership object holds at any moment, that is, the list of Car objects, is the state of the Dealership object. An interaction with the Dealership object may modify its state. For example, adding and removing a Car from the Dealership will modify its list of cars. It is important to understand here that software objects are designed to exhibit only such behavior as is relevant for the application. A real dealership does a thousand different things but they are not relevant for the inventory management application. Since we are modeling a dealership from inventory management perspective, we abstract out only those operations that are important for managing the inventory. That is why, our Dealership object may support only a few operations. Similarly, our Car objects may exhibit only a few properties such as the make, model, and year. This process of identifying relevant behavior for designing software objects is called **abstraction** in OOP terminology.

The above discussion illustrates the fundamental thought process involved in designing a solution using the OOP paradigm. Some people find OOP less intuitive at the beginning because there are no physical objects in software. Modeling software parts as objects containing code as well as data requires abstract thinking. However, it does payoff in the long run. In contrast, there is the "Procedural Programming" paradigm, which perceives software as a series of processes executed over a given set of data. An application is broken up into various processes which call one another as and when required. For example, in the car dealership example, you could have "show cars", "add car", "remove car" as procedures that manipulate a list of cars stored somewhere. Beginners typically find procedural programming a lot easier to comprehend but experience has shown that OOP is more suitable for developing modern applications.

Over the years, the process of modeling software objects has evolved and several strategies and practices have been established to make this process more efficient. Discussing all the details of OOP will require a book of its own. So, here, I will only cover the four important principles of OOP that you should keep in mind. Do not worry if you do not understand or appreciate them fully at this stage. You will get the hang of OOP slowly as you write more and more programs. In fact, one of the reasons why Java has been so, successful is that it makes OOP easier to implement. You will gradually realize that Java has features that prevents programmers from veering off towards procedural programming and that gently steer them towards OOP.

Abstraction

Abstraction is the process of capturing relevant details of an entity or a concept. In Java, such an entity is generally described using the "class" artifact. For example, in the car dealership example above, the concepts of dealership and car can be described using a Dealership class and a Car class. Once the concept is defined, multiple instances of the concept can be created.

Encapsulation

Encapsulation means to hide the implementation details of an entity. An entity should only expose the "what" and not the "how". For example, anyone interacting with a car dealership only needs to be able to browse all the cars that the dealership has, not how the dealership maintains its cars. As long as the dealership is able to provide a way to add/remove/display cars to the users, it

shouldn't matter if the dealership changes the way it manages its inventory. Hiding implementation details ensures that various components of an application do not unwittingly depend on each other's implementation details. This allows the components to evolve without breaking the system. In Java, encapsulation is typically achieved by using appropriate access modifiers for the members of a class. Typically, data members of a class are declared private while methods are declared public.

Inheritance

Inheritance is process of creating specialized entities by extending existing entities. It is also seen as establishing an "is a" or a "parent child" relationship between two entities. Inheritance provides a way to add new entities or new features to the system quickly by reusing existing ones. For example, the inventory management system for the car dealership may want to display Sports cars differently while still treating them as regular cars in other respects. One could, therefore, create a specialized entity named SportsCar that extends Car. By extending Car, a SportsCar would automatically inherit all the features and functionality of a Car.

Polymorphism

Polymorphism means same entity exhibiting different behavior depending on the context. Polymorphism is actually a direct consequence of inheritance. For example, when you want to count the number of cars in a dealership, it doesn't matter whether a car is a sports car or just a regular car. But when you feel the difference when you drive a sports car. So, while counting, all the cars behave in the same way, but while driving, a sports car behaves differently from other cars.

3.6.4 OOP in Java

Java is an Object-Oriented Programming language. What it means is that the Java programming language provides constructs that help you implement your application in the OOP way. It doesn't mean that Java allows applications to be developed only in the OOP way. Some programmers, who are more conversant with procedural programming, think in terms of functions and procedures and end up writing their programs in that manner. That is not the right approach though. In fact, you will realize later that Java makes it harder for you to write your programs in the procedural way. If you are going to be writing programs in Java, you should start thinking in terms of objects and their interactions instead of functions calling each other. A Java program or a Java application is, therefore, nothing but a bunch of objects interacting with each other.

As explained before, to create objects, you first need to describe the relevant details of those objects using the process called abstraction. The most basic building block to do that in Java is called a "class".

Describing Objects

A class allows you to describe the data elements and the behavior of the objects that you want to deal with in your application. In OOP parlance, data elements are called "attributes" or "properties" and a behavior is called a "method".

For example, the inventory management application that we talked about earlier has two kinds of objects - Dealership and Car. The details of these two can be captured using two classes as follows:

```
class Car{
    String make; String model; int year;
}
class Dealership{
    java.util.ArrayList cars = new java.util.ArrayList();
    void addCar(Car c){ cars.add(c); }
    void removeCar(Car c){ cars.remove(c); }
    void printCars(){
        for(Car c : cars){
            System.out.println("Make: "+c.make+" Model: "+c.model+" Year: "+c.year);
        }
    }
}
```

The `Car` class captures the details of a car. At this time, we care only about the make, model, and year of a car. The `Dealership` class captures the details of the dealership. It has an attribute named `cars`, which is supposed to contain the list of available cars and a few methods to manage and display that list. Observe that these `Car` and `Dealership` are classes, not objects. They merely describe what a car and a dealership objects contain and how they behave. To make a program, we will have to create objects using these classes and make them interact.

Describing Behavior

Let me ask you a question, do you really care about the circuitry that is involved when you switch things on and off? No, I guess. As long as you see a switch that switches a device on or off, you are good to go, right? You simply push the switch on or off, and it is up to the device connected to that switch to perform the switching.

What we have just identified is a behavioral aspect of things. If we say that something switches on or off, we are essentially describing a behavior of that thing. For example, a Fan, a LightBulb, and a Car can all be switched on or off. Here, "switchable" is the behavior that is exhibited by the Fan, LightBulb, and Car objects.

It is the same with Java objects as well. Once we identify a particular behavior of a range of objects, we can deal with all of them in the same manner without worrying about what those objects are or how they implement that behavior. Java allows us to capture just the behavior using an "interface". For example,

```
interface Switchable{
    void on();
    void off();
}
```

The above is merely a description of the behavior. To us, anything that can be switched on or off is a **Switchable**. The important thing to understand here is that we are not actually trying to switch anything on or off because we don't even know how different things do this activity. Indeed, a Fan and a Car switch on and off differently. In other words, there are no implementation details in an interface.

It is the job of the class of objects to decide how to implement a particular behavior as is appropriate for them. In Java, we tie the behavior to a class using the keyword "**implements**". For example,

```
class Fan implements Switchable{
    boolean on;
    void on(){
        //perform all the things that are need to switch it on
        on = true;
    };
    void off(){
        //perform all the things that are need to switch it off
        on = false;
    };
}
```

The benefit of abstracting out just the behavior using interfaces is that they making the interaction between different objects a lot easier to maintain. Think of it this way - a care taker of a hotel can be directed to switch on/off anything as long as it has the on/off switch. What if different devices had different ways to turn them on or off? People would go crazy learning new ways to deal with their devices. Similarly, in programming, it is important to identify and abstract out behavior of the objects in your application as much as possible.

The JFCJA exam does not expect you to know about interfaces but I have discussed it in brief because some of the topics covered in JFCJA do require you to be aware of the concept.

Creating Objects

Once you have described the kind of objects you want to deal with, you can create objects at will using "**new**". For example, `Car car1 = new Car();` or `Dealership dealer = new Dealership();` You might have noticed `new java.util.ArrayList()` in the `Dealership` class. Yes, `java.util.ArrayList` is a class and `new java.util.ArrayList()` creates an `ArrayList` object. We are using it to store the car objects. It represents the current inventory of the dealership. Don't worry about where this class came from or about the `java.util` part right now.

Making Objects interact

Objects interact with each other by calling methods on or accessing data fields of each other. For example, the `showCars` method of the `Dealership` class accesses the `make`, `model`, and `year`, attributes of a `Car` object by using a "reference" to a `Car` object.

Launching a Java program

I said above that a Java program is nothing but a bunch of objects interacting with each other. But who creates those objects? Who initiates the interaction among those objects? It is like all the players of the game are on the field but who starts the play? In Java, we do that by creating special method called "main". We launch the Java program by asking the JVM to execute this main method. For example:

```
class MyApplication{
    public static void main(String[] args){
        Dealership dealer = new Dealership();
        Car c = new Car();
        dealer.addCar(c);
        dealer.showCars();
    }
}
```

Don't worry about all the details such as `public`, `static`, `void`, and `String[]` that you see in code above. We will get to all that later. Just observe that we are creating a few objects (a Dealership object and a Car object) and calling a few methods (`addCar` and `showCars`). The ball is set rolling when the JVM executes the `main` method. For this reason, the `main` method is also called the "entry point" of a Java application.

When objects start interacting in an application, they may create even more objects as governed by the code written in those classes. This is basically what any Java program does.

It is important to understand where exactly are the objects are created and what do we mean by "reference". I will explain it with a detailed example now.

3.6.5 Relation between Class, 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. When you use the "new" keyword, you create a new object of that 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:

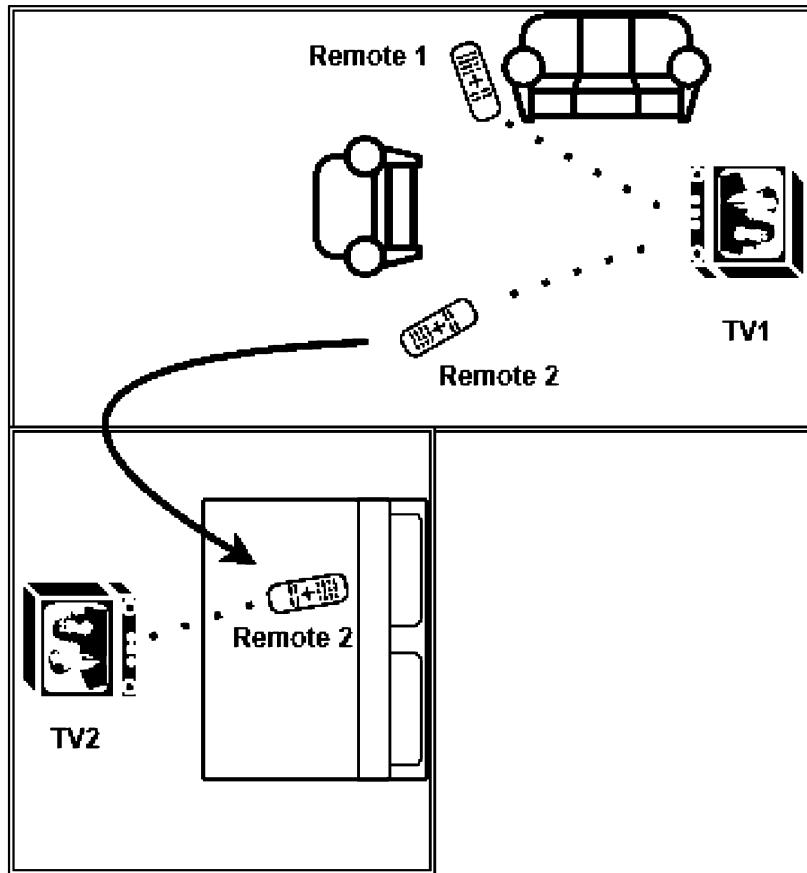


Figure 3.6.1: Relation between Class, Object, and Reference

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.

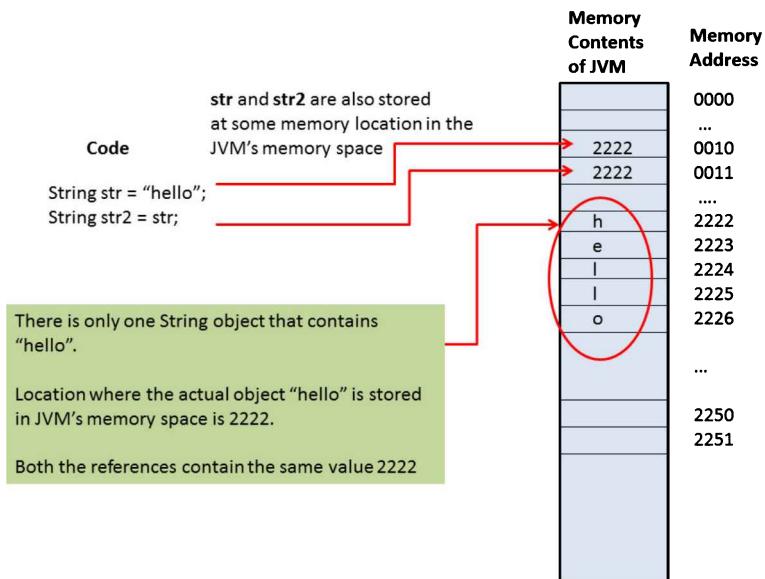


Figure 3.6.2: Object and Reference

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.
  
```

3.6.6 static and instance ↗

We saw previously how we use a class to abstract an entity and how we instantiate objects of that class to create a program or an application. I also mentioned earlier that it is possible to write programs in Java using the procedural programming paradigm, in which we think of tasks or routines that can be invoked to achieve our goal. For example, if we want to compute simple interest for a given principle, rate, and time, we would want to simply invoke a procedure named

`computeSimpleInterest` with three arguments. It would compute the interest and return the value. Basically, we are not interested in abstracting any entity here. We are just interested in writing a procedure or a function to compute the simple interest for a given set of inputs. There is no need for objects or the interplay of objects in this situation. Since, we don't need any object here, ideally, we shouldn't need any class either because the whole purpose of a class is to create objects!

However, unlike in other languages such as C/C++/Python, you can't just have independent procedures or functions in Java. Every piece of code or every set of instructions must belong to one or another class. This may seem like a big restriction to you at first but it is not. Java has a slightly roundabout way of achieving the same by using the "**static**" keyword.

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. If it helps, you can think of it as something that remains the same for all instances of a class. The idea here is that if we use something that belongs to the class instead of an object of the class, we won't need to create an object of that class. Let me show you how this works by creating a function to compute simple interest as a static method - I am calling it "method" instead of "function" because method is the right terminology in the Java world - of a class:

```
class FinancialFunctions{
    static double computeSimpleInterest(double p, double r, double t){
        return p*r*t/100;
    }
}
class TestClass{
    public static void main(String[] args){
        double interest = FinancialFunctions.computeSimpleInterest(1000, 10, 1);
        System.out.println(interest);
    }
}
```

You can copy paste the above program in a file named `TestClass.java` and run it. Observe that our main method uses the `computeSimpleInterest` method without creating any object of the `FinancialFunctions` class. There is no usage of the `new` keyword anywhere in the above program.

Just like static methods, you can have static variables in a class. As we discussed earlier, 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. That is why, static variables are also called "class variables".

The following partial code explains how it works:

```
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.
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.

The purpose of using `static` for defining the main method should also be clear now. When we run our program, the JVM doesn't have to create an object of our class. It simply invokes the main method of the class whose name we pass as an argument on the command line.

3.7 Exercise ↗

1. Run `java -help` and `javac -help` commands on the command line. Observe the output.
2. How important is the name of the file of a Java source code? What are the advantages and disadvantage of writing a class in a file with the same name?
3. What should be the name of a file that contains a class with the main method?
4. In their zeal to make their class as useful and functional as possible, a developer has created the following class:

```
class DoEverything{
    int INTERESTRATE = 10;
    double computeInterest(double p, double t){
        ...
    }
    String defaultFilePath
    double saveDataToFile(String data){
        ...
    }
}
```

Which OOP principles does this class violate and why?

5. Which part of a class does encapsulation tries to control - data or functionality? Why?
6. Information hiding is a key principle of OOP. What exactly does it try to hide when applied to a Java class?
7. Think of different ways of code reuse. (Hint: Inheritance and Aggregation) Which one falls under OOP?

Exam Objectives

1. Identify the conventions to be followed in a Java program
2. Use Java reserved words
3. Use single-line and multi-line comments in Java programs
4. Import other Java packages to make them accessible in your code
5. Describe the `java.lang` package

4.1 Conventions

4.1.1 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 an 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.

4.1.2 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, although not encouraged, may include underscores. For example, `current_account` is a valid name but `currentAccount` is better.
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. You may sometimes see the type information incorporated in a name, for example, `strAccountId` or `double_balance`. These are valid but it is not a recommended practice in Java because Java is strongly typed. If `accountId` is defined as a `String`, the Java compiler will never allow you to assign an `int` value to it. Languages such as JavaScript are loosely typed and so, developers try to make sure they don't inadvertently assign a value of a different type to a variable by adding the type information in the variable name.

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.

4.2 Java Identifiers and reserved words

4.2.1 Keywords, Literals, Reserved words, and Identifiers

Certain words have a special meaning in the Java language. Such words are called **keywords**. You cannot use these words in your program to mean something else because as soon as the compiler sees those words, it takes certain predefined actions. For example, as soon as the compiler sees the word `class` written in a program, it realizes that the programmer is trying to define a class.

In addition to the keywords, Java restricts the use certain words in the program. For example, `goto`. Even though `goto` doesn't mean anything to the compiler, Java language designers have prohibited its use in a Java program for reasons that are not relevant right now.

Furthermore, there are certain words that the Java compiler interprets literally. Meaning, the compiler simply uses them as the value that they are meant to represent. Such words are called **literals**. For example, `true`, `false`, and `null` are literals. The first two represent the two boolean values and the third represents that a null value. Although not words per se, numeric values such as 0, 1, 100, and 99.9 are also literals because they are interpreted by the compiler literally for the value that they represent.

The set of all such words, i.e., keywords, prohibited words, and literals is called **reserved words**.

The programmer is free to assign any meaning to other words in a Java program. For example, while abstracting the concept of a car, you may use the word `Car` to define the class `Car`. Thus, in your program, the word `Car` has a meaning that is specific to your program only. Another person may very well use the word `Car` to name a variable! Such words whose meaning is specific to a program are called "identifiers". You use identifiers to name classes, methods, and variables. Java has specific rules to that you need to follow while naming an identifier.

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 reserved word or a literal (i.e. `true`, `false`, or `null`). The following is a list of reserved words in Java:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while
<u>_</u> (underscore)				

Table 4.2.1: Java Reserved Words

Based on the above rules, 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
String _; //valid in Java 8 but not in Java 9
```

Observe that `var` is not a keyword. It is classified as an identifier with special meaning. It is actually replaced by the compiler with the type deduced from the value that is being assigned to the variable. So, `var var = "hello";` would be a valid statement because the compiler will effectively replace the first `var` with `String`.

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). So, while underscore itself is not a valid identifier, a valid identifier may start with an underscore.

4.2.2 Summary of important Java Keywords

Let us now take a quick look at a few keywords that we will be working with most of the times and the ones that are important for the exam. We will get into the details of all of these later.

1. **class:** As you have already seen before, the `class` keyword is used to define a class of objects. Technically, a class defines a new "reference type". The other two keywords which define reference types are `interface` and `enum` but they are not on the exam. In this book, we will work only with `class`.
2. **new:** The `new` keyword is used to create a new instance of a class. For example, `new Car()`, `new Person()`, `new ArrayList()`, `new Integer()` and so on.
3. **if and else:** The `if` and `else` keywords are used to create decision statements. For example, `if(number%2 == 0) System.out.println("even"); else System.out.println("odd");`

Although commonly referred to as an "if-then-else" statement, the actual statement does not involve the word "then" at all. In fact, "then" is not even a valid keyword.

4. **switch, case, default, and break:** The `switch` and `case` keywords is used to create more complicated decision structures involving multiple paths instead of just two (as supported by an if/else structure). For example:

```
switch(number){  
    case 0: System.out.println("zero"); break;  
    case 1: System.out.println("one"); break;  
    default : System.out.println("neither zero nor one");  
}
```

The `default` and `break` statements are used to tweak the working of the switch statement.

5. **for, while, do, break, and continue:** The `for`, `while`, and `do` keywords are used to create different flavors of looping statements. For example:

```
do{  
    System.out.println("hello");  
}while(condition);
```

The `break` and `continue` statements are used to tweak the working of the loops.

6. **package and import:** The `package` keyword is used to organize classes into packages, while the `import` statement is used to refer to the classes defined in other packages.

7. **public, private, and protected:** Recall the principle of Encapsulation from the OOP basics section, which means to hide implementation or other unnecessary details of a class. The `public`, `private`, and `protected` keywords are used to achieve encapsulation.

8. **extends and implements:** The `extends` and `implements` keywords are used to create "inheritance" relationships between classes. These relationships are also known as "is-a" relationship. I will not discuss either of these keywords because they are not covered in the JFCJA exam.

9. **boolean, byte, char, short, int, long, float, and double:** These keywords define the "primitive" data types available in Java. They are used while declaring primitive variables in your Java code. For example, `int count;` `double amount;` and so on.

10. **super and this:** These keywords are used to the instance members of the parent class or the current class.

11. **return:** The `return` keyword is used to return a value from a method.

12. **static:** The `static` keyword is used to specify that a member defined in a class belongs to the class itself and not to an instance of that class.

13. **throws, catch, and throw:** The `throws`, `throw`, and `catch` keywords is used to perform exception handling in the code.

I have not mentioned other Java keywords here because they are not on this exam.

4.3 Create and import packages

4.3.1 The package statement

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

Important points about the unnamed package

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

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

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

4.3.2 Quiz

Q1. Which of the following code snippets are valid?

Select 1 correct option.

A.

```
//in Test.java
package;
public class Test{
}
```

B.

```
//in Test.java
package mypackage;
public class Test{
}
```

C.

```
//in Test.java
package x;
public class Test{
}
package y;
class AnotherTest{
}
```

D.

```
//in Test.java
package x;
package y;
public class Test{
}
```

Correct answer is B.

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

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

4.3.3 The import statement ↗

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

If you want to refer to this class from another class in a different package, say `com.xyz.abc`, you need to use the FQCN, i.e., `com.enthware.ocp.Test`. For example, `com.enthware.ocp.Test t = new com.enthware.ocp.Test();`

If you try to use just the simple class name, i.e., `Test t = new Test();`, the compiler will assume that you mean to use the **Test** class from the same package, i.e., `com.xyz.abc` and if it

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

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

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

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

The word "**import**" is really a misnomer here. The import statement doesn't import anything into your class. It is merely a hint to the compiler to look for classes in the imported package. You are basically telling the compiler that the simple class names referred in this code are actually referring to classes in the packages mentioned in the import statements. If the compiler is unable to resolve a simple class name (because that class is not in the same package as this class), it will check the import statements and see if the packages mentioned there contain that class. If yes, then that class will be used, if not, a compilation error will be generated.

Important points about the import statement -

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

What you cannot do:

1. There is no way to import a "subpackage" using the import statement. For example, `import com.enthuware.*;` will import all the class in package `com.enthuware` but it will not import any class under `com.enthuware.ocp` package. Furthermore, `import com.enthuware.*.*;` is illegal. This essentially means that technically, there is no concept of "subpackage" in Java. Each package must be imported separately.
2. You cannot import a package or a class that doesn't exist. For example, if you try to use some random package name such as `import xyz.*;` the compiler will raise an error saying,

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

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

3. Unpackaged classes (the phrases "unpackaged classes" and "classes in the default or unnamed package" mean the same thing, i.e., classes that do not have any package statement) cannot be imported in any other package. You cannot do something like

```
import *;
```

4. If a class by the same name exists in multiple packages and if you import both the packages in your code, you cannot use the simple class name in your code because using the simple name will be ambiguous. The compiler cannot figure out which class you really mean. Therefore, you have to use FQCN in such a case. You may import one package or class using the import statement and use simple name for a class in that package and use FQCN for classes in the other package.

The requirement to use two classes with same name but from different packages typically used to arise a lot while using JDBC. JDBC related classes are in `java.sql` package and classes in this package use `java.sql.Date` class instead of `java.util.Date`. But the non-JDBC related code of the application uses `java.util.Date`. In such a situation, it is preferable to use FQCN of each class in the code to avoid any confusion to the reader even though you can import one package and use simple name Date to refer to the class of that package.

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

import static statement

Java has an "import static" statement that is used to import static members of a class. I will not talk about it because it is not on the exam and is not recommended to be used in professional code either.

4.3.4 Quiz

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

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

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

Which of the following approaches will work?

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

Correct answer is B and C.

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

4.4 Structure of a Java class

4.4.1 Class disambiguated

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

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

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

4.4.2 Structure of a Java source file

You are already aware of the basic structure of a Java source file from the previous chapter. I will do a quick recap and then move on to the interesting situations and gotchas that you need to know for the exam.

A Java source file has the following three parts:

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

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

Part 3: zero or more **type declarations** (i.e. class, interface, or enum definitions)

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

All of the three parts are optional. You can define multiple reference types within a single source file as well. I will talk about the rules of that later.

4.4.3 Members of a class

Within a class definition, you can have **field declarations**, **methods**, **constructors**, and **initializers**. You can also have annotations, classes, interfaces, and enums, but we will ignore all those in this book. All of these are called "**members**" of that class.

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

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

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

```

public class Student
{
    private static int count = 0; //static field
    private String studentId; //instance field

    static{ //static initializer
        System.out.println("Student class loaded");
    }

    { //instance initializer
        Student.count = Student.count +1;
        System.out.println("Student count incremented");
    }

    public Student(String id){ //constructor
        this.studentId = id;
        System.out.println(
            new Date() +
            " Student instance created. Total students created = "+count);
    }

    public String toString(){ //instance method
        return "Student[studentId = "+studentId+"]";
    }

    public static void main(String[] args) { //static method
        Student s = new Student("A1234");
        System.out.println(s.toString());
    }
}

```

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

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

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

Then there is a constructor that allows you to create **Student** objects with a given id and the static **main method** that allows you to execute this class from command line. (Notice that I have commented out the package statement so that it will be easier to execute the class from

command line without worrying about the directory structure.)

The following output is produced upon executing this class:

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

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

```
public class Student{
    String id = ""; //this is ok because this statement is a declaration

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

Comments

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

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

The following are a few examples:

```
//this is a single line comment

/*
This is a multi line
comment.
*/

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

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

JavaDoc Comments

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

```
/*
 * Observe the start of the comment. It has two *
 * Each line starts with a *
 * There is a space after each *
 * <h3>You can write HTML tags here.</h3>
 * Description of each parameter starts with @param
 * Description of the return value starts with @return
 * @see tag is used to add a hyperlink to description of another class
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see SomeOtherClassName
 */
public String sayHello(String name) {
    return "Hello, "+name;
}
```

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

4.4.4 Relationship between Java source file name and class name

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

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

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

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

A. No, you certainly can have multiple classes in a single file. But only one of them can be public and the name of that public class must be the same as the name of the file. It is okay even if there

is no public class in a file.

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

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

Q. What about interfaces? Enums?

A. The rule applies to all types, i.e., classes, interfaces, and enums. For example, you cannot have a public class and a public interface in the same file. There can be only one public type in one file. However, for the purpose of this exam, we will just focus on the classes.

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

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

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

But again, this aspect is too advanced for this exam, so, you don't have to worry about it.

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

Remember that this restriction is imposed by the Java compiler and not the JVM. Compiler converts the source code into class files and generates an independent class file for each type (irrespective of whether that type is public or not) defined in that source file. Thus, if you define three classes in Java file (one public and two non-public), three separate class files will be generated. The JVM has no idea about the Java source file(s) from which the class files originated.

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

Exam Tip

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

Directory in which source files should reside 

Although it is a common (and a good) practice to keep the source file in the directory that matches the package name in the file, there is no restriction on the directory in which the source file should reside. For example, if the package statement in your **Student.java** file is **com.university.admin**, then you should keep **Student.java** file under **com/university/admin** directory. IDEs usually enforce this convention. So, if you are using an IDE, you may see errors if you keep **Student.java** file anywhere else but remember that this is not required by the Java language. You can still compile the file from the command line.

4.4.5 Quiz 

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

Select 2 correct options.

A.

```
//Start of file
public class Student{
    public static void main(String[] args){ }
}
public class Grade{ }
//End of file
```

B.

```
//Start of file
class Student{
    public static void main(String[] args){ }
}
class Grade{ }
class Score{ }
//End of file
```

C.

```
//Start of file
public class Gradable{
}
public class Person{
}
```

```
//End of file
```

D.

```
//Start of file
class Student{
    public static void main(String[] args){ }
}
public class Professor{
}
//End of file
```

E.

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

Correct answer is B, E.

Options **A** and **C** are **incorrect** because you cannot define more than one public class in a source file. Option **D** is **incorrect** because the Professor class is public. A public class must reside in a file by the same name but here, the name of the file is `Student.java`. Option **B** is **correct** because Java allows a file to have any number of non-public types. Option **E** is correct because all of the three parts of a Java source file (i.e. package statement, import statements, and type declarations) are optional.

Observe that it is not wrong to have a main method in any class. Any class, irrespective of whether it is public or not, can have a method named main. However, if you want to execute a class, then that class must have the main method.

4.5 Advanced compilation and execution

4.5.1 Compilation and execution involving packages

Earlier, you saw how to compile and execute simple Java programs that did not have any package or import statements. Adding package and import statements greatly affect the steps required for compilation and execution. Let's see how with an 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, since the package is set to `accounting`, `accounting.Account` is the fully qualified class name of the `Account` 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 they care.

Let us create `Account.java` file and put it in our work folder. For this example, I am using `c:\javatest\` instead of `c:\Users\ceo\jfcja\chapter4` to reduce clutter in the book text. Copy the above mentioned code in the file and compile it from the command line 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
   └─ Acccount.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, you might wonder. Well, by default the Java compiler compiles the Java source file and puts the class file in the same folder as the source file. But the Java command that launches the JVM expects the class file to be in a directory path that mimics the package name. In this case, it expects the `Accounting.class` file to be in a directory named `accounting`. The accounting directory itself may lie anywhere on your file system but then that location must be on the "classpath" for the JVM to find it.

One of the many command line options that `javac` supports is the `-d` option. It directs the compiler to create the directory structure as per the package name of the class and put the class file in the right place. In our example, it creates a directory named `accounting` in the current directory and puts the class file in that directory. The dot after `-d` in the `javac` command tells the compiler that the dot, i.e., the current directory is the target directory for the resulting output. You can replace dot with any other directory and the compiler will create the new package based directory structure there. For example, the command `c:\javatest> javac -d c:\myclassfiles Account.java` will cause the accounting directory to be created in `c:\myclassfiles` folder.

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

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

Observe that when you talk about the location of a class, it is not the location of the class file that you are interested in but the location of the directory structure of the class file. Thus, in the above command line, `c:\myclassfiles` should contain the `accounting` directory and not `Account.class` file. `Account.class` should be located inside the `accounting` directory. The JVM searches in all the locations specified in the `-classpath` option for classes.

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

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

Compiling multiple source files at once

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

Contents of `A.java`:

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

Contents of `B.java`:

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

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

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

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

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

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

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

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

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

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

4.5.2 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:

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

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 space after the colon and 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
```

Although packaging classes into Jar files is not on the exam, it is good to know anyway.

4.5.3 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 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.

4.6 The java.lang and other standard Java packages

I explained earlier that one of the biggest advantages of the Java platform is the availability of a large library of readymade classes. This library makes it possible to develop applications rapidly.

The classes in the Java library are organized into several packages according to their purpose. The following table lists a few important packages, their brief description, and commonly used classes of those packages.

Package	Brief description	Commonly used classes
java.lang	Provides classes that are critical for the functioning of any Java program	Object, Math, System, Runtime, Exception, RuntimeException, wrapper classes
java.util	Provides tools and utility classes for creating commonly used data structures, internationalization, date handling	Collection, List, ArrayList, Set, Map, Date, Locale
java.io	Provides classes for performing input and output (I/O) activities involving files and other I/O devices	InputStream, OutputStream, FileReader, FileWriter, IOException
java.sql	Provides classes for dealing with relational databases	DriverManager, Connection, Statement, ResultSet
java.net	Provides classes for performing network communication	Socket, ServerSocket
java.awt and java.swing	Provides classes for building Graphical User Interfaces (GUI)	Frame, Dialog, Button, ActionEvent, LayoutManager

Table 4.6.1: Important packages in Java Standard Library

The JFCJA exam requires you to know only about the `java.lang` package but it is good to have a basic idea about other packages as well, specially the `java.util` package because almost every non-trivial Java program uses classes from this package.

The `java.lang.Object` class

The most important class in the `java.lang` package is the `java.lang.Object` class. This class is the root class, i.e., the ultimate parent, of all classes in Java. In other words, the `Object` class is the root of the Java class hierarchy. If a class doesn't extend any other class explicitly, then it implicitly extends `Object`. `Object` is the only class that doesn't have any parent. That is why it is also said that everything is an object in Java. Well, everything except primitive data types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`).

I will talk about data types in detail in the next chapter, but briefly, primitive types represent pure data without any behavior while reference types contain both - data and behavior. Unlike reference types, primitives types are not objects and so, the two are treated very differently by the Java language. This difference in their treatment by the language at a fundamental level was the cause of one of the criticisms of the Java language in its initial years. However, Java does have object oriented versions of the primitive data types as well. They are called "wrapper classes". The `java.lang` package contains `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, and `Double` classes that wrap their corresponding primitive types.

Java 1.5 introduced the feature of "**autoboxing**", that allows seamless interoperation between primitive types and their corresponding wrapper classes. For example, consider the following

code:

```
int i = 10; //10 is a primitive value and int is a primitive type
Integer iW = 10; //assigning a primitive value 10 to a reference variable iW
iW = i; //assigning a primitive variable to reference variable
i = iW; //assigning a reference variable to a primitive variable
```

The last three statements are interesting. `iW=10` and `iW=i` assign a primitive value to a reference variable, while `i = iW` assigns a reference variable to a primitive variable. This is normally not allowed. However, it works in Java because of the feature called "autoboxing". If you try to assign a primitive value to its corresponding wrapper class variable, the value is automatically wrapped into the corresponding wrapper class. Similarly, if you try to assign a wrapper class object to a primitive variable, the wrapper class object is automatically unboxed and its value is assigned to the corresponding primitive variable.

Wrapper classes and autoboxing have several nuances, which are beyond the scope of this exam. For now, you can assume that primitive values and wrapper objects are interchangeable.

The `java.lang.System` class

This is a class that you will most likely use in all of your programs because it allows you to print output to the console! Whenever you do `System.out.println`, you are using this class. Among other things, the `System` class has a variable named `out`. This variable is of type `PrintStream`, which has various `print`/ `println` methods.

The `java.lang.Math` class

The `Math` class provides methods for computing commonly used mathematical functions such as exponentials, trigonometric values, and logarithms. It also has methods for rounding decimal values.

The `java.util.Random` class

The `Random` class provides methods for generating random numbers.

Personally, I have used the `Math` and `Random` classes only a few times in my whole career (excluding this time when I am writing about them!) and so I do not find them particularly important. However, the JFCJA certification exam requires you to know about them in detail. I will discuss them with examples in a separate chapter.

Importing the `java.lang` package

Since the classes in the `java.lang` package are so important for a Java program, the requirement to explicitly import this package (or use FQCN for its classes otherwise) in any Java program is waived. The Java compiler automatically imports this package if you do not specify `import java.lang.*;` in your Java code. That is why you don't ever have to write

`java.lang.System.out.println("hello");`! Since the `java.lang` package is implicitly imported, the compiler figures out that when you use the `System` class, you mean the `java.lang.System` class.

4.7 Exercise

1. Create classes in two different named packages. Define static and instance fields in one of those classes and access those fields from the other class. See what happens when both the classes try to access the fields of each other.
2. Create a class `Admin` in package `hr` and another class `TimeCard` in package `hr.reporting` with a static method `add()`. Invoke the static method from the `Admin` class using different import statements.
3. Create a class with a `main` method and execute the class multiple times with different arguments. Include arguments that contain spaces and special characters. Print the number of arguments.
4. Write down three Java literals, any three Java keywords, and one reserved word that is neither a literal nor a keyword.
5. Write down the names of any three classes from the `java.lang` package.
6. A programmer has given the following names to a few of their classes, methods, and variables. Modify them as per the conventions used in a Java program:

```
class manager{ }
    class fullTimeEmployee{ }
    void FindPrime();
    void computemean(int[] marksobtained)
    int LOOP;  String str_lastName; Double Average;
    final int Tax_Rate;
```

Exam Objectives

1. Declare and initialize variables including a variable using final
2. Cast a value from one data type to another including automatic and manual promotion
3. Declare and initialize a String variable

5.1 Data types in Java

Java has support for two kinds of Data.

5.1.1 Data types and Variable types ↗

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

Data types are important for a programming language because they allow you tell the compiler the kind of data you want to work with. For example, when you say `int i;` you are telling the compiler that `i` is of type `int`. The compiler will then allow you to store only an integer value in this variable. Ignoring a few exceptions, it is not possible to store data of one type into a variable of another type because of the difference in the amount of space required by different data types and/or because of their compatibility.

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

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

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

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

Reference data types, on the other hand, are designed to be used when you are working with data that has a special meaning for code that Java has no knowledge of. For example, if you are developing an application for student management, you might define entities such as Student, Course, and Grade. Java has no knowledge of what Student, Course, and Grade mean. It doesn't know how much space they take, what operations they support, or what properties they have. Java will expect you to define all these things. Once you define them, you can use them to implement the business logic of your application. When you write a class, interface, or enum, you are essentially defining a reference data type. Reference data types are built by combining primitive data types and other reference data types.

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

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

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

Exam Tip

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

Name	Bits	Range	Examples	Supported operations
byte	8	-2^7 to $2^7 - 1$ i.e. -128 to 127	-1, 0, 1	mathematical, bitwise
char	16	0 to $2^{16}-1$ i.e. 0 to 65,535	0, 1, 2, 'a', '\u0061'	mathematical, bitwise
short	16	-2^{15} to $2^{15}-1$ i.e. -32,768 to 32,767	-1, 2, 3	mathematical, bitwise
int	32	-2^{31} to $2^{31}-1$	-1, 2, 3	mathematical, bitwise
long	64	-2^{63} to $2^{63}-1$	-1, 2, 3	mathematical, bitwise
float	32	approximately $\pm 3.40282347E+38F$	1.1f, 2.0f	mathematical
double	64	approximately $\pm 1.79769313486231570E+308$	1.1, 2.0	mathematical
boolean	1	true or false	true, false	logical

Table 5.1.1: Primitive data types and their sizes

Notes:

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

A word on void

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

Difference between null and void

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

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

Types of variables

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

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

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

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

Size of variables

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

Since a reference variable stores only the address of a memory location, the size of a reference variable depends on the addressing mechanism of the machine. On a system with 32 bit OS, a reference variable will be of 4 bytes, while on a 64 bit systems, it will be of 8 bytes.

Size of reference data types

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

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

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

5.2 Difference between reference variables and primitive variables

5.2.1 Reference variables and primitive variables ↗

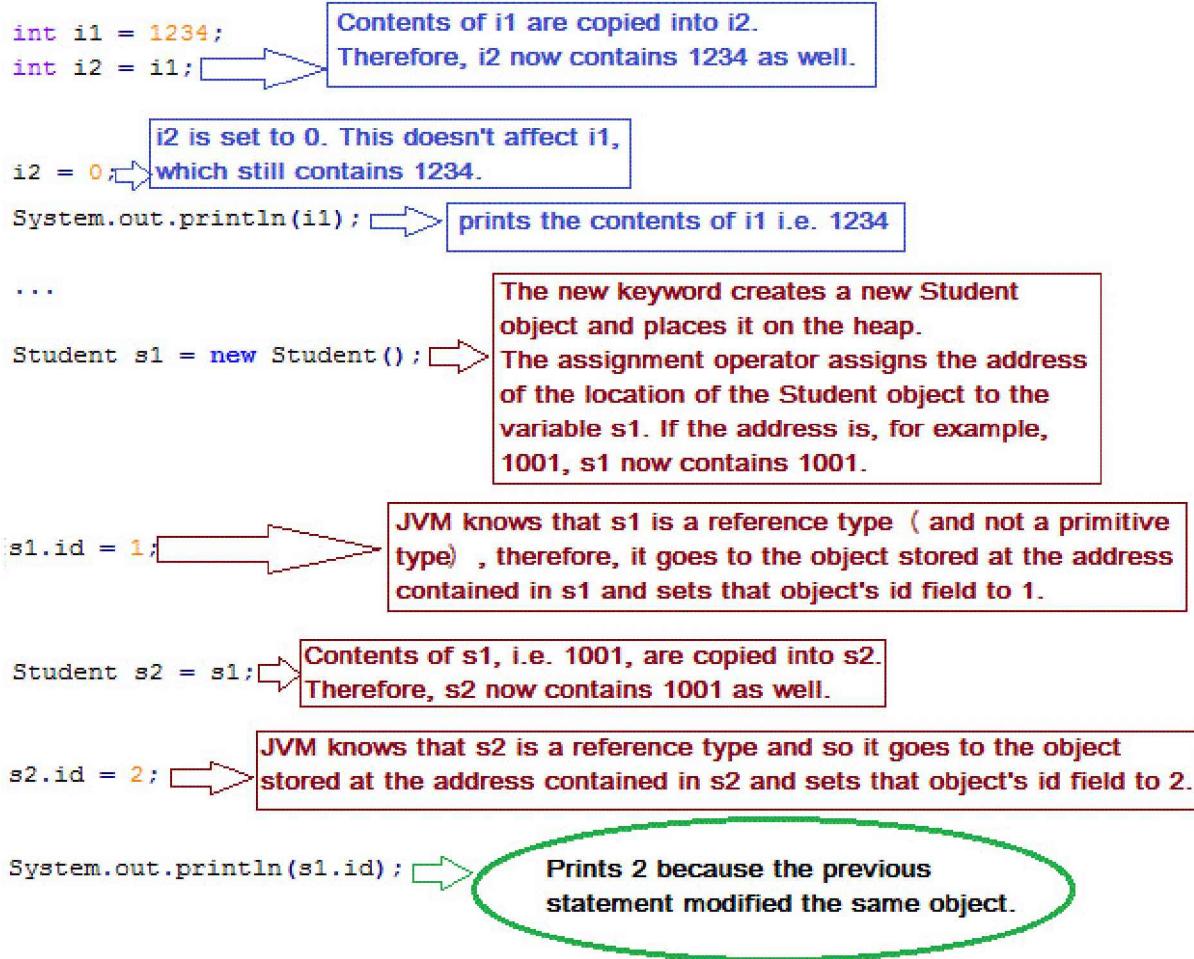
In the "3.6.5 Relation between Class, Object, and Reference" section, we discussed the relationship between a class, an object, and a reference. I explained the fundamental difference between an object reference and a primitive. To recap, there is no difference between an object reference and a primitive variable from the program memory perspective. In memory, both just store a raw number. The difference is in how the JVM interprets that raw number. In the case of a reference variable, the JVM interprets the number as an address of another memory location where the actual object is stored, but in the case of a primitive variable, it interprets the raw number as a primitive data type (i.e. a byte, char, short, int, long, float, double, or boolean). In that sense, primitives do not have references. There is nothing like a primitive "reference" because there is no object associated with a primitive variable.

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

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

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

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



Difference between reference variable and primitive variable

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

This concept is very important and you will see many questions that require you to have a clear understanding of it. The only thing that you need to remember is that a variable, be it of any kind, contains just a simple raw number. Assigning one variable to another simply copies that number from one variable to another. It is the JVM's job to interpret what that number means based on the type of the variable. Everything else just follows from this fundamental rule.

5.3 Declare and initialize variables

5.3.1 Declaration and Definition

In programming, it is important to be precise because even slight ambiguity has the potential to create serious bugs in the code. From that perspective, let's first get the difference between the terms **declaration** and **definition** right 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>

5.3.2 Declare and initialize variables

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

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

```
int x;
String str;
Object obj;
```

2. --

```
int a, b, c; //a, b, and c are declared to be of type int
String s1, s2; //s1 and s2 are declared to be type String
```

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

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

Mixing the two styles mentioned above:

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

And the following are some illegal ones:

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

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

Naming rules for a variable

A variable name must be a valid Java identifier. Conventionally however, a variable name starts with a lower case letter and names for constant variables are in upper case. Variables created by code generation tools usually start with an underscore or a dollar (_or \$) sign.

Declare and initialize a String variable

The examples given above include declaration of String variables. Since String is an important class in Java, I will discuss it separately in a chapter dedicated just for it.

5.3.3 Uninitialized variables and Default values

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

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

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

Try compiling the following code:

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

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

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

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

```
public class TestClass{
    static int i;
```

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

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

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

You can draw the following conclusions from this exercise:

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

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

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

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

```
public class TestClass{
    static int i; //i is of numeric type and is therefore, initialized to 0
    static double d; //d is a floating numeric type and is therefore, initialized to 0.0
    static boolean f; //f is of boolean type and is therefore, initialized to false
    static String s; //s is of reference type and is therefore, initialized to null
    public static void main(String[] args){
        System.out.println(i);
        System.out.println(d);
        System.out.println(f);
        System.out.println(s);
    }
}
```

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

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

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

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

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

```
public class TestClass {
    public static void main(String[] args) throws Exception {
        int val; //val not initialized here
        val = 10;
        System.out.println(val); //compiles fine
    }
}
```

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

at compile time. It can take into account the value of a variable only if the variable is a **compile time constant**. This is illustrated by the following code:

```
public class TestClass {
    public static void main(String[] args) throws Exception {
        int val;
        int i = 0; //LINE 4
        if(i == 0){
            val = 10;
        }
        System.out.println(val); //val may not be initialized
    }
}
```

This code will not compile. Even though we know that `i` is 0 and so, `i == 0` will always be true, the compiler doesn't know what the actual value of the variable `i` will be at the time of execution because `i` is not a compile time constant. Therefore, the compiler concludes that if the `if` condition evaluates to `false`, the variable `val` will be left uninitialized. In other words, the compiler notices one execution path in which the variable `val` will remain uninitialized before it is accessed. That is why it refuses to accept the print statement. If you change line 4 to `final int i = 0;` the compiler can take the value of `i` into account in its flow analysis because `i` will now be a compile time constant. The compiler can then draw the conclusion that `i==0` will always be true, that the `if` block will always be executed, and that `val` will definitely be assigned a value before it is accessed.

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

```
int val;
    int i = 0; //i is not final
    if(i == 0){
        val = 10;
    }else{
        val = 20;
    }
    System.out.println(val);
```

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

Let us now change our if condition a bit.

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

It doesn't compile. It has the exact same problem that we saw in the first version. We, by looking at the code, know that `val` will definitely be initialized in this case. We know this only because we executed the code mentally. As far as the compiler is concerned, these are two independent `if` statements. Since the compiler cannot make inferences based on the results of execution of expressions that are not compile time constants, it cannot accept the argument that `val` will definitely be assigned a value before it is accessed in the print statement.

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

5.3.4 Assigning values to variables

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

Literals

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

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

Let me list a few important rules about literals:

1. To make it easy to read and comprehend large numbers, Java allows underscores in numeric literals. For example, `1000000.0` can also be written as `1_000_000.0`. You cannot start or end a literal with an underscore. You can use multiple underscores consecutively. You need not worry about the rules governing the usage of underscores in hexadecimal, binary, and octal number formats.
2. A number without a decimal is considered an **int literal**, whereas a number containing a decimal point is considered a **double literal**.
3. A **long literal** can be written by appending a lowercase or uppercase **L** to the number and a **float literal** can be written by appending a lowercase or uppercase **f**. For example, `1234L` or `1234.0f`.

4. A `char` literal can be written by enclosing a single character within single quotes, for example, '`a`' or '`A`'. Since it may not always be possible to type the character you want, Java allows you to write a `char` literal using the hexadecimal Unicode character format ('`\uxxxx`'), where `xxxx` is the hexadecimal value of the character as defined in unicode charset. For some special characters, you can also use escape character `\`. For example, a new line character can be written as '`\n`'.

Note that writing character literals using a unicode or escape sequence is not on the exam. I have presented this brief information only for the sake of completeness.

5. There are only two boolean literals: `true` and `false`.
6. `null` is also a literal. It is used to set a reference variable to point to nothing.
7. Java allows numeric values to be written in hexadecimal, octal, as well as binary formats. In hexadecimal format (aka hex notation), the value must start with a `0x` or `0X` and must follow with one or more hexadecimal digits. For example, you could write `0xF` instead of `15`. In octal format, the number must start with a `0` and must follow with one or more ocal digits. For example, `017` is `15` in octal. In binary format, the number must start with a `0b` or `0B` and must follow with one of more binary digits (i.e. zeros and ones).Understanding of these formats is not required for the exam and so, I will not discuss these formats any further.

Assignment using another variable

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

Assignment using return value of a method

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

Assigning value of one type to a variable of another type

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

1. Simple assignments involving primitive types - This includes the assignment of compile time constants and the concept of casting for primitive variables. I will discuss this topic next.

2. Primitive assignments involving mathematical/arithmetic operators - This includes values generated using binary operators as well as compound operators, and the concept of implicit widening and narrowing of primitives. I will discuss this topic in the "Using Operators" chapter.
3. Assignments involving reference types - This expands the scope of casting to reference types. This is beyond the scope of this exam and so, I will not discuss it in this book.

Primitive assignment

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

```
byte b = 10; //b is 8 bits
char c = 'x'; //c is 16 bits
short s = 300; //c is 16 bits
int i; //i is 32 bits
long l; //l is 64 bits
float f; //f is 32 bits
double d; //d is 64 bits
//no special care is needed for any of the assignments below
i = b;
i = s;
l = i;
f = i;
d = f;
//observe that the type of the target variable is larger than the type of the source
variable in all of the assignments above.
```

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

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

```
//assuming variable declarations specified above
c = i;
i = l;
b = i;
f = d;
```

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

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

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

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

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

What if the source variable is not a constant? Since the compiler cannot determine the value held by the variable at run time, it forces the programmer to make a promise that the actual value held by the source variable at run time will fit into the target variable. This promise is in the form of a "**cast**". Java allows you to cast the value of one primitive type to another primitive type by specifying the target type within parentheses. For example, **int i = (int) 11.1;** Here, I am casting the floating point value **11.1** to an **int**. You can use a cast to assign any primitive integral (i.e. **byte, char, short, int, long**) or floating point type (i.e. **float** and **double**) value to any integral or floating point variable. You cannot cast a **boolean** value to any other type or vice versa.

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

```
int i = 10;
char c = (char) i; //explicitly casting i to char
long l = 100;
i = (int) l; //explicitly casting l to int
byte b = (byte) i; //explicitly casting i to byte
double d = 10.0;
float f = (float) d; //explicitly casting d to float
```

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

But what will happen if there is spillage?, i.e., what will happen if the actual value held by the source variable is indeed larger than the size of the target variable? What will happen to the extra value that can't fit into the target? For example, what will happen in this case - **int i = 128;**

`byte b = (byte) i; ?` The explicit cast should simply assign the value that can fit into the target variable and throw away the extra. Thus, it should just assign 127 to `b` and ignore the rest, right? Wrong! If you print the value of `b`, you will see -128 instead of 127. There doesn't seem to be any relation between 127 and -128! Understanding why this happens is not required for the exam. You will not be asked about the values assigned to variables in such cases. But I will discuss it briefly because it is useful to know.

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

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

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

Here are a few examples that make this clear:

```
char c1 = '\u0061'; //ok, unicode for 'a'
short s1 = '\u0061'; //ok, no cast needed because '\u0061' is a compile time constant
                    that can fit into a short.
short s2 = c1; //will not compile - c1 is not a compile time constant, explicit cast is
               required.
char c2 = '\uFEFO'; //ok, unicode for some character.
short s2 = '\uFEFO'; //will not compile, value is beyond the range of short.
short s3 = (short) '\uFEFO'; //ok because explicit cast is present

char c3 = 1; //ok, even though 1 is an int but it is a compile time constant whose
            value can fit into a char.
char c4 = -1; //will not compile because -1 cannot fit into a char
short s4 = -1;
```

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

Assigning float to int or double to long and vice-versa

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

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

Here are a few examples that make this clear:

```
int i = 2147483647; //Integer.MAX_VALUE
float f = i; //loses precision but ok, implicit widening of int to float is allowed
long g = 9223372036854775807L; //Long.MAX_VALUE;
double d = g; //loses precision but ok, implicit widening of long to double is allowed
```

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

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

5.3.5 final variables

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

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

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

        final TestClass tc = new TestClass();

        //x = 30; //will not compile
        //y = 40; //will not compile
        //args = new String[0]; //will not compile
        //tc = new TestClass(); //will not compile
        System.out.println(tc.x+" "+y+" "+args+" "+tc);
    }
}
```

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

You cannot reassign any value to a final variable, therefore, the four statements that try to modify their values won't compile.

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

```
class Data{
    int x = 10;
}

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

In the above code, we cannot make `d` refer to a different `Data` object once it is initialized because `d` is final, however, we can certainly use `d` to manipulate the `Data` object to which it points. If you have any confusion about this point, go through section "3.6.5 Relation between Class, Object, and Reference".

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

5.4 Declare and instantiate Java objects

5.4.1 Declare and instantiate Java objects

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

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

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

```
String str;
Object obj;
SomeClass scRef; //assuming com.abc.SomeClass has been imported
com.xyz.SomeOtherClass socRef; //assuming com.xyz.SomeClass hasn't been imported
```

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

```
SomeClass scRef1, scRef2;
```

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

```
SomeClass scRef1 = null; //scRef1 is initialized with null
SomeClass scRef2 = scRef1, scRef3; //scRef2 is initialized with the same value as
    scRef1 but scRef3 remains uninitialized
Object a = null, b = a; //a is initialized to null and then a is assigned to b
```

Here are a couple of invalid declarations:

```
String str1, Object o1; //can't change the type within a statement
Object a = b, b = null; //b is being assigned to a but b hasn't been defined at that
    point
```

Exam Tip

You will not get overly tricky questions on declaration of variables in the exam. If you stick to the basics shown above, you will not have trouble answering exam questions.

Instantiating objects

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

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

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

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

The above statement creates an instance of `java.lang.Object` class. Since a class is just a template...a cookie cutter, if you will, which is used to cut objects out of free memory, when you

use the new keyword on a class, the JVM takes a chunk of free memory and formats that memory into the various fields defined in the class. This formatted chunk of memory becomes an object of the class. You can create as many instances as you want using this template.

Depending on how a class has been defined, you may be able to (or even required to) pass argument(s) while instantiating an object. For example, the following statement creates an instance of String class with a string parameter:

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

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

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

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

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

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

Similarly, the words "create" and "instantiate" are also used interchangeably in the context of an object. So, "create an object" and "instantiate an object" mean the same thing.

Assigning objects to reference variables

Since Java is a strongly typed language, you cannot assign just any object to a reference variable of any type. The rule regarding object assignment is actually quite simple but requires an understanding of inheritance and polymorphism. Since these concepts are not covered in this exam, assume that if the type of the variable is `A`, then only an object of type `A` or a subtype of `A` can be assigned to that variable. For example, `Object obj = new String();` is valid because `String` is a subtype of `Object` but `String str = new Object();` is not valid.

Exam Tip

In the exam, you will see code that assigns an `ArrayList` object to a `List` variable. For example, `List l = new ArrayList();` This is fine because an `ArrayList` is a subtype of `List`.

5.5 Use local variable type inference

5.5.1 Local variable type inference

As of Oct 2020, the JFCJA exam is based on Java 8 and since Local Variable type inferencing was introduced in Java 10, this section is not applicable for the JFCJA exam. I have talked about it in this book because it is an interesting feature. You should skip this section if you are short on time.

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

On the other hand, too much verbosity poses problems of its own. It would be pretty annoying if you had to type `com.enthware.ets.data.MultipleChoiceQuestion q = new com.enthware.ets.data.MultipleChoiceQuestion();` (103 characters) every time you wanted declare and create a `MultipleChoiceQuestion` object. Of course, importing `com.enthware.ets.data` package reduces it to just `MultipleChoiceQuestion q = new MultipleChoiceQuestion();`. This is possible because the compiler is able to infer from the context that by `MultipleChoiceQuestion`, the developer really means `com.enthware.ets.data.MultipleChoiceQuestion`. Well, Java 10 helps you make the above statement even shorter. Java 10 onward, you can just write `var q = new MultipleChoiceQuestion();` (38 characters) and it will mean the same thing. Since you are creating an object of type `MultipleChoiceQuestion`, the compiler has no problem in inferring that the type of the variable would also be the same, i.e., `MultipleChoiceQuestion`. This is what type inference essentially means. The logic behind this shortcut is that if the compiler can unambiguously infer the type of the variable from the context, why make the programmer type it explicitly?

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

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

```
public class LVTITest1 {
    static{
        var str1 = "hello1"; //valid in static as well as instance initializers
    }
    public LVTITest1(){
        var str2 = "hello2"; //valid in constructors
    }
    public static void main(String[] args)
    {
        var i = 10; //type of i is int because 10 is an int
        var f = 1.0f; //type of f is float because 1.0f is a float
        var strA = new String[]{"a", "b"}; //type of strA is String[]
        var d = Math.random(); //type of d is double because return type of Math.random is
        double

        Object obj = "hello"; //valid, assigning a String to an Object variable
        var obj2 = obj; //type of obj2 is Object and not String

        for(var str : strA){ //type of str is String
            var p = str; //type of p is String because type of str is String
        }
        switch(strA[0]){
            case "a":
                var m = new Object(); //type of m is Object
        }
    }
}
```

Observe that all of the usages of `var` are scoped locally. Even the variable declared within the static block is local to that block. Another important point illustrated in the above code is that the type of `obj2` will be inferred as `Object` and not `String`. This is because the type of the source variable `obj` is `Object`. The compiler has no knowledge of the type of the actual object that this variable will point to at run time. Therefore, it only goes by the declared type of `obj` to infer the type of `obj2`.

Here are a few examples of invalid usages that you will be required to identify in the exam:

```
public class LVTITest2 {
    static var value1 = 10; //can't use LVTI for class members
    var value2 = 10; //can't use LVTI for instance members
    public static void main(var args) //can't use LVTI for method parameters
    {
        var p; //can't use LVTI for uninitialized variable
        var n = null; //invalid because type of null can't be determined
    }
}
```

```
var doubleArray = {1, 2}; //type must be specified in array initializer if using
LVTI for variable
var[] ia = new int[]{1, 2}; //can't apply [] to var because var is not a type
}
public static var getValue(){ //can't use LVTI for declaring return type of a method
    return "hello";
}
}
```

Observe that in the above code there is no way for the compiler to infer the type of `p` because there is no source variable or source value that is being assigned to `p` in the same statement. In other words, the context does not have enough information for the compiler to determine the type of `p`. Similarly, the compiler cannot figure out the type of `doubleArray` because the type of the value given is ambiguous. `{1, 2}` could be interpreted as an int array or a byte array also. In fact, to avoid confusion, Java prohibits type inference if the type of the array is not specified explicitly in the array initializer.

The case of fields `value1` and `value2` in the above code is interesting. It is possible to infer the type of the variables from the value. However, Java prohibits using type inference for class and instance members because such members are part of the API of a class and are therefore, used by other classes. Inferring the type instead of explicitly stating the type makes the type of the field dependent on the value that is being assigned to it. If you change this value later on, the type of the variable may change and that may adversely impact other classes.

When should var be used?

LVTI is a powerful shortcut. But you should remember that that is exactly what it is. A shortcut. It is not a keyword or even a reserved word (so, this is actually a valid line of code: `var var = 10;`). When the compiler sees `var`, it simply converts the statement to a full blown declaration. As with all shortcuts, care must be taken to ensure that its usage doesn't affect the readability of the code. Consider the following line of code appearing in a method:

```
var x = getValue();
```

Compiler can figure out the type of `x` and will accept the code happily, but can you tell the type of `x` just by looking at the above code? You can't do that unless you look at the method declaration. In fact, every time someone goes through the method, they will have to check the `getValue()` method declaration to be sure of the type of `x`. Obviously, saving a few keystrokes has reduced readability tremendously here. Now, consider the following lines of code:

```
var mapOfStateCapitals = new HashMap<String, String>();
var state_listOfTownsMap = new HashMap<String, List<String>>();
```

The usage of `var` in the above code actually improves readability by reducing clutter.

5.6 Exercise

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

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

2. What are the values of the primitive variables used in the above code. What operations can be performed on these variables?
3. Identify the methods that can be invoked using the reference variables used in the above code and also identify the objects on which those methods will be invoked.
4. Change the declaration of various instance members of Person class to include initial values.
5. Write statements declaring variables of type `byte`, `int`, `float`, and `double` and assign the values `100`, `1000`, and `1000.0` to those variables. Which assignments involve automatic promotion and which ones require manual promotion? Which assignments require a cast?

Exam Objectives

1. Use basic arithmetic operators to manipulate data including +, -, *, /, and %
2. Use the increment and decrement operators
3. Use relational operators including ==, !=, >, >=, <, and <=
4. Use arithmetic assignment operators
5. Use conditional operators including &&, ||, and ?
6. Describe the operator precedence and use of parenthesis

6.1 Java Operators

A program is nothing but an exercise in manipulating the data represented by variables and objects. You manipulate this data by writing statements and expressions with the help of operators. In that respect, operators are kind of a glue that keeps your code together. You can hardly write a statement without using any operator. Something as simple as creating an object or calling a method on an object requires the use of an operator (the new operator and the dot operator!). It is therefore, important to know about all the operators that Java has and to understand how they work.

6.1.1 Overview of operators available in Java

Java has a large number of operators. They can be classified based on the type of operations they perform (arithmetic, relational, logical, bitwise, assignment, miscellaneous) or based on the number of operands they require (unary, binary, and ternary). They may also be classified on the basis of the type of operands on which they operate, i.e., primitives (including primitive wrappers) and objects.

While, as a Java programmer, you should be aware of all of them, for the purpose of the exam, you can ignore a few of them. The following sections provide a brief description of all the operators. The ones that are not required for the exam are noted as such.

Arithmetic Operators

Arithmetic operators are used to perform standard mathematical operations on all primitive variables except boolean. They can also be applied to **wrapper objects for numeric types** (i.e. `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`) due to auto-unboxing.

Operator(s)	Brief description and Examples
<code>+, -, *, /</code> (Binary)	<p>Addition, subtraction, multiplication, and division.</p> <p>Example:</p> <pre>int a = 10; Integer b = 100; //using primitive wrapper here int c = a + b;</pre>

% (Binary)	<p>Modulus operator - returns the remainder of the division of first operand by the second one.</p> <p>Example:</p> <pre>int a = 10; int b = 3; int c = a % b;</pre> <p>c is assigned a value of 1 because when 10 is divided by 3, the remainder is 1. Here is another example -</p> <pre>Integer i = 10; Character c = 'a'; System.out.println((i%c)); //prints 10</pre> <p>The above example illustrates that these operators work on wrapper objects including Character. Don't worry, you will not be required to perform mathematical calculations involving the modulus operator in the exam. But as an exercise, you should try to find out why the above code prints 10</p>
- (Unary)	<p>Unary minus - returns a negated value of a literal value or a variable without changing the value of the variable itself.</p> <p>A unary plus may also be used on a literal or a variable but it is not really an operator because it doesn't do anything.</p> <p>Example:</p> <p>Using - on a literal :</p> <pre>int a = -10; //assigns -10 to a</pre> <p>Using - on a variable:</p> <pre>int b = -a;</pre> <p>Here, b is assigned the negated value of a i.e. $-(\text{-}10)$ i.e 10. a remains -10.</p> <p>Using + on a variable:</p> <pre>int c = +a;</pre> <p>This is valid but will not assign 10 to c. It will assign -10 to c because a is -10. a remains -10 as well.</p>

++ , -- (Unary)	<p>Unary increment and decrement operators - Unlike the unary minus operator, these operators can only be used on a variable and they actually change the value of the variable on which they are applied.</p> <p>Also unlike the unary minus, they can be applied before (pre) as well as after (post) the variable. I will explain the difference between pre and post later.</p> <p>Example:</p> <pre>int a = 10; int b = -10;</pre> <p>Post increment:</p> <pre>a++; //a is incremented from 10 to 11 b++; //b is incremented from -10 to -9</pre> <p>Pre increment:</p> <pre>++a; //a incremented from 11 to 12 ++b; //b is incremented from -9 to -8</pre> <p>(works the same way)</p>
---------------------------	--

Relational Operators

Relational operators are used to compare integral and floating point values. They can also be applied to **wrapper objects for these types** due to auto-unboxing.

Operator(s)	Brief description and Examples
<, >, <=, >= (Binary)	<p>Less than, greater than, less than or equal to, and greater than or equal to. What they do is self explanatory. They work only on numeric types and return a boolean value.</p> <p>Example:</p> <pre>int a = 10; Integer b = 100; //using primitive wrapper here boolean flag = a <b; //flag is assigned a value of true because the value of a is indeed less than the value of b. flag = (10.0 > 10); //flag is assigned a value of false because primitive values 10.0 and 10 are considered equal flag = (10.0f < 10L); //flag is assigned a value of false because primitive values 10.0f and 10L are considered equal flag = (10.0 > 10L); //flag is assigned a value of false because primitive values 10.0 and 10L are considered equal</pre>

==, !=
(Binary) **Equal to and Not equal to** - These operators are a bit special because they work on all primitive types (i.e. not just numeric ones but boolean as well) and reference types.

When used on two primitive values or a primitive value and a primitive wrapper, they check whether the two values are same or not.

Example:

```
int a = 10; Integer b = 20; char ch = 'a'; Double d = 10.0; Boolean flag
    = false;
System.out.println(a == b); //comparing an int with an Integer, prints
    false because 10 is not equal to 20
System.out.println(a == 10.0); //comparing an int with a double, prints
    true because Java considers 10 and 10.0 as equal
System.out.println(a == ch); //comparing an int with a char, prints
    false because 10 is not equal to 'a'
System.out.println(97 == ch); //comparing an int with a char, prints
    true because int value of 'a' is indeed 97
System.out.println(a != d); //comparing an int with a Double, prints
    false because a and d have the same value
System.out.println(a != 10); //comparing two ints, prints false because
    a is 10
System.out.println(false != flag); //comparing a boolean with a Boolean,
    prints false because flag is false}
```

You cannot compare a numeric value and a non-numeric value such as a `double` and a `boolean` or a primitive and a reference (unless the reference is to a primitive wrapper, of course) or even two references of “different types”. For example, the Java compiler knows that a numeric value can never be the same as a boolean value or as a reference to a non-numeric wrapper object. If a piece of code tries to make such nonsensical comparison, the compiler deems it to be a coding error. Therefore, the following statements will not compile -

```
STARTRCS
System.out.println(10 == false); //can't compare a number with a
    boolean
Object obj = new Object();
System.out.println(obj != 10); //can't compare a reference with a number
System.out.println(obj == true); //can't compare a reference with a
    boolean
Integer INT = 10;
Double D = 10.0;
System.out.println(INT == D); //can't compare an Integer reference with a
    Double reference
ENDRC
```

When used on references, `==` and `!=` check whether the two references point to the same object in memory or not. Example:

```
Object o1 = new Object();
Object o2 = o1;
boolean e = (o1 == o2); //e is assigned a value of true because o1 and
    o2 do point to the same object in memory
o2 = new Object();
System.out.println(o1 == o2); //prints false because o1 and o2 now point
    to two different objects
String s1 = "hello";
String s2 = "hello";
System.out.println(s1 == s2); //prints true because s1 and s2 point to
    the same String object
```

Comparing references using `==` and `!=` operators looks straight forward but it is a source of trick questions in the exam. There are actually two parts to it. One that deals with their usage on String references and one that deals with their usage on other references. I will discuss the first part in the “Working with the String class” chapter. The second part requires dwelling too deep into polymorphism, which is beyond the scope of the exam and so, I will not talk about it in this book.

Logical / Conditional Operators

Logical aka conditional operators are used to form boolean expressions using boolean variables and boolean values. They cannot be applied to any data type other than boolean (or Boolean).

Operator (s)	Brief description and Examples
<code>&&, </code> (Binary)	<p>Short circuiting “and” and “or”. They return a boolean value. Example:</p> <pre>boolean iAmHungry = false; boolean fridgeHasFood = false; boolean eatUp = iAmHungry && fridgeHasFood; //eat if you are hungry and if there is food in the fridge boolean tooMuchExcitement = true; boolean eatAnyway = eatUp tooMuchExcitement; //eat if eatUp is true or if there is too much excitement in the air!</pre> <p>They are called short circuiting operators because they avoid evaluating parts of an expression if the value of that part does not make any difference to the final value of the expression. In that sense, the evaluation of the second operand is “conditional”. It is evaluated only if it is required.</p> <p>Let me explain how it works with the example I gave above. You eating food depends upon two things - you being hungry and there being food in the fridge. Now, if you are not hungry, would you still go and check the fridge to see if there is food in it or not? Of course not. Since you are not hungry, you can already decide that you won’t eat food irrespective of whether there is food in the fridge or not. Thus, the second part of the expression i.e. the check for <code>fridgeHasFood</code> part, can be short circuited (i.e. not evaluated) if the first part i.e. check for <code>iAmHungry</code> is false.</p>

Similarly, if you are hungry, do you still need excitement in the air to eat food? Of course not. Since you are hungry, you can decide right there to eat food irrespective of whether there is excitement in the air or not. Therefore, even here, the second part of the expression i.e. the check for `tooMuchExcitement` can be short circuited if the first part i.e. `iAmHungry` is true.

Short circuiting behavior is helpful in cases where parts of an expression are too time consuming to evaluate. Think of the above example again. Would you get up and walk up to the fridge to see whether it is empty or not when you are not hungry? Nah, it is too much of an effort, right? Thus, if you have an expression such as `iAmHungry && checkFridge()`, where `checkFridge()` is a method that returns `true` or `false` depending on whether there is food in the fridge or not, this method won't be invoked if `iAmHungry` is `false`. Similarly, evaluating some conditions, such as those that require looking up the database, may be too time consuming and it may be desirable to avoid their evaluation if their value doesn't make a difference to the final value of the expression.

You need to understand this behavior very clearly because it gets exploited a lot while building logical expressions in professionally written code.

`&` , `|`
(Binary)

Non-Short circuiting “and” and “or” (`|` is also known as inclusive or)
Example:

```
boolean iAmHungry = false;
boolean fridgeHasFood = false;
boolean eatUp = iAmHungry & fridgeHasFood; //eat if you are hungry and
    if there is food in the fridge
boolean tooMuchExcitement = true;
boolean eatAnyway = eatUp | tooMuchExcitement; //eat if eatUp is true or
    if there is too much excitement in the air!
```

They are actually bitwise operators and are mostly used to operate on numeric types but they can also be used on boolean values just like `&&` and `||`. The only difference between the two is that they do not short circuit any part of an expression.

This behavior is useful in cases where parts of an expression has side effects that you do not want to avoid even if their value is irrelevant to the final value of the expression. For example, consider the following expression - `boolean accessGranted = authenticateUser(userid) & logToAudit(userid);`. Now, it is possible to decide that access has not to be granted if the user is not authenticated irrespective of what `logToAudit` method returns. However, you may still want to make sure every request for access is logged. Thus, you may want the `logToAudit` method to be invoked irrespective of whether `authenticateUser` method returns `true` or `false`. Usage of `&` in this case is appropriate because if you use `&&` instead of `&`, `logToAudit` method will not be invoked if `authenticateUser` method returns `false`

\wedge (Binary)	Xor aka Exclusive Or - Just like <code>&</code> and <code> </code> , this is also a bitwise operator but when applied to <code>boolean</code> operands, it returns <code>true</code> if and only if exactly one of the operands is true. Example: <pre>boolean a = false; boolean b = true; boolean c = a ^ b; //c is assigned the value true</pre> <p>The question of short-circuiting does not arise here because both the operands have to be evaluated to determine the result. In other words, it can never short-circuit an expression</p>
<code>!</code> (Unary)	Negation - This operator returns the compliment of given a <code>boolean</code> value. Example: <pre>boolean hungry = false; boolean stuffed = !hungry; //assigns true to stuffed</pre>
<code>:?</code> (Ternary)	Ternary - To be precise, “ternary” is not really the name of this operator. “Ternary” means three and since this is the only operator in Java that requires three operands, it is conveniently called as the “ternary” operator. This operator is kind of a short form for the if-else statement and has no other meaningful name. It evaluates either the second or the third operand depending on the value of the first operand. For example, <pre>int a = 5; String str = a == 5 ? "five" : "not five"; System.out.println(str); //prints five</pre> <p>There are quite a few rules that govern the type of a ternary expression and the type of each operand. I will discuss them in the next chapter along with <code>if/if-else</code></p>

Assignment Operators

Assignment operators are used to assign the value of an expression given on the right hand side to a variable given on the left hand side. There are twelve of them in total: `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and, `|=`. The first one i.e. `=` is the simple assignment operator while the rest are called compound assignment operators.

Operator(s)	Brief description and Examples
= (Binary)	<p>Simple assignment - It simply copies the value on the right to the variable on the left. In case of primitive values, it is the value of the primitive that is copied and in case of references, it is the value of the reference (not the actual object pointed to by the reference) that is copied to the variable on the left.</p> <p>Example:</p> <pre>byte b1 = 1; //assign 1 to variable b1 Object o1 = "1234"; //assign the address of the location where the string "1234" is kept, to variable o1 Object o2 = o1; //assigns the value contained in o1 to o2. Thus, o2 starts pointing to the same memory address as o1 //Note that there is only one instance of the String containing "1234" //but two variables o1 and o2 pointing to it</pre> <p>(If you are not clear about the difference between an object and a reference, I suggest you to go through section 3.6.5 Relation between Class, Object, and Reference.)</p>
*=, /=, %= +=, -=, <<=, >>=, >>>=, &=, ^=, = (Binary)	<p>Compound assignment - These operators are called compound assignment operators because they do two things at once. They perform an arithmetic or bitwise operation and then assign the result of the operation to the variable on the left. These operators work only on numeric types (primitives as well as wrappers). Unlike the simple assignment operator, these operators do not apply to boolean and reference types.</p> <p>Example:</p> <pre>int i1 = 2; int i2 = 3; i2 *= i1; //assigns the value of i2*i1 i.e. 3*2 to i2 byte b1 = 8; b1 /= 2; //assigns 4 to b1</pre> <p>The easiest way to understand how these operators work is to expand them mentally (or on a paper, if you prefer) into two different operations. For example, <code>i2 *= i1;</code> can be expanded to <code>i2 = (int) (i2 * i1);</code> Similarly, <code>b1 /= 2;</code> can be expanded to <code>b1 = (byte)(b1 / 2);</code> Notice the explicit cast in the expanded form. I will explain its reason later while discussing numeric promotion.</p> <p>The <code>+=</code> operator is overloaded to work with Strings as well. It combines String concatenation and assignment in one step.</p> <p>Example:</p> <pre>String s = "hello"; s += " world"; //creates a new String "hello world" and assigns it back to s</pre> <p>Just like the other compound assignment statements mentioned above, the easiest way to evaluate it is to expand it like this:</p> <pre>s = s + " world";</pre>

While the primary function of an assignment operator is quite simple, there are a few nuances about these operators that you should know -

1. They are all **right associative**, which means `a = b = c = 10;` will be evaluated as `a = (b = (c = 10))` instead of `((a = b) = c) = 10.` I will discuss this separately in the “Operator

- precedence and evaluation of expressions” section.
2. The left operand of these operators must be a variable. It can either be a named variable (for example, a local variable or a field of an object) or a computed variable (for example, an array element). Thus, you cannot do something like `10 = b;` because `10` is not a variable that can be assigned a value. But more importantly, you cannot do something like `aMethodThatReturnsAnObject() = 20;` either because a method returns value of a reference and not the reference variable or the object itself. Note that this is a direct implication of the fact that Java does not have “**pass-by-reference**”.
 3. The right operand of these operators must be an expression whose type must be same as or “compatible” with the type of the target variable on the left. For example, you cannot assign `boolean` expression to an `int` variable. In case of primitive types, compatibility is easy to understand. You can assign a value of any numeric type to a variable of any numeric type if the type of the value fits within the range of the type of the variable. Otherwise, you have to use a cast. I have already discussed this in “Working With Java Data Types” chapter. Compatibility in case of references is a bit complicated. I will talk about this more while discussing polymorphism and subclassing.

Bitwise Operators

(Not required for the exam but good to know)

Bitwise operators are used to apply logical operations on individual bits of given numeric values (including their respective wrapper objects). I will not discuss them in detail because they are not required for the exam. They are quite straight forward to apply though.

(If you want to understand these operators better, try applying them to various numeric values and use `Integer.toBinaryString` method to print out the bit pattern of any given numeric value.)

Operator(s)	Brief description and Examples
<code>&, , ^</code> (Binary)	<p>Bitwise “and”, “or”, and “xor”</p> <p>Example:</p> <pre>byte b1 = 1; //bit pattern of 1 is 00000001 byte b2 = 2; //bit pattern of 2 is 00000010 byte b3 = (byte) (b1 & b2); //b3 gets 0 byte b4 = (byte) (b1 b2); //b4 gets 3 byte b5 = (byte) (b1 ^ b2); //b5 gets 3</pre> <p>(I will explain the reason for explicit casting of the results to byte later while discussing numeric promotion.)</p>
<code>~</code> (Unary)	<p>Bitwise complement - It toggles i.e. turns a 0 to 1 and a 1 to 0, individual bits of a given numeric value (including primitive numeric wrapper objects)</p> <p>Example:</p> <pre>byte b1 = 1; //bit pattern of 1 is 00000001 byte b2 = (byte) ~b1; //b2 gets 11111110, which is -2</pre>

>>, << (Binary)	<p>Bitwise signed right and left shift - They shift the given bit pattern of the left operand towards right or left by number of places specified by the right operand while keeping the sign of the number same.</p> <p>Example:</p> <pre>byte b1 = -4; //bit pattern of 4 is 11111100 byte b2 = 1; byte b3 = (byte) (b1 >> b2); //b3 gets 11111110 i.e. -2 byte b4 = (byte) (b1 << b2); //b4 gets 11111000 i.e. -8</pre> <p>Observe that when we shifted the bits of -2 to right by 1 place, the sign bit (i.e. the left most bit) got copied over</p>
>>> (Binary)	<p>Bitwise unsigned right shift- This operator works the same way as <code>>></code> but it does not carry forward the sign bit. It pushes in zeros from the left irrespective of the sign of the number. Thus, a negative int will become a positive int.</p> <p>Example:</p> <pre>int i = -4; //bit pattern of -4 is 11111111 11111111 11111111 11111100 (32 bits) int i2 = 1; int i3 = i1 >>> i2; //i3 gets 01111111 11111111 11111111 11111110 i.e. 2147483646</pre> <p>Observe that when we shifted the bits of -4 to right by 1 place, the sign bit (i.e. the left most bit) did not get copied over. Also observe that I have used <code>int</code> instead of <code>byte</code> variables in this example to prevent numeric promotion and casting from affecting the result. Again, this is not required for the exam, but you should try out this example with <code>byte</code> variables instead of <code>int</code> and compare the values of <code>i3</code> using <code>Integer.toBinaryString</code> method.</p>

Miscellaneous Operators

Operator (s)	Brief description and Examples
+	<p>String concatenation - + operator can also be used to concatenate two strings together.</p> <p>Example:</p> <pre>String s1 = "hello "; String s2 = " world"; String s = s1 + s2; //creates a new String "hello world"</pre> <p>The above example illustrates the most straight forward use of the + operator. However, you will learn soon that this operator is quite versatile and can be used to join any kind of object or primitive value with a String.</p>
. (dot)	<p>The dot operator - You have definitely seen it but most likely have not noticed it :) Anytime you call a method using a reference variable or access a member of a class or an object, you use the dot operator. It doesn't do anything else but it is an operator nonetheless. It is applied to a reference to access the members of the object pointed to by that reference.</p> <p>It will always throw a <code>NullPointerException</code> if you apply it to a null reference (i.e. a reference that is not pointing to any object)</p>
()	<p>The cast operator - This operator can be used on numeric values and references. You have already seen its use on numeric values while assigning values of a larger type to variables of a smaller type. For example, <code>int i = (int) 1.0;</code>.</p> <p>When used on a reference, it casts the reference of one type to another. A discussion on this aspect is beyond the scope of this book.</p>
[]	<p>The array access operator - This operator is used access the elements of an array. I will discuss it in the “Arrays and ArrayList” chapter.</p>
instanceof	<p>instanceof - The <code>instanceof</code> operator is used to check whether an object pointed to by a reference variable given on the left is of the type given on the right of this operator. It returns <code>true</code> if the object pointed to by the reference variable on the left is of the type (or a subtype) of the type given on the right and <code>false</code> otherwise. This operator is not required for the JFCJA exam and so, I will not talk about it in this book.</p>
->	<p>The lambda operator - This operator is used to write lambda expressions. It is also not in scope for the JFCJA exam.</p>

6.1.2 Expressions and Statements

Difference between an Expression and a Statement

Before I move on to the intricacies of various operators, let me talk a bit about **expressions** and **statements**. This will give you a good perspective on how various operators work and why they work so.

Wikipedia defines an **expression** as a combination of one or more explicit values, constants,

variables, operators, and functions that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce ("to return", in a stateful environment) another value.

The point to note here is that an **expression has a value**. This value could be a primitive value or a reference. You can combine these values together using various operators to create even bigger expressions as per the rules of the language. You can say that if something has a value that can be assigned to a variable, then that something is an expression.

A **statement**, on the other hand, is a complete line of code that may or may not have any value of its own. You cannot combine statements to produce another statement. You can, of course, write one statement after another to create a **program**.

Thus, an expression may be a statement on its own. For example, consider the following code:

```
public class TestClass{
    public static void main(String[] args){
        int a = 10;
        int b = 20;
        a + b; //this line will not compile
        a = a + b;
    }
}
```

In the above code, `int a = 10;` is a statement. `int b = 20;` is another statement. But neither of them are expressions because they don't have a value of their own. You cannot assign `int a = 10` to a variable. On the other hand, `a + b` is an expression but is not a valid statement. If you try to compile the above code, you will get an error saying "**Not a statement**" at line `a + b;`

However, `a = a + b;` is a valid statement as well a valid expression. It is, in fact, an expression made by combining two expressions - `a` and `a + b` using the `=` operator. Furthermore, `a + b` is also an expression made by combining two expressions - `a` and `b` using the `+` operator.

The question that should pop into your head now is, if `a = a + b` is a valid expression, does it have a value? Yes, it does. As a matter of fact, the value of an expression built using the assignment operator is the same as the value that is being assigned to the variable on the left side of the `=` operator. In this case, for example, `a` is being assigned a value of `a + b`. Thus, the value of the expression `a = a + b` is the value produced by the expression on the right of `=` operator, i.e., `a + b`. You can actually test it out by assigning this whole expression to another variable like this - `int k = (a = a + b);`

What values can be combined using which operators is really the subject of this chapter.

6.1.3 Post and Pre Unary Increment/Decrement Operators

The `++` and `--` operators can be applied to a **variable** in the **postfix** form (i.e. appearing after the variable) or in the **prefix** form (i.e. appearing before the variable). In both the cases, the value of the variable will be incremented (or decremented) by `1`. The difference between the two is that the postfix operator returns the existing value of the variable while prefix operator returns the updated value of the variable. This is illustrated in the following code:

```
int i = 1, post = 0, pre = 0;
post = i++;
System.out.println(i+" "+post); //prints 2, 1
i = 1; //resetting to i back to 1
pre = ++i;
System.out.println(i+" "+pre); //prints 2, 2
```

To understand this, you need to look at process of evaluation of the expression `post = i++` step by step. The expression `post = i++`; is composed of two parts - the variable `post` and the expression `i++` - joined using the assignment (`=`) operator. To evaluate this expression, you need to first evaluate the expression that is on the right side of `=`, i.e., `i++`.

Now, since a postfix operator increments the variable but returns the existing value of the variable, and since the existing value of the variable `i` is `1`, the expression `i++` evaluates to `1` even though the value of `i` has been incremented to `2`. Thus, the variable `post` is assigned a value of `1`.

Let us follow the same process for evaluating the expression `pre = ++i`. Here, since the prefix operator increments the variable and returns the updated value of the variable, and since the updated value of the variable `i` is `2`, the expression `++i` evaluates to `2`. Thus, the variable `pre` is assigned a value of `2`.

You need to appreciate the fact that the value of the variable `i` and the value of the expression `i++` (or `++i`) are two different things and they may or may not be the same depending on whether you use pre or post form of the increment/decrement operator. When you do `int x = ++i;` you are not assigning the value of the variable `i` to `x`. You are assigning the value of the expression `++i` to `x` independently from the process of applying `++` to `i`. If you truly understand this concept then you should be able to determine the output of the following code:

```
int i = 1;
i = i++;
System.out.println(i); //what will this line print?
```

postfix and prefix for the exam

In the exam, you may get questions where you are required to evaluate a compound expression containing multiple pre/post increment operators. They do get tricky sometimes but the key to solving such problems is to apply the concept explained above without losing focus. I will now show you a representative question and the steps to work out the answer.

Q. What will the following code print?

```
int a = 2;
int b = 5;
int c = a * (a++ - --b) * a * b;
System.out.println(a+" "+b+" "+c);
```

1. We start with putting the values of the variables in the expression from left to right. Since the value of **a** at the beginning of the evaluation is **2**, the expression becomes:

```
c = 2 * (a++ - --b) * a * b
```

2. Next, **a** is incremented using the post increment operator. Therefore, the value of **a** used in the expression will be the existing value of **a**, i.e., **2** and then **a** is incremented to **3**. Therefore, the expression now becomes:

c = 2 * (2 - --b) * a * b; (a is 3 now)

3. **b** is decremented using the pre decrement operator. Therefore, **b** is decremented first to from **5** to **4** and then the new value of **b**, i.e., **4** is used in the expression. Therefore, the expression becomes:

c = 2 * (2 - 4) * a * b; (a is now 3 and b is now 4)

4. There are no further operations left to be applied on **a** and **b** anymore so, their values can be substituted in the expression. The expression now becomes:

c = 2 * (2 - 4) * 3 * 4;

5. Now you can apply the usual rules of operator precedence and parentheses to evaluate the expression. You can easily see that the value of **2 * (2 - 4) * 3 * 4** is **-48**, which is the value that is assigned to the variable **c**.

6. Thus, the print statement will print **3 4 -48**

The key point that you should observe in the evaluation process described above is how the value of the variables change while the expression is being evaluated and their impact on the expression. For example, the value of variable **a** changed from **2** to **3** while the expression was being evaluated. This change causes part of the expression to use old value of **a** while the subsequent part to use the updated value of **a**. Although a similar change is not noticeable in case of **b** in this expression because **b** is decremented using pre decrement operator instead of post decrement, had there been a use of **b** in the expression before encountering **--b**, the old value of **b** would have been used just like in the case of **a**.

To test this theory, try to compute the value of this slightly modified expression:

```
c = b * a * (a++ - --b) * a * b
```

When to use postfix and when to use prefix?

As shown in the code above, the difference between the two is very subtle and is material only when you use the unary increment and decrement operators inside of another expression. So, the obvious answer is to use postfix when you want to use the existing value of the variable and then update it, and use prefix when you want to update the value first and then use its value.

Ideally, however, you should avoid using these operators in a compound expression altogether because they cause confusion and are a common source of hard to find bugs in the code. My advice is to get into the habit of using just one form of increment or decrement operator in all of your code.

Using unary increment/decrement operators on wrappers

You can use `++` and `--` operators on any numeric primitive wrapper and they work the same way as explained above. You should, however, remember that primitive wrapper objects are immutable and therefore, incrementing or decrementing a wrapper object does not change that particular wrapper object. Instead, a new wrapper object is assigned to the reference on which increment or decrement operator is applied. The following code illustrate this point:

```
Integer i = 1;
Integer j = i; //now, i and j point to the same Integer object
i++; //a different Integer object containing 2 is assigned to i
//j still points to the same Integer object containing 1
System.out.println(i+" "+j); //prints 2 1
```

6.1.4 String concatenation using `+` and `+=` operators

The `+` operator is quite versatile. Besides performing the mathematical addition of numeric operands, it can also "add" two Strings together to create a new String. In that sense, you can say that the `+` operator is overloaded because its behavior changes based on the type of operands. When both of its operands are numeric values (or their primitive wrappers), it performs the mathematical addition but if either of its operands is of type String, it performs the String addition. By the way, the technical term for String addition is "**concatenation**" and so, I will use this term from now onward.

To trigger the String concatenating behavior of the `+` operator, the declared type of at least one of its operands must be a String. If one of the operands is a String and the other one is not, the other operand will be converted to a String first and then both the operands will be concatenated to produce the new String. Let us see a few examples to make it clear:

```
String s1 = "hello" + " world"; //both the operands are Strings
System.out.println(s1); //prints "hello world"

String s2 = "hello " + 1; //first operand is a String and second is an int
System.out.println(s2); //prints "hello 1"

Double d = 1.0;
String s = "2";
```

```
String s3 = d + s; //first operand is a Double and second is a String
System.out.println(s3); //prints "1.02"
```

There is no restriction on the type of the non-String operand. It can be of any type. The interesting part is how the non-String operand is converted to a String. The answer is simple. Recall that **java.lang.Object** is the root class of all objects in Java. This class contains a `toString` method that returns a String representation of that object. The `+` operator invokes this `toString` method on the non-String operand to get a String value. If the non-String operand is a primitive, then the primitive value is first converted to its corresponding wrapper object and `toString` is invoked on the resulting wrapper object. The following program illustrates this process.

```
public class TestClass{
    public static void main(String[] args){
        TestClass tc = new TestClass();
        String str = tc.toString();
        System.out.println( str );
        System.out.println( "hello " + tc );
    }
}
```

This program produces the following output:

```
TestClass@15db9742
hello TestClass@15db9742
```

As you can see, the `TestClass@15db9742` part of the concatenated String on the second line of the output is the same as the String returned by the `toString` method.

The `+` operator has one more trick up its sleeve. Check out this code:

```
public class TestClass{
    public static void main(String[] args){
        TestClass tc = null;
        System.out.println( "hello " + tc );
    }
}
```

It prints `hello null`. Normally, when you invoke any method using a null reference, the JVM throws a **NullPointerException**. But in this case, no `NullPointerException` was thrown. How come? The reason is that the `+` operator checks whether the operand is `null` before invoking `toString` on it. If it is null, it uses the String `"null"` in place of that operand.

String concatenation using `+=` operator

The `+=` operator works in the same way as the `+` operator but with the additional responsibility of an assignment operator. The operand on the left of `+=` must be a String variable but the operand on right can be a value or variable of any type. For example,

```
String str = "2";
str += 1; //this is the same as str = str + 1;
```

```
System.out.print(str); //prints "21"
```

Here is another example:

```
public class TestClass{
    public static void main(String[] args){
        TestClass tc = new TestClass();
        String str = null;
        str += tc; //same as str = str + tc;
        System.out.print( str ); //prints nullTestClass@15db9742
    }
}
```

Here, even though `str` is `null` and the second operand to `+=` is not a `String`, the `String` concatenation behavior of `+=` will be triggered because the declared type of `str` is `String`. Thus, `toString` will be called on the non-string operand `tc`. Furthermore, since `str` is `null`, the `String` "null" will be used while concatenating `str` and the string value returned by the call to `tc.toString()`.

You need to keep in mind that compound assignment operators do not work in declarations. For example, the statement `String str += "test";` will not compile because `str` is being declared in this statement.

6.1.5 Numeric promotion and casting

Take a look at the following code:

```
public class TestClass{
    public static void main(String[] args){
        byte b = 1;
        short s = -b;
        System.out.println(s);
    }
}
```

Looks quite simple, right? One may believe that it will print `-1`. But the fact is that it doesn't compile! It produces the following error message:

```
TestClass.java:4: error: incompatible types: possible lossy conversion from int to short
    short s = -b;
               ^
1 error
```

There is no use of `int` anywhere in the code, yet, the error message is talking about converting something from `int` to `short`. The reason is that Java applies the rules of "**numeric promotion**" while working with operators that deal with numeric values, which are, in a nutshell, as follows:

- Unary numeric promotion** - If the type of an operand to a unary operator is smaller than `int`, the operand will automatically be promoted to `int` before applying the operator.

2. Binary numeric promotion - Both the operands of a binary operator are promoted to an `int` but if any of the operands is larger than an `int`, the other operand is promoted to the same type as the type of the larger one. Thus, for example, if both the operands are of type `byte`, then both are promoted to `int` but if any of the operands is a `long`, `float`, or `double`, the other one is also promoted to `long`, `float`, or `double`, respectively. Similarly, if the operands are of type `long` and `float`, `long` will be promoted to `float`.

A direct implication of the above two rules is that the result of applying an operator to numeric operands is of the same type as the type of the larger operand but it can never be smaller than an `int`.

If you look at the above code in light of these rules, the error message is quite obvious. Before applying the unary minus operator on `b`, the compiler promotes `b` to `int`. The result of applying the operator is, therefore, also an `int`. Now, `int` is a larger type than `short` and the compiler is concerned about possible loss of information while assigning a `int` value to a `short` variable and so, it refuses to compile the code. The language designers could have easily allowed such assignments without a fuss, but an inadvertent loss of information is often a cause of bugs, and so, they decided to make the programmer aware of this issue at the compile time itself.

You may wonder here that the value of `b` is `1` and the result of `-b` is just `-1`, which is well within the range of `short`. Then what's the problem? What loss of information is the compiler talking about? You need to realize that compiler does not execute any code. It can't take into account the values that the variables may take at run time, while making decisions. So, even though you know that the result of the operation is small enough to fit into a `short`, the compiler cannot draw the same inference at compile time.

Let us now take look at few more examples where numeric promotion plays a role:

```
short s1 = 1;
byte b1 = 1;
byte b2 = 2;

short s2 = +s1; //won't compile because the result will be an int
byte b = s1 + 2; //won't compile because the result will be an int
b = b1 & b2; //won't compile because the result will be an int

s2 = s1 << 1; //won't compile because the result will be an int
s2 = s1 * 1; //won't compile because the result will be an int

float f = 1.0f; //recall that to write a float literal you have to append it with an f
                 or an F
double d = 1.0;

int x = f - 1; //won't compile because the result will be a float
float f2 = f + d; //won't compile because the result will be a double
```

All of the above operations are affected by the rule of numeric promotion and therefore, fail to

compile. To make them compile, you need to assure the compiler that you know what you are doing, and that you are okay with the possible loss of information if there is any. You give this assurance to the compiler by explicitly casting the result back to the type of target variable. I have already discussed casting of primitives in the "Working with Java Data Types" chapter. The following code is the fixed version of the code shown above:

```
short s1 = 1;
byte b1 = 1;
byte b2 = 2;

short s2 = (short) +s1;
byte b = (byte) (s1 + 2);
b = (byte) (b1 & b2); //numeric promotion happens for bit-wise operators as well
s2 = (short) (s1 << 1);
s2 = (short) (s1 * 1);

float f = 1.0f; //recall that to write a float literal you have to append it with an f
    or an F
double d = 1.0;

int x = (int) (f - 1);
float f2 = (float) (f + d);
```

An important point to note here is these rules come into picture only when the operands involve a variable and not when all the operands are constants and the result of the operation lies within the range of the target variable. The need for promotion does not arise while dealing with constants because the values are known at compile time and therefore, there is no possibility of loss of information at run time. If the value produced by the constants doesn't fit into the target variable, the compiler will notice that and refuse to compile it. Thus, the statement `byte b = 200 - 100;` will compile fine because `200` and `100` are compile time constants and the result of the operation fits into a `byte` even though one operand of the `-` operator falls outside the range of `byte`. But `byte b = 100 + 100;` will not compile because the result of `100 + 100` cannot fit into a `byte`. Similarly, the following code will also compile without any error:

```
final int I = 10;
byte b = I + 2;
```

No cast is needed because `I` is a compile time constant and the result of `I + 2` fits into a `byte`.

Curious case of unary increment/decrement and compound assignment operators

To put it simply, the rules of numeric promotion do not apply to `++`, `--` and the compound assignment operators such as `+=`, `-=` and `*=`. They are the exceptions to the rules of numeric promotion. Thus, the following statements will compile fine without any explicit cast.

```
byte b1 = 1;
byte b2 = ++b1; //result of ++b1 will be a byte
b2 = b1--; //result of b1-- will be a byte
b1 *= b2; //result will be a byte
```

```
double d = 1.0;
float f = 2.0f;
f += d; //result of will be a float
```

Observe the statements `b1 *= b2;` and `f += d;`. They behave as if they are the short hand for `b1 = (byte)(b1*b2);` and `f = (float)(f+d);` In other words, the result of a compound assignment operation is implicitly cast back to the target type irrespective of the type of the second operand. Similarly, `++b1;` behaves like `b1 = (byte) (b1 + 1);`

One could certainly make the "source of bugs" argument in the case of compound assignments as well. I guess, convenience superseded that argument in this case :)

Numeric Promotion and Primitive Wrapper Objects

Remember that to apply an operator to wrapper objects, they have to be unboxed first into their respective primitive values. Thus, the rules of numeric promotion and their exceptions will come into play here as well. Therefore, for example, `Byte bW = 1; bW = -bW;` will not compile but `--bW;` will compile fine due to the presence of an implicit cast as explained before.

Be careful with the usage of `final` on wrapper variables as compared to the usage of `final` on primitive variables. For example, the following code will not compile even though `b1` is `final`:

```
final Byte b1 = 1;
Byte b2 = -b1; //will not compile even though b1 is final
```

This doesn't work because `b1` is a reference to an object. It is this reference that is `final`, not the contents of the object to which it points. We know that wrapper objects are immutable and therefore, a `Byte` object's value will not change once it is initialized but the compiler doesn't know that. Thus, as far as the compiler is concerned, it is not sure that the result of `-b1` will fit into a `byte` and so, it refuses to accept the assignment. Compare this with the code shown earlier where we were able to assign a `final int` variable to a `byte` variable.

6.1.6 Operator precedence and evaluation of expressions

I am sure you have come across simple mathematical expressions such as $2 + 6 / 2$ at school. You know that this expression evaluates to 5 and not 4 because division has higher precedence than addition and so, $6 / 2$ will be grouped together instead of $2 + 6$. To change the default grouping, you use brackets (aka parentheses), i.e., $(2+6)/2$. You have most likely also come across the acronym **BODMAS** (or PEDMAS, in some countries), which stands for Brackets/Parentheses, Orders/Exponents, Division, Multiplication, Addition, and Subtraction). It helps memorize the conventional precedence order in which brackets are evaluated first, followed by powers, and then the rest in that order.

Java expressions are not much different from mathematical expressions. Their evaluation is determined by similar conventions and rules. The only problem is that there are a lot of operators to worry about and, as we saw earlier, the operators are not just mathematical. They are logical, relational, assignment, and so many other types as well. This makes Java expression evaluation a

lot more complicated than a mathematical expression. But don't worry, you will not be required to evaluate complicated expressions in the exam. But you still need to know a few basic principles of expression evaluation to analyze code snippets presented in the questions correctly.

Precedence

Precedence determines which operator out of two is evaluated "first", in a conceptual sense. Another way to put it is, precedence determines how tightly an operator binds to its operands as compared to the other applicable operator in an expression. For example, in the case of `2 + 6 / 2`, the operand `6` can be bound to `+` or to `/`. But the division operator, having higher precedence than addition operator, binds to an operand more tightly than the addition operator. The addition operator, therefore, is not able to get hold of `6` as its second operand and has to wait until the division operator is done with it. In an expression involving multiple operators, the operator with highest precedence gets the operand, followed by the operator with second highest precedence, and so on.

The following table shows the precedence of all Java operators:

Operator Name (In order of precedence from High to Low)	Operator
Array access, object creation, member access, method reference	<code>[], new, ., ::</code>
Postfix	<code>expr++, expr--</code>
* Unary and pre-increment/pre-decrement	<code>+expr, -expr, !, ~, ++expr, --expr</code>
* Cast	<code>()</code>
Multiplicative	<code>*, /, %</code>
Additive	<code>+, -</code>
Shift	<code><<, >>, >>></code>
Relational	<code><, >, <=, >=, instanceof</code>
Equality	<code>==, !=</code>
Bitwise AND	<code>&</code>
Bitwise exclusive OR	<code>~</code>
Bitwise inclusive OR	<code> </code>
Logical AND	<code>&&</code>
Logical OR	<code> </code>
Ternary	<code>? :</code>
* Assignment	<code>=, +=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=, >>>, =</code>
* Lambda and switch expression	<code>-></code>
* These operators associate from Right to Left.	

An important thing to observe from the above table is that the access operator, postfix increment/decrement and the cast operators have a higher precedence than the assignment operators and the lambda operator.

This explains why the following code doesn't compile:

```
int i = 0;
```

```
byte b = (byte) i + 1;
```

Since the cast operator has higher precedence than the `+` operator, `i` is first cast to `byte` and then the addition is performed. The end result, therefore, is an `int` instead of a `byte`. You need to put `i + 1` in parentheses like this:

```
byte b = (byte)(i + 1);
```

Associativity

Associativity of operators determines the grouping of operators when an expression has multiple operators of same precedence. For example: the value of the expression `2 - 3 + 4` depends on whether it is grouped as `(2 - 3) + 4` or as `2 - (3 + 4)`. The first grouping would be used if `-` operator is **left-associative** and the second grouping would be used if `-` operator is **right-associative**. It turns out that operators are usually grouped in the same fashion in which we read the expression, i.e., from **left to right**. In other words, most of the operators in Java are defined to be **left-associative**. The notable exceptions are the assignment operators (simple as well as compound), the ternary operator, and the lambda and switch expression operators. Thus, the expression `2 - 3 + 4` will be grouped as `(2 - 3) + 4` and will evaluate to 3. But the expression `a = b = c = 5;` will be grouped as `a = (b = (c = 5)) ;` because the assignment operator is right associative. Here is another example that shows the impact of associativity:

```
String s1 = "hello";
int i = 1;
String s2 = s1 + 1 + i;
System.out.println(s2); //prints hello11
```

The above code prints `hello11` instead of `hello2` because the `+` operator is left-associative. The expression `s2 = s1 + 1 + i;` is grouped as `s2 = (s1 + 1) + i;`. Thus, `s1+1` is computed first, resulting in the string `hello1`, which is then concatenated with `1`, producing `hello11`.

A programming language could easily prohibit ambiguous expressions. There is no technical necessity for accepting the expression `2 + 6 / 2` as valid when it can be interpreted in two different ways. The only reason ambiguous expressions are accepted is because it is considered too onerous for the programmer to resolve all ambiguity by using parenthesis when a convention already exists to evaluate mathematical expressions. Rules of Operator Precedence and Associativity are basically a programming language extension to the same convention that includes all sorts of operators. You can, therefore, imagine that operator precedence and evaluation order are used by the compiler to insert parenthesis in an expression. Thus, when a compiler see `2 + 6 / 2`, it converts the expression to `2 + (6 / 2)`, which is what the programmer should have written in the first place.

You should always use parenthesis in expressions such as `2 - 3 + 4` where the grouping of operands is not very intuitive.

Parenthesis

You can use parentheses to change how the terms of an expression are grouped if the default grouping based on precedence and associativity is not what you want. For example, if you don't want $2 - 3 + 4$ to be grouped as $(2 - 3) + 4$, you could specify the parenthesis to change the grouping to $2 - (3 + 4)$.

Evaluation Order

Once an expression is grouped in accordance with the rules of precedence and associativity, the process of evaluation of the expression starts. This is the step where computation of the terms of the expression happens. In Java, expressions are evaluated from left to right. Thus, if you have an expression `getA() - getB() + getC()`, the method `getA` will be invoked first, followed by `getB` and `getC`. This means, if the call to `getA` results in an exception, methods `getB` and `getC` will not be invoked.

Java also makes sure that operands of an operator are evaluated fully before the evaluation of the operator itself. Obviously, you can't compute `getA() + getB()` unless you get the values for `getA()` and `getB()` first.

The important point to understand here is that evaluation order of an expression doesn't change with grouping. Even if you use parentheses to change the grouping of `getA() - getB() + getC()` to `getA() - (getB() + getC())`, `getA()` will still be invoked before `getB()` and `getC()`.

Let me show you another example of how the above rules are applied while evaluating an expression. Consider the following code:

```
public class TestClass{  
    static boolean a ;  
    static boolean b ;  
    static boolean c ;  
    public static void main(String[] args) {  
        boolean bool = (a = true) || (b = true) && (c = true) ;  
        System.out.println(a + ", " + b + ", " + c );  
    }  
}
```

Can you tell the output? It prints `true, false, false`. Surprised?

Many new programmers think that since `&&` has higher precedence, `(b = true) && (c = true)` would be evaluated first and so, it would print `true, true, true`. It would be logical to think so in a Mathematics class. However, evaluating a programming language expression is a two step process. In the first step, you have to use the rules of precedence and associativity to group the terms of the expression to remove ambiguity. Here, the operand `(b = true)` can be applied to `||` as well as to `&&`. However, since `&&` has higher precedence than `||`, this operand will be applied to `&&`. Therefore, the expression will be grouped as `(a = true) || ((b = true) && (c = true))`. After this step, there is no ambiguity left in the expression. Now, in the second step, evaluation of the expression will start, which, in Java, happens from left to right. So, now, `a = true` will be evaluated first. The value of this expression is `true` and it assigns `true` to `a` as well. Next,

since the first operand of `||` is true, and since `||` is a short circuiting operator, the second operand will not be evaluated and so, `(b = true) && (c = true)` will not be executed.

6.2 Exercise

1. Work out the values of the variables after each of the following statements on paper:

```
String str = "7" + 5 + 10;
str = 7 + 5 + "10";
str = "7" + (5 + 10);

int m = 50;
int n = ++m;
int o = m--;
int p = --o+m--;
int x = m<n?n<o?o<p?p:o:n:m;

int k = 4;
boolean flag = k++ == 5;
flag = !flag;
```

2. Which of the following lines will fail to compile and why? Write down the value of the variables after each line.

```
byte b = 1;
b = b<<1;
int c = b<<1;
byte d += b;
byte e = 0;
e += b;
```


Exam Objectives

1. Using Decision Statements
2. Use the decision making statement (if-then and if-then-else)
3. Use the switch statement
4. Compare how `==` differs between primitives and objects
5. Compare two String objects by using the `compareTo` and `equals` methods

7.1 Create if and if/else constructs

7.1.1 Basic syntax of if and if-else ↗

The **if** statement is probably the most used decision construct in Java. It allows you to execute a single statement or a block of statements if a particular condition is true. If that condition is false, the statement (or the block of statements) is not executed. The following are two ways you can write an **if** statement: `if(booleanExpression) single_statement;`

Notice that there are no curly braces for the statement. The **if** statement ends with the semi-colon. If you have multiple statements that you want to execute instead of just one, you can put all of them within a block like this:

```
if ( booleanExpression ) {  
    zero or more statements;  
}
```

Observe that there is no semi-colon after the closing curly brace. It is not an error if present though.

If-else statement ↗

The **if-else** statement is similar to an **if** statement except that it also has an **else** part where you can write a statement that you want to execute if the **if** condition evaluates to false:

```
if( booleanExpression ) single statement; //semi-colon required here  
else single statement; //semi-colon required here, if-else statement ends with this  
    semi-colon
```

Again, if you have multiple statements to execute instead of just one, you can put them within a block:

```
if ( booleanExpression ) single statement; //semi-colon is required here  
else {  
    zero or more statements;  
} //no semi-colon required here, but not an error if it exists.
```

or, if both the **if** and the **else** parts have multiple statements:

```
if ( booleanExpression ) {  
    zero or more statements;  
} //must NOT have a semi-colon here, error if it exists  
else {  
    zero or more statements;  
} //no semi-colon required here, but not an error if it exists.
```

If `booleanExpression` evaluates to `true`, the statement (or the block of statements) associated with **if** will be executed and if the `booleanExpression` evaluates to `false`, the **else** part will be executed.

Remember that an empty statement (i.e. just a semicolon) is a valid statement and therefore, the following if and if-else constructs are valid:

```
boolean flag = true;

if(flag); //does nothing, but valid

if(flag); else; //does nothing, but valid

if(flag);else System.out.println(true); //does nothing because flag is true

if(flag) System.out.println(true); else; //prints true
```

By the way, you may see **if/if-else** statements being called **if-then/if-then-else** statement. This is not entirely correct. In some languages such Pascal, "then" is a part of the syntax but it is not in Java. However, "then" is not a keyword in Java and there is no "then" involved in the syntax of **if/if-else**.

7.1.2 Usage of if and if-else in the exam ↗

Let us now look at a few interesting ways if/if-else is used in the exam that might trip you up.

Bad syntax ↗

```
boolean flag = true;
if( flag )
else System.out.println("false"); //compilation error
```

In the above code, there is no statement or a block of statements for the if part. If you don't want to have any code to be executed if the condition is true but want to have code for the else part, you need to put an empty code block for the if part like this:

```
boolean flag = false;
if( flag ) {
}
else System.out.println("false");
```

or

```
boolean flag = false;
if( flag ) ; else System.out.println("false");
```

Instead of having an empty if block, it is better to negate the if condition and put the code in the if block. Like this:

```
boolean flag = false;
if( !flag ) {
    System.out.println("false");
}
```

Bad Indentation

Unlike some languages such as Python, indentation (and extra white spaces, for that matter) holds no special meaning in Java. Indentation is used solely to improve readability of the code. Consider the following two code snippets:

```
boolean flag = false;
if(flag)
    System.out.println("false");
else System.out.println("true");
{
    System.out.println("false");
}
```

and

```
boolean flag = false;
if(flag)
    System.out.println("false");
else
    System.out.println("true");

{
    System.out.println("false");
}
```

The above two code snippets are equivalent. However, since the second snippet is properly indented, it is easy to understand that the last code block is not really associated with the if/else statement. It is an independent block of code and will be executed irrespective of the value of `flag`. This code will, therefore, print `true` and `false`.

Here is another example of bad indentation:

```
boolean flag = false;
int i = 0;
if(flag)
    i = i +1;
    System.out.println("true");
else
    i = i + 2;
    System.out.println("false");
```

The above code is trying to confuse you into thinking that there are two statements in the if part and two statements in the else part. But, with proper indentation, it is clear what this code is really up to:

```
boolean flag = false;
int i = 0;

if(flag)  i = i +1;
```

```
System.out.println("true");

else i = i + 2;

System.out.println("false");
```

It turns out that the last three lines of code are independent statements. The else statement is completely out of place because it is not associated with the if statement at all and will, therefore, cause compilation error.

Missing else

As we saw earlier, the else part is not mandatory in an if statement. You can have just the if statement. But when coupled with bad indentation, an if statement may become hard to understand as shown in the following code:

```
boolean flag = true;
if(flag)
System.out.println("true");
{
System.out.println("false");
}
```

The above code prints `true` and `false` because there is no `else` part in this code. The second `println` statement is in an independent block and is not a part of the `if` statement.

Dangling else

"Dangling else" is a well known problem in programming languages that have `if` as well as `if-else` statements. This is illustrated in the following piece of code that has two `if` parts but only one `else` part:

```
int value = 3;
if(value == 0)
if(value == 1) System.out.println("b");
else System.out.println("c");
```

The question here is with which `if` should the `else` be associated? There are two equally reasonable answers to this question as shown below:

```
int value = 3;
if(value == 0) {
    if(value == 1) System.out.println("b");
}
else System.out.println("c");
```

and

```
int value = 3;
if(value == 0) {
    if(value == 1)
        System.out.println("b");
    else
        System.out.println("c");
}
```

In the first interpretation, **else** is associated with the first **if**, while in the second interpretation, **else** is associated with the second **if**. If we go by the first interpretation, the code will print **c**, and if we go by the second interpretation, the code will not print anything. For a compiler, both are legally valid ways to interpret the code, which makes the code ambiguous.

Since neither of the interpretations is more correct than the other, Java language designers broke the tie by deciding to go with the second interpretation, i.e., the **else** is to be associated with the nearest **if**. That is why the above code does not print anything as there is no **else** part associated with the first **if**. Based on the above discussion, you should now be able to tell the output of the following code:

```
int value = 3;
if(value == 0)
    if(value == 1)
        System.out.println("b");
    else
        System.out.println("c");
else
    System.out.println("d");
```

Just follow the rule that an **else** has to be associated with the nearest **if**. The following is how the above statement will be grouped:

```
int value = 3;

if(value == 0){
    if(value == 1)
        System.out.println("b");
    else
        System.out.println("c");
}
else System.out.println("d");
```

It will print **d**.

Using assignment statement in the if condition

Recall that every assignment statement itself is a valid expression with a value of its own. Its type and value are the same as the ones of the target variable. This fact can be used to write a very tricky if statement as shown below:

```

boolean flag = false;
if(flag = true){
    System.out.println("true");
}
else {
    System.out.println("false");
}

```

Observe that `flag` is not being compared with `true` here. It is being assigned the value `true`. Thus, the value of the expression `flag = true` is true and that is why the `if` part of the statement is executed instead of the `else` part. While this type of code is not appreciated in a professional environment, a similar construct is quite common:

```

String data = null;
if( (data = readData()) != null ) //assuming that readData() returns a String
{
    //do something
}

```

The above code is fine because the assignment operation is clearly separated from the comparison operation. The value of the expression `data = readData()` is being compared with `null`. Remember that the value of this expression is the same as the value that is being assigned to `data`. Thus, the `if` body will be entered only if `data` is assigned a non-null value.

Using pre and post increment operations in the if condition ↗

You will see conditions that use pre and post increment (and decrement) operators in the exam. Something like this:

```

int x = 0;
if(x++ > 0){ //line 2
    x--; //line 3
}

if (++x == 2){ //line 6
    x++; //line 7
}
System.out.println(x);

```

You can spot the trick easily if you have understood the difference between the value of an expression and the value of a variable used in that expression (I explained this in the "Using Operators" chapter). At line 2, `x` will be incremented to `1` but value of the expression `x++` is `0` and therefore, the condition will be evaluated to `false`. Thus, line 3 will not be executed. At line 6, `x` is incremented to `2` and the value of the expression `++x` is also `2`. Therefore, the condition will be evaluated to `true` and line 7 will be executed, thereby increasing the value of `x` to `3`. Thus, the above code will print `3`.

Remember that conditions are used in ternary expressions and loops as well, so, you need to watch out for this trick there also.

7.2 Create ternary constructs

7.2.1 The ternary conditional operator ?: ↗

The ternary operator (?:) is not mentioned in the JFCJA exam objectives explicitly. We haven't seen any one getting a question on it either. However, it is one of the commonly used operators in Java and from that perspective, you should know about it. You may skip through the following discussion if you are short on time.

The syntax of the ternary operator is as follows:

```
operand 1 ? operand 2 : operand 3;
```

Operand 1 must be an expression that returns a `boolean`. The `boolean` value of this expression is used to decide which one of the rest of the two other operands should be evaluated. In other words, which of the operands 2 and 3 will be evaluated is conditioned upon the return value of operand 1. If the boolean expression given in operand 1 returns true, the ternary operator evaluates and returns the value of operand 2 and if it is false, it evaluates and returns the value of operand 3. From this perspective, it is also a "**conditional operator**" (as opposed to &, |, !, and , which are really just "**logical**" operators).

Examples:

```
boolean sweet = false;
int calories = sweet ? 200 : 100; //assigns 100 to calories
boolean sweetflag = (calories == 100 ? false : true); //assigns false to sweetflag

boolean hardcoded = false;
//assuming getRateFromDB() method returns a double
double rate = hardcoded ? 10.0 : getRateFromDB(); //invokes method getRateFromDB

String value = sweetflag ? "Sweetened" : "Unsweetened";

Object obj = sweetflag ? "Sweetened" : new Object();
```

The ternary conditional operator is often thought of a short form for the **if/else statement** but it is similar to the if/else statement only up to conditional evaluation of its other two operands part. Their fundamental difference lies in the fact that the ternary expression is just an **expression** while an if/else statement is a **statement**. As discussed earlier, every expression must have a value and so, must the ternary expression. Since the value of a ternary expression is the value returned by the second or the third operand, the second and third operands of the ternary operator can comprise any expression. As the example given above shows, they can also include **non-void method invocations**. There is no such restriction with the if/else statement. The following example highlights this difference:

```
boolean flag = true;
if(flag) System.out.println("true");
else System.out.println("false");
```

The above if/else statement compiles fine and prints `true` but a similar code with ternary expression does not compile.

```
flag ? System.out.println("true") : System.out.println("false");
```

The reason for non-compilation of the above code is two fold. The first is that a ternary expression is not a valid statement on its own, which means, you cannot just have a free standing ternary expression. It can only be a part of a valid statement such as an assignment. For example,

```
int x = flag ? System.out.println("true")
: System.out.println("false");
```

This brings us to the second reason. The second and third operands in this example are invocations methods that return `void`. Obviously, you cannot assign void to an `int` variable. In fact, you cannot assign void to any kind of variable. Therefore, it fails to compile.

Type of a ternary conditional expression

Now that we have established that a ternary conditional expression must return a value, all that is left to discuss is the type of the value that it can return. It can return values of three types: **boolean**, **numeric**, and **reference**. (There are no other types left for that matter!)

If the second and third operands are expressions of type `boolean` (or `Boolean`), then the return type of the ternary expression is `boolean`. For example:

```
int a = 1, b = 2;
boolean flag = a == b? true : false; //ternary expression that returns a boolean
```

If the second and third operands are expressions of numeric types (or their wrapper classes), then the return type of the ternary expression is the wider of the two numeric types. For example, `double d = a == b? 5 : 10.0;`. Observe here that the second operand is of type `int` while the third is of type `double`. Since `double` is wider than `int`, the type of this ternary expression is `double`. You cannot, therefore, do `int d = a == b? 5 : 10.0;` because you cannot assign a double value to an `int` variable without a cast.

If the second and third operands are neither of the above, then the return type of the ternary expression is reference. For example,

`Object str = a == b? "good bye" : "hello";` Here operands 2 and 3 are neither numeric nor boolean. Therefore, the return type of this ternary expression is a reference type.

The first two types are straight forward but the third type begs a little more discussion. Consider the following line of code:

```
Object obj = a == b? 5 : "hello";.
```

Here, the second operand is of a numeric type while the third is of type `String`. Since this falls in the third category, the return type of the expression `a == b? 5 : "hello";` must be a reference. The question before the compiler now is what should be the type of the reference that is returned by the expression. One operand is an `int`, which can be boxed into an `Integer` object and another one is a `String` object. If the `boolean` condition evaluates to `true`, the expression will return an `Integer` object and if the condition evaluates to `false`, the expression will return a `String` object. Remember that the compiler cannot execute any code and so, it cannot determine what the expression will return at run time. Thus, it needs to pick a type that is compatible with both kind of values. The compiler solves this problem by deciding to pick the most specific common superclass of the two types as the type of the expression. In this case, that class is `java.lang.Object`. By selecting the most specific common super class, the compiler ensures that irrespective of the result of the condition, the value returned by this expression will always be of the same type, i.e., `java.lang.Object`, in this case. Here is an example, where the most specific common super class is not `Object` :

```
Double d = 10.0;
Byte by = 1;
Number n = a == b? d : by;
```

Here, the most specific common superclass of `Double` and `Byte` is `Number`. (Recall that all wrapper classes for integral types extend from `java.lang.Number`, which in turn extends from `java.lang.Object`). You can therefore, assign the value of the expression to a variable of type `Number`.

You should now be able to tell the result of the following two lines of code:

```
int value = a == b? 10 : "hello"; //1
```

```
System.out.println(a == b? 10 : "hello"); //2
```

As discussed above, the type of the expression `a == b? 10 : "hello";` is `Object`. Can you assign an `Object` to an `int` variable? No, right? Therefore, the first line will not compile. Can you pass an `Object` to the `println` method? Of course, you can. Therefore, the second line will compile and run fine.

The short circuiting property of ?:

Depending on whether the value of operand one is true or false, either operand 2 or operand 3 is evaluated. In other words, if the condition is `true`, operand 3 is not evaluated and if the condition is `false`, operand 2 is not evaluated. In this respect, the ternary conditional operator is similar to the other two short circuiting operators, i.e., `&&` and `||`. However, there is an important difference between the two. While with `&&` and `||`, evaluation of both the operands is possible in certain situations, it is never the case with `?:` operator. `?:` evaluates exactly one of the two operands in all situations.

The short circuiting nature of `?:` provides a good opportunity for trick questions in the exam. For example, what will the following code print?

```
int x = 0;
int y = 1;
System.out.println(x>y? ++x : y++);
System.out.println(x+" "+y);
```

This code prints:

```
1
0 2
```

Since the value of `x` is not greater than `y`, `x > y` evaluates to `false` and therefore, the ternary expression returns the value of the third operand, which is `y++`. Since `y++` uses post-increment operator, the return value of `y++` will be the current value of `y`, which is `1`. `y` will then be incremented to `2`. Observe that evaluation of the second operand is short circuited because the first operand evaluates to false. Therefore, `++x` is never executed. Thus, the second print statement prints `0 2`.

7.3 Use a switch statement

7.3.1 Creating a switch statement

A switch statement allows you to use the value of a variable to select which code block (or blocks) out of multiple code blocks should be executed. Here is an example:

```
public class TestClass {
    public static void main(String[] args){
        int i = args.length;

        switch(i) { //switch block starts here

            case 0 : System.out.println("No argument");
                       break;
            case 1 : System.out.println("Only one argument");
                       break;
            case 2 : System.out.println("Two arguments");
                       break;
            default : System.out.println("Too many arguments!");
                       break;

        } //switch block ends here

        System.out.println("All done.");
    }
}
```

There are four blocks of code in the above switch statement. Each block of code is associated with a **case label**. Depending of the value of the variable `i`, the control will enter the code block associated with that particular case label and keep on executing statements until it finds a `break`

statement. For example, if the value of `i` is 0, the control will enter the first code block. It will print `No argument` and then encounter the `break` statement. The break statement causes the control to exit the switch statement and move on to the next statement after the switch block, which prints `"All done."`.

If the value of `i` doesn't match with any of the case labels, the control looks for a block labelled `default` and enters that block. If there is no default block either, the control does not enter the switch block at all. Since this "switching" is done based on the expression specified in the switch statement (which is just `i` in this example), this expression is aptly called the **"switch expression"**.

Operationally, this seems quite similar to a cascaded `if/else` statement. Indeed, the above code can very well be written using an `if/else` statement as follows:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = args.length;  
  
        if(i == 0) {  
            System.out.println("No argument");  
        } else if (i == 1) {  
            System.out.println("Only one argument");  
        } else if (i == 2) {  
            System.out.println("Two arguments");  
        } else {  
            System.out.println("Too many arguments!");  
        }  
        System.out.println("All done.");  
    }  
}
```

Well, why do you need a switch statement then, you may ask. To begin with, as you can see, an `if/else` statement is a lot more verbose than a switch statement. The switch version is also a little easier to comprehend than the `if/else` version. But underneath this syntactic ease lies a complicated beast. This is evident when we look at the moving parts involved in a switch statement more closely.

The switch expression

A switch expression must evaluate to one of the following three kinds:

1. a limited set of integral types (`byte`, `char`, `short`, `int`), and their wrapper classes. Observe that even though `long` is an integral type, it cannot be the type of a switch variable. `boolean`, `float`, and `double` are not integral types anyway and therefore, cannot be the type of a switch variable either.
2. `enum` type
3. `java.lang.String` - Generally, reference types are not allowed as switch expressions but an exception for `java.lang.String` was made in Java 7. So now, you can use a String expression as a switch expression.

Compare this to an **if/else** statement where branching is done based on the value of a **boolean expression**. This limits an if/else statement to at most two branches. Of course, as we saw earlier, you can cascade multiple if/else statements to achieve multiple branches.

The case labels

Case labels must consist of **compile time constants** that are assignable to the type of the switch expression. For example, if your switch expression is of type **byte**, you cannot have a case label with a value that is larger than a byte. Similarly, if your switch expression is of type **String**, the case labels must be constant string values as illustrated in the following code:

```
public static void main(String[] args){  
    switch(args[0]){  
        case "1" : System.out.println("one"); //valid because "1" is a compile time constant  
        case "1"+"2" : System.out.println("one"); //valid because "1"+"2" is a compile time constant  
        case args[1] : System.out.println("same args"); //will not compile because args[1] is not a compile time constant  
        case "abc".toUpperCase() : System.out.println("ABC"); //will not compile because "abc".toUpperCase() is not a compile time constant  
    }  
}
```

Observe that **"1"** and **"1"+"2"** are compile time constants because the value of these expressions is known at compile time, while **args[1]** and **"abc".toUpperCase()** are not compile time constants because their values can only be determined at run time when the code is executed.

The interesting thing about case labels is that they are **optional**. In other words, a switch statement doesn't necessarily have to have a case label. The following is, therefore, a superfluous yet valid switch statement.

```
switch(i){  
    default : System.out.println("This will always be printed");  
}
```

Another point worth repeating here is that although it is very common to use a single variable as the switch expression but you can use any expression inside the switch. And when you talk of an expression, all the baggage of numeric promotion, casting, and operator precedence that we saw previously, comes along with it. You need to consider all that while checking the validity of case labels. For example, while the following code fails to compile:

```
byte b = 10;  
switch(b){ //type of the switch expression here is byte  
    case 1000 : //1000 is too large to fit into a byte  
        System.out.println("hello!");  
}
```

the following code compiles fine:

```
byte b = 10;  
switch(b+1){ //type of the switch expression here is now int due to numeric promotion
```

```
case 1000 : //1000 can fit into an int
    System.out.println("hello!");
}
```

The default block

There can be at most one default block in a switch statement. The purpose of the default block is to specify a block of code that needs to be executed if the value of the switch expression does not match with any of the case labels. Just like the case labels, this block is also **optional**.

The order of case and default blocks

Java does not impose any particular order for the case statements and the default block. Thus, although it is customary to have the default block at the end of a switch block, you can have it even at the beginning. Similarly, Java does not care about the ordering of the case labels.

However, "does not care" does not mean "not important"! Ordering of case and default blocks becomes very important in combination with the use of the **break** statement as we will see next.

The break statement

I mentioned in the beginning that the case labels determine the entry point into a switch statement and the break statement determines the exit. That is true but the interesting thing is that even the break statement is **optional**. A case block does not necessarily have to end with a break. Let me modify the program that I showed you in the beginning:

```
public class TestClass {
    public static void main(String[] args){

        switch(args.length) { //switch block starts here

            case 0 : System.out.println("No argument");
                       //break;
            case 1 : System.out.println("Only one argument");
                       //break;
            case 2 : System.out.println("Two arguments");
                       //break;
            default : System.out.println("Too many arguments!");
                       //break;

        } //switch block ends here

        System.out.println("All done.");
    }
}
```

I have commented out all the break statements. Now, if you run this program **without any argument**, you will see the following output:

```
No argument
Only one argument
Two arguments
Too many arguments!
All done.
```

The control entered at the block labelled **case 0** (because `args.length` is 0), and executed all the statements of the switch block...even the statements associated with other case blocks that did not match the value of `args.length`. This is called "**fall through**" behavior of a switch statement. In absence of a break statement, the control falls through to the next case block and the next case block, and so on until it reaches the end of the switch statement. This feature is used when you want to have one code block to execute for multiple values of the switch expression. Here is an example:

```
char ch = 0;
int noOfVowels = 0;
while( (ch = readCharFromStream()) > 0) {
    switch(ch) {

        case 'a' :
        case 'e' :
        case 'i' :
        case 'o' :
        case 'u' :
            noOfVowels++;

        default :    logCharToWhatever(ch);
    }
}
System.out.println("Number of vowels received "+noOfVowels);
```

The above code logs each character it receives but increments `noOfVowels` only if the character received is a lower case vowel.

You will see questions in the exam on this behavior of the switch statement. So pay close attention to where in the switch block does the control enter and where it exits. Always check for missing break statements that cause the control to fall through to the next case block.

The following is a typical code snippet you may get in an exam. Try running it with different arguments and observe the output in each case:

```
public class TestClass {
    public static void main(String[] args){
        int i = 0;
        switch(args[0]) {
```

```
        default : i = i + 3;
        case "2" : i = i + 2;
        case "0" : break;
        case "1" : i = i + 1;

    }

    System.out.println("i is "+i);
}

}
```

The fall through behavior of a switch statement does not really get used a lot in Java but it has been used in interesting ways to optimize code in other languages. If you have time, you might want to google "duff's device" to see one such usage. This is, of course, not required for the exam :)

7.4 Exercise

1. Write a method that accepts a number as input and prints whether the number is odd or even using an if/else statement as well as a ternary expression.
2. Accept a number between 0 to 5 as input and print the sum of numbers from 1 to the input number using code that exploits the "fall through" behavior of a switch statement.
3. Accept a number as input and generate output as follows using a cascaded and/or nested if/else statement - if the number is even print "even", if it is divisible by 3, print "three", if it is divisible by 5, print "five" and if it is not divisible by 2, 3, or 5, print "unknown". If the number is divisible by 2 as well as by 3, print "23", and if the number is divisible by 2, 3, and 5, print "235".
4. Indent the following code manually such that it reflects correct if - else associations. Use a plain text editor such as Notepad. Copy the code into a Java editor such Netbeans or Eclipse and format it using the editor's auto code formatting function. Compare your formatting with the editor's.

```
int a = 0, b = 0, c = 0, d = 0;
boolean flag = false;
if (a == b)
if (c == 10)
{
if (d > a)
{
} else {
```

```
}

if (flag)
System.out.println("");
else
System.out.println("");
}
else if (flag == false)
System.out.println("");
else if (a + b < d) {
System.out.println("");
}
else
System.out.println("");
else d = b;
```


Exam Objectives

1. Using Looping Statements
2. Describe looping statements
3. Use a for loop including an enhanced for loop
4. Use a while loop
5. Use a do- while loop
6. Compare and contrast the for, while, and do-while loops
7. Develop code that uses break and continue statements

8.1 What is a loop

8.1.1 What is a loop ↗

A loop causes a set of instructions to execute repeatedly until a certain condition is reached. It's like when the kids keep asking, "are we there yet?", when you are on a long drive in a car. They ask this question in a "loop", until you are really there :) Or until you "break" their loop by putting on a DVD.

Well, in Java, loops work similarly. They let you execute a group of statements multiple times or even forever depending on the **loop condition** or until you **break** out of them. Every repetition of execution of the statements is called an **iteration**. So, for example, if a group of statements is executed 10 times, we can say that the loop ran for 10 iterations.

Loops are a fundamental building block of **structured programming** and are used extensively for tasks ranging from the simple such as counting the sum of a given set of numbers to the complicated such as showing a dialog box to the user until they select a valid file.

Java has three different ways in which you can create a loop - using a while statement, using a do/while statement, and using a for statement. Let us take a look at each of them one by one.

8.2 Create and use while loops

8.2.1 The while loop ↗

As the name of this loop suggests, a while loop executes a group of statements **while** a condition is true. In other words, it checks a condition and if that condition is true, it executes the given group of statements. After execution of the statements, i.e., after finishing that iteration, it loops back to check the condition again. If the condition is false, the group of statements is not executed and the control goes to the next statement after the while block. The syntax of a while loop is as follows:

```
while (boolean_expression) {
    statement(s);
}
```

and here is an example of its usage:

```
public class TestClass {
    public static void main(String[] args){
        int i = 4;
        while(i>0){
            i--;
            System.out.println("i is "+i);
        }
        System.out.println("Value of i after the loop is "+i);
    }
}
```

It produces the following output:

```
i is 3  
i is 2  
i is 1  
i is 0  
Value of i after the loop is 0
```

Observe that the condition is checked **before** the group of statements is executed and that once the condition `i > 0` returns `false`, the control goes to the next statement after the while block.

As you have seen in the past with if/else blocks, if you have only a single statement that you need to execute in a loop, you may get rid of the curly braces if you want. In this case, the syntax becomes:

```
while(boolean_expression) statement;
```

The example program given above can also be written to use a single statement like this:

```
public class TestClass {  
    public static void main(String[] args){  
        int i = 4;  
        while(i-->0) System.out.println("i is "+i); //no curly braces  
        System.out.println("Value of i after the loop is "+i);  
    }  
}
```

The above code looks the same as the previous one but produces a slightly different output:

```
i is 3  
i is 2  
i is 1  
i is 0  
Value of i after the loop is -1
```

The difference is that I have used the post-decrement operator to decrement the value of `i` within the condition expression itself. Remember that when you use the **post**-decrement (or the **post**-increment) operator, the variable is decremented (or incremented) **after** its existing value is used to evaluate the expression. In this case, when `i` is `0`, the condition expression evaluates to false and the while block is not executed, but the variable `i` is nevertheless decremented to `-1`. Therefore, the value of `i` after the loop is `-1`.

8.2.2 Using a while loop

Control condition and control variable

When using a while loop you should be careful about the expression that you use as the while condition. Most of the time, a while condition comprises a single variable (which is also called the "**control variable**", because it controls whether the loop will be entered or not) that you compare against a value. For example, `i > 4` or `name != null` or `bytesRead != -1` and so on. However, it

can get arbitrarily large and complex. For example, `account == null || (account != null && account.accountId == null) || (account != null && account.balance == 0)`. The expression must, however, return a `boolean`. Unlike some languages such as C, Java does not allow you to use an integer value as the while condition. Thus, `while(1) System.out.println("loop forever");` will not compile.

while body

Ideally, the code in the while body should modify the control variable in such a way that the control condition will evaluate to false when it is time to end the loop. For example, if you are processing an array of integers, your control variable could be set to the index of the element that you are processing, and each iteration should increment that variable. For example,

```
int[] myArrayOfInts = //code to fetch the data
int i = 0; //control variable
while(i<myArrayOfInts.length){
//do something with myArrayOfInts[i]
i++; //increment i so that the control condition will evaluate to false after the last
      element is processed
}
```

There are situations where you do not want a loop to end at all. For example, a program that listens on a socket for connections from clients may use a while loop as follows:

```
Socket clientSocket = null;
while( (clientSocket = serverSocket.accept() ) != null ){
    //code to hand over the clientSocket to another thread and go back to the while
    //condition to keep listening for connection requests from clients
}
```

The above is a commonly used while loop idiom.

A never ending while loop can also be as simple as this:

```
while(true){
    System.out.println("keep printing this line forever!");
}
```

On the other extreme, keep an eye for a condition that never lets the control enter the while body:

```
int i = 0;
while(i>0){ // the condition is false to begin with
    System.out.println("hello"); //this will never be printed
    i++;
}
```

It is possible to exit out of a while loop without checking the while condition. This is done using the `break` statement. I will discuss it later.

8.3 Create and use do/while loops

8.3.1 The do-while loop

A do-while loop is similar to the while loop. The only difference between the two is that in a do-while loop the loop condition is checked after executing the loop body. Its syntax is as follows:

```
do {
    statement(s);
}while(boolean_expression);
```

Observe that a do-while statement starts with a "do" but there is no "do" anywhere in a while statement. Another important point to understand here is that since the loop condition is checked after the loop body is executed, the loop body will always be executed at least once.

As with a while statement, it is ok to remove the curly braces for the loop body if there is only one statement in the body. Thus, the following two code snippets are actually the same:

```
int i = 4;
do {
    System.out.println("i is "+i);
} while(i-->0);
System.out.println("Loop finished. i is "+i);

int i = 4;
do
    System.out.println("i is "+i);
while(i-->0);
System.out.println("Loop finished. i is "+i);
```

Deciding whether to use a while loop or a do while loop is easy. If you know that the loop condition may evaluate to false at the beginning itself, i.e., if the loop body may not execute even once, you must use a while loop because it lets you check the condition first and then execute the body. For example, consider the following code:

```
Iterator<Account> acctIterator = accounts.iterator();
while(acctIterator.hasNext()){ //no need to enter the loop body if accounts collection
    is empty
    Account acct = acctIterator.next();
    //do something with acct
}
```

Don't worry about the usage of `Iterator` or `<Account>` in the above code. The point to understand here is that you want to process each account object in the accounts collection and if there is no account object, you don't want to enter the loop body at all. If you use a do-while loop, the code will look like this:

```
Iterator<Account> acctIterator = accounts.iterator();
do {
```

```

Account acct = acctIterator.next(); //will throw an exception
//do something with acct
}while(acctIterator.hasNext());
}

```

The above code will work fine in most cases but will throw an exception if the account collection is empty. To achieve the same result with a do-while loop, you would have to write an additional check for an empty collection at the beginning. Something like this:

```

Iterator<Account> acctIterator = accounts.iterator();
if(acctIterator.hasNext()) { //no need to enter the if body if accounts collection is
    empty
    do{
        Account acct = acctIterator.next();
        //do something with acct
    }while(acctIterator.hasNext())
}

```

I think the choice is quite clear. A while loop is a natural fit in this case.

Generally, a while loop is considered more readable than a do-while loop and is also used a lot more in practice. I have not needed to use a do-while loop in a long while myself :)

8.4 Create and use for loops

8.4.1 Going from a while loop to a for loop ↗

The **for loop** is the big daddy of loops. It is the most flexible, the most complicated, and the most used of all loop statements. It has so many different flavors that many programmers do not get to use some of its variations despite years of programming in Java. But don't be scared. It still follows the basic idea of a loop, which is to let you execute a group of statements multiple times.

To ease you into it, I will morph the code for a while loop into a for loop. Here is the code that uses a while loop:

```

public class TestClass {
    public static void main(String[] args){
        int i = 4;
        while(i>0){
            System.out.println("i is "+i);
            i--;
        }
        System.out.println("Value of i after the loop is "+i);
    }
}

```

The output of the above code is:

```
i is 4
i is 3
```

```
i is 2
i is 1
Value of i after the loop is 0
```

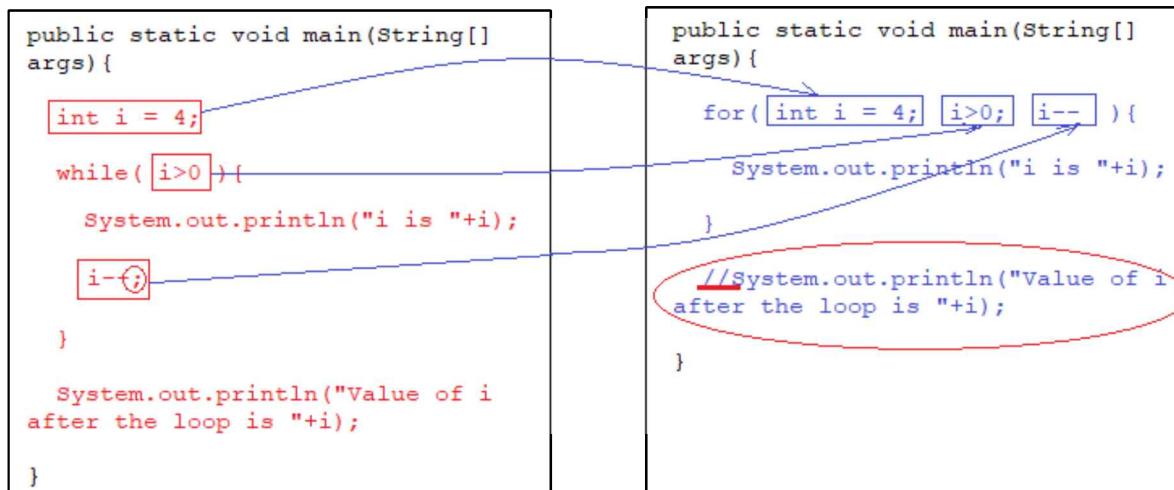
Here is the same code with a for loop:

```
public class TestClass {
    public static void main(String[] args){
        for(int i = 4; i>0; i--){
            System.out.println("i is "+i);
        }
        //System.out.println("Value of i after the loop is "+i);
    }
}
```

Its output is:

```
i is 4
i is 3
i is 2
i is 1
```

As you can see, it produces the same output except the last line. The following image shows how the elements of a while loop of the first code were mapped to the for loop of the second code.



Converting a while loop to a for loop

Here are a few things that you should observe in the transformation shown above.

1. The while statement contains just the comparison but the for statement contains initialization, comparison, and updation. Technically, the declaration of the loop variable i and its updation through `i--` is not really a part of the while statement. But the for loop has a provision for both.

2. The implication of declaring the loop variable `i` in a for statement is that it is scoped only to the for block and is not visible outside the for block, which is why I have commented out the last print statement in the second code.
3. There is no semi-colon after `i--` in the for statement.

This example illustrates that fundamentally there is not much difference between the two loop constructs. Both the loops have the same three basic components - declaration and/or initialization of a loop variable, a boolean expression that determines whether to continue next iteration of the loop, and a statement that updates the loop variable. But you can see that the for loop has a compact syntax and an in-built mechanism to control the initialization and updation of the loop variable besides the comparison as compared to the while loop.

8.4.2 Syntax of a for loop

A for loop has the following syntax:

```
for( optional initialization section ; optional condition section ; optional updation
    section ) {
    statement(s);
}
```

Predictably, if you have only zero or one statement in the for block, you can get rid of the curly braces and end the statement with a semicolon like this:

```
for( optional initialization section ; optional condition section ; optional updation
    section ) ;
//no body at all
```

or

```
for( optional initialization section ; optional condition section ; optional updation
    section )
    single_statement;
```

All three sections of a for statement are optional but the **two semi-colons** that separate the three sections are not. You are allowed to omit one, two, or all of the three sections. Thus, `for(; ;)`; is actually a valid for statement but `for()`; and `for();`; are not.

Order of execution

Various pieces of a for loop are executed in a specific order, which is as follows:

1. The **initialization section** is the first that is executed. It is executed **exactly once** irrespective of how many times the for loop iterates. If this section is **empty**, it is **ignored**.
2. Next, the **condition section** is executed. If the result of the expression in this section is **true**, the next iteration, i.e., execution of the statements in the for block is kicked off. If the result is **false**, the loop is terminated immediately, i.e., neither the for block and nor the updation section is executed. If this section is **empty**, it is assumed to be **true**.

3. The statements in the **for block** are executed.
4. The **updation section** is executed. If this section is **empty**, it is **ignored**.
5. The control goes back to the **condition section**.

Let us see how the above steps are performed in the context of the following for loop.

```
for(int i = 3; i>0; i--){
    System.out.println("i is "+i);
}
```

1. `int i = 3;` is executed. So, `i` is now **3**.
2. Expression `i > 0` is evaluated and since **3** indeed greater than **0**, it evaluates to **true**. Therefore, the next iteration will now commence.
3. The print statement is executed, which prints **3**.
4. `i--` is executed thereby reducing `i` to **2**.
5. Control goes back to the condition section. `i > 0` evaluates to **true** because **2** is greater than **0**. Therefore, the next iteration will now commence.
6. The print statement is executed, which prints **2**.
7. `i--` is executed thereby reducing `i` to **1**.
8. Control goes back to the condition section. `i > 0` evaluates to **true** because **1** is greater than **0**. Therefore, the next iteration will now commence.
9. The print statement is executed, which prints **1**.
10. `i--` is executed thereby reducing `i` to **0**.
11. Control goes back to the condition section. `i > 0` evaluates to **false** because **0** is not greater than **0**. Therefore, the loop will be terminated. The statements in the for block and the updation section will not be executed and the control goes to the next statement after the for block (which is the end of the method in this case).

So far, the for statement looks quite straight forward. You will see the complications when we turn our attention to the intricacies of the three sections of the for statement next.

8.4.3 Parts of a for loop

The initialization section

The initialization section allows you to specify code that will be executed at the beginning of the for loop. This code must belong to a category of statements that are called "**expression statements**". Expression statements are expressions that can be used as statements on their own. These are: **assignment**, **pre/post increment/decrement expression**, **a method call**, and **a class instance creation expression** (e.g. using the new operator). Besides expression statements, this section also allows you to declare local variables.

Here are a few examples of valid expression statements in a for loop:

```
int i = 0;
for(i = 5; i<10; i++); //assignment

Object obj;
for(obj = "hello"; i<10; i++); //assignment

int i = 0;
int k = 0;
Object obj = "";
for(i = 0, k = 7, obj = "hello"; i<10; i++); //multiple assignments

int k = 0;
for(++k; i<10; i++); //pre-increment

for(new ArrayList(); i<10; i++); //instance creation

int i = 0;
for(System.out.println("starting the loop now"); i<10; i++); //method call

for(k++, i--, new String(); i<10; i++); //multiple expressions
```

Observe that there doesn't need to be any relationship between the variable used in the initialization section and the variable used in other sections of a for loop.

The following are a few examples of valid local variable declarations:

```
for(int i = 5; i<10; i++); //single variable declaration

for(int i = 5, k = 7; i<10; i++); //multiple variable declaration
```

You can only declare variables of one type. So the following is **invalid** :

```
for(int i = 5, String str = ""; i<10; i++); //invalid
```

Redeclaring a variable is also invalid:

```
int i = 0;
for(int i = 5; i<10; i++); //invalid because i is already declared
```

Another important point to note here is that a variable declared in the initialization section is visible only in the for loop. Thus, the following **will not compile** because `i` is not visible outside the loop.

```
for(int i = 5; i<10; i++){
    System.out.println("i is "+i);
}
System.out.println("Final value of i is "+i); //this line will not compile
```

The condition section

This one is simple. No, really :) You can only have an expression that returns a `boolean` or `Boolean` in this section. If there is no expression in this section, the condition is assumed to be `true`.

The updation section

The rules for the updation section are the same as the initialization section except that you cannot have any declarations here. This section allows only "expression statements", which I have already discussed above. Generally, this section is used to update the loop variable (`i++` or `k = k + 2` and so on) but as you saw in examples above, this is not the only way to use it. There doesn't need to be any relationship between the code specified here and in other sections. The following are a few valid examples:

```
int i = 0;
for(;i<10; i++); //post-increment

for(;i<10; i = i + 2); //increment by two

for(;i<10; i = someRef.getValue()) //assignment

for(Object obj = new Object(); obj != null; ) { //empty updation section
    System.out.println(obj);
    obj = null;
}

for(int i = 0; i<10; callSomeMethod() ); //method call
```

Observe that there is no semi-colon after the expression statement. It is terminated by the closing parenthesis of the for statement.

An infinite for loop

Based on the above information, it should now be very easy to analyze the following loop:

```
for( ; ; ) ;
```

The initialization section is empty. The condition section is empty, which means it will be assumed to be `true`. The updation section is empty. The loop code is also empty. There is nothing that

makes the condition section to evaluate to `false` and therefore, there is nothing to stop this loop from iterating forever.

Now, as an exercise, try analyzing the following loop and determine its output:

```
boolean b = false;
for(int i=0 ; b = !b ; ) {
    System.out.println(i++);
}
```

8.5 Create and use for each loops

8.5.1 The enhanced for loop

Motive behind the enhanced for loop

In professional Java programming, looping through a collection of objects is a very common requirement. Here is how one might do it:

```
String[] values = { "a", "bb", "ccc" };
for(int i = 0; i<values.length; i++){
    System.out.println(values[i]); //do something with each value
}
```

The above code iterates through an array but the same pattern can be used to iterate through any other collection such as a List. You can code this loop using a while or do-while construct as well.

Java designers thought that this is too much code to write for performing such a simple task. The creation of an iteration variable `i` and the code in the condition section to check for collection boundary is necessary only because of the way the for (or other) loops work. They have no purpose from a business logic point of view. All you want is a way to do something with each object of a collection. You don't really care about the index at which that element is inside that collection.

Furthermore, some collections such as a set have no notion of index. Although you are not expected to know about sets for the JFCJA exam, you can't really avoid knowing a bit about it if you want to understand the "enhanced for loop". So, briefly, `java.util.HashSet` is a class in standard Java library that allows you to collect a bunch of unique objects but without any specific order. Unlike an array, where all the elements are ordered (note that I am saying ordered, not sorted) and each element is accessible through its index, there is no index in a `HashSet`. You can't tell which element is the first element and which one is the last because there is no order. How will you iterate through every element of a `HashSet` then?

Here is the code that iterates through the elements of a `HashSet` just to give you an idea of how you may iterate though a collection of objects that does not support index based retrieval:

```

import java.util.HashSet;
import java.util.Iterator;
public class TestClass{
    public static void main(String[] args){
        Set s = new HashSet();
        s.add("a");
        s.add("bb");
        s.add("ccc");

        Iterator it = s.iterator();

        while( it.hasNext() ){
            Object value = it.next();
            System.out.println(value); //do something with each value
        }
    }
}

```

Don't worry if you feel overwhelmed by the above code. The only purpose of the above code is to show you how cumbersome it was to iterate through the objects of a collection. The above code creates an `Iterator` object, which has no relation to the business logic, to iterate through the elements. Such code that is not required by the business logic but is required only because of the way a programming language works is called "boilerplate code". Such code can be easily eliminated if a programming language provides higher level constructs that internalize the logic of this code.

In Java 5, Java designers simplified the iteration process by doing exactly that. They introduced a new construct that internalizes the creation of iteration variable and the boundary check and called it the "enhanced for loop" or the "for-each loop".

The enhanced for loop

Let me first show how the two code snippets given above can be simplified and then I will get into the details:

```

String[] values = { "a", "bb", "ccc" };
for(String value : values){
    System.out.println(value); //do something with each value
}

```

and

```

Set s = new HashSet();
s.add("a");
s.add("bb");
s.add("ccc");

for(Object value : s){
    System.out.println(value); //do something with each value
}

```

```
}
```

Isn't that simple? It reads easy too - "for **each** **value** in **values** do ..." and "for **each** **value** in **s** do ...".

8.5.2 Syntax of the enhanced for loop ↗

The syntax of the enhanced for loop is as follows:

```
for( Type variableName : array_or_Iterable ){
    statement(s);
}
```

or if there is only one statement in the for block:

```
for( Type variableName : array_or_Iterable ) statement;
```

Type is the type of the elements that the array or the collection contains, **variableName** is the local variable that you can use inside the block to work with an element of the array or the collection, and **array_or_iterable** is the array or an object that implements `java.lang.Iterable` interface.

I know that I have been using the word "collection" up till now and suddenly I have switched to **Iterable**. The reason for the switch is that **Iterable** is a super interface of `java.util.Collection` and although for-each loop is used mostly to iterate over collections, technically, it can iterate through an object of any class that implements **Iterable**. In other words, any class that wants to allow a user to iterate through the elements that it contains using the enhanced for loop must implement the **Iterable** interface.

The **Iterable** interface was introduced in Java 1.5 specifically to denote that an object can be used as a target of the for-each loop. It has only one method named **iterate**, which returns a `java.util.Iterator` object. Since `java.util.Collection` extends **Iterable**, all standard collection classes such as `HashSet`, and `ArrayList` can be iterated over using the enhanced for loop.

For the purpose of the exam, you don't need to worry about the **Iterable** or the **Iterator** interface but it is a good idea to develop a mental picture of the relationship between the **Iterable** and the **Iterator** interfaces. Always remember that for-each can only be used to "iterate" over an object that is "**Iterable**".

Note that `java.util.Iterator` itself is not a collection of elements and does not implement **Iterable**. Therefore, an Iterator object cannot be a target of a for-each loop. Thus, the following code will not compile:

```
Iterator it = myList.iterator();
for(Object s : it){ //this line will not compile
}
```

For the purpose of the JFCJA exam, you only need to know that you can use the for-each loop to iterate over any collection such as a **List** and an **ArrayList**.

8.5.3 Enhanced for loop in practice

Using enhanced for loop with typified collections

In the code that I showed earlier, I used Object as the type of the variable inside the loop. Since I was only printing the object out I didn't need to cast it to any other type. But if you want to invoke any type specific method on the variable, you would have to cast it explicitly like this:

```
List names = //get names from somewhere
for(Object obj : names){
    String name = (String) obj;
    System.out.println(name.toUpperCase());
}
```

The true power of the enhanced for loop is realized when you use generics (which were also introduced in Java 5, by the way) to generify the collection that you are trying to iterate through. Here is the same code with generics:

```
List< String> names = //get names from somewhere
for(String name : names){
    System.out.println(name.toUpperCase());
}
```

Although the topic of generics is not on the JFCJA exam, you will see questions on foreach that use generics. You don't need to know much about generics to answer the questions, but you should be aware of the basic syntax. I will discuss it more while talking about ArrayList later.

Counting the number of iterations

In a regular for loop, the iteration variable (usually named i) tells you which iteration is currently going on. There is no such variable in a foreach loop. If you do want to know about the iteration number, you will need to create and manage another variable for it. For example:

```
List<String> names = //get names from somewhere
int i = 0;
for(String name : names){
    i++;
    System.out.println(i+" : "+name.toUpperCase());
}
System.out.println("Total number of names is "+i);
```

8.6 Use break and continue

8.6.1 Terminating a loop using break

A loop doesn't always have to run its full course. For example, if you are iterating through a list of names to find a particular name, there is no need to go through the rest of the iterations after you have found that name. The break statement does exactly that. Here is the code:

```
String[] names = { "ally", "bob", "charlie", "david" };
for(int i=0; i<names.length; i++){
    System.out.println(names[i]);
    if("bob".equals(names[i])){
        System.out.println("Found bob at "+i);
        break;
    }
}
```

The output of the above code is:

```
ally
bob
Found bob at 1
```

Observe that **charlie** and **david** were not printed because the loop was broken after reaching **bob**.

It doesn't matter which kind of loop (i.e. while, do-while, for, or enhanced for) it is. If the code in the loop encounters a break, the loop will be broken immediately. The condition section and the updation section (in case of a for loop) will not be executed anymore and the control will move to the next statement after the loop block.

8.6.2 Terminating an iteration of a loop using continue ↗

The **continue** statement is a little less draconian than the **break** statement. The purpose of a **continue** statement is to just skip the rest of the statements in the loop while executing that particular iteration. In other words, when a loop encounters the continue statement, the remaining statements within the loop block are skipped. The rest of the steps of executing a loop are performed as usual, i.e., the control moves on to updation section (if it is a for loop) and then it checks the loop condition to decide whether to execute the next iteration or not.

This is useful when you don't want to execute the rest of the statements of a loop after encountering a particular condition. Here is an example:

```
String[] names = { "ally", "bob", "charlie", "david" };

for(String name : names){ //using a for-each loop this time

    if("bob".equals(name)){
        System.out.println("Ignoring bob!");
        continue;
    }
    System.out.println("Hi "+name+"!");
}
```

This code produces the following output:

```
Hi ally!
Ignoring bob!
```

```
Hi charlie!
Hi david!
```

Observe that the print statement saying **Hi** was skipped only when the name was **bob**.

Just like the **break** statement, the **continue** statement is applicable to all kind of loops.

Both of these statements are always used in conjunction with a conditional statement such as an if/if-else. If a continue or a break executes unconditionally then there would be no point in writing the code below a continue or a break. For example, the following code will fail to compile:

```
String[] names = { "ally", "bob", "charlie", "david" };

for(int i=0; i<names.length; i++){
    continue; //or break;
    System.out.println("Hi "+names[i]+"!");
}
```

The compiler will complain that the print statement is unreachable.

8.7 Nested loops

8.7.1 Nested loop ↗

A nested loop is a loop that exists inside the body of another loop. This is not a big deal if you realize that the body of a loop is just another set of statements. Among these set of statements, there may also be a loop statement. For example, while looping through a list of Strings, you can also loop through the characters of each individual String as shown in the following code:

```
String[] names = { "ally", "bob", "charlie", "david" };

for(String name : names) {
    int sum = 0;

    for(int i=0; i<name.length(); i++){
        inner char ch = name.charAt(i);
        loop  int letterNumber = ch - 96; //converts an 'a' to 1, 'b' to 2, etc.
            sum = sum + letterNumber;
    }

    System.out.println("Lucky number for "+name+" is "+sum);
}
```

Example of a nested loop

The above code nests a regular for loop inside an enhanced for loop. The following is the output produced by this code:

```
Lucky number for ally is 50
Lucky number for bob is 19
```

```
Lucky number for charlie is 56
Lucky number for david is 40
```

One thing that you need to be very careful about when using nested loops is managing the loop variables correctly. Consider the following program that is supposed to calculate the sum of all values in a given multidimensional array of ints:

```
int[][] values = { {1, 2, 3}, {2, 3}, {2}, {4, 5, 6, 7} };

int sum = 0;
for(int i = 0; i<values.length; i++) {
    for(int j=0; j<values[i].length; i++) {
        sum = sum + values[i][j];
    }
}
System.out.println("Sum is "+sum);
```

Read the above code carefully and try to determine the output.

It actually throws an exception:	Exception in thread "main"
<code>java.lang.ArrayIndexOutOfBoundsException: 4</code>	

Observe that the inner for loop is incrementing `i` instead of incrementing `j`. Now, let's execute the code step by step:

Step 0: Before the outer loop starts, `values.length` is 4 and `sum` is 0.

Step 1: Outer loop is encountered - `i` is initialized to 0, `i<values.length` is `true` because 0 is less than 4, so, the loop is entered.

Step 2: Inner loop is encountered - `j` is initialized to 0, `j<values[i].length` is true because `i` is 0, therefore, `values[i]` refers to { 1, 2, 3} and `values[i].length` is 3. Therefore, the loop is entered.

Step 3: Loop body is executed. `sum` is assigned `0 + values[0][0]`, i.e., 1. `sum` is now 1.

Step 4: Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to 1.

Step 5: Inner loop's condition `j<values[i].length` is executed and it evaluates to `true` because `j` is still 0 and `i` is 1, so, `values[i]` refers to {2, 3} and `values[i].length` is 2. Thus, second iteration of the inner loop will start.

Step 6: Loop body is executed. `sum` is assigned `1 + values[1][0]`, i.e., 1+2. `sum` is now 3.

Step 7: Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to 2.

Step 8: Inner loop's condition `j<values[i].length` is executed and it evaluates to `true` because `j` is still 0 and `i` is 2, so, `values[i]` refers to {2} and `values[i].length` is 1. Thus, third iteration of the inner loop will start.

Step 9: Loop body is executed. `sum` is assigned `3 + values[2][0]`, i.e., 3+2. `sum` is now 5.

Step 10: Inner loop body is finished. Inner loop's updation section is executed, so `i` is incremented to 3.

Step 11: Inner loop's condition `j<values[i].length` is executed and evaluates to `true` because

`j` is still 0 and `i` is 3, so, `values[i]` refers to {4, 5, 6, 7} and `values[i].length` is 4. Thus, fourth iteration of the inner loop will start.

Step 12: Loop body is executed. `sum` is assigned `5 + values[3][0]`, i.e., `5 + 4`. `sum` is now 9.

Step 13: Inner loop body is finished. Inner loop's updation section is executed, so, `i` is incremented to 4.

Step 14: Inner loop's condition `j < values[i].length` is executed. Now, here we have a problem. `i` is 4 and because the length of the `values` array is only 4, `values[4]` exceeds the bounds of the array and therefore `values[i]` throws an `ArrayIndexOutOfBoundsException`.

This is a very common mistake and while in this case the exception message made it easy to find, such mistakes can actually be very difficult to debug as they may produce plausible but incorrect results. That is why, although there is no technical limit to how many levels deep you can nest the loops, most professionals avoid nesting more than two levels.

Exam Tip

In the exam you will be presented with questions containing two levels of loops. Many students ask if there is an easy way to figure out the output of such loops. Unfortunately, there is no short cut. It is best if you execute the code step by step while keeping track of the variables at each step as shown above on a piece of paper. The code in the exam will be tricky and it may trip you up even if you are an experienced programmer.

As an exercise, try executing the same code given above after changing `i++` to `j++`.

8.7.2 breaking out of and continuing with nested loops ↗

You can use break and continue statements in nested loops exactly like I showed you earlier with single loops. They will let you break off or continue with the loop in which they are present. For example, check out the following code that tries to find "bob" in multiple groups of names:

```
String[][] groups = { { "ally", "bob", "charlie" } , { "bob", "alice", "boone"}, {  
    "chad", "dave", "elliot" } };  
  
for(int i = 0; i<groups.length; i++){  
    for(String name : groups[i]){  
        System.out.println("Checking "+name);  
        if("bob".equals(name)) {  
            System.out.println("Found bob in Group "+i);  
            break;  
        }  
    }  
}
```

The following is the output of the above code:

```
Checking ally  
Checking bob
```

```
Found bob in Group 0
Checking bob
Found bob in Group 1
Checking chad
Checking dave
Checking elliot
```

When the inner loop encounters the String "bob", it executes break, which causes the inner loop to end immediately. The control goes on to execute the next iteration of the outer loop, i.e., it starts looking for "bob" in the next group of names. This is the reason why the above output does not contain "Checking charlie", "Checking alice", and "Checking boone".

Now what if you want to stop checking completely as soon as you find "bob" in any of the groups? In other words, what if you want to break out of not just the inner loop but also out of the outer loop as soon as you find "bob"? Java allows you to do that. You just have to specify which loop you want to break out of. Here is how:

```
String[][] groups = { { "ally", "bob", "charlie" } ,
                      { "bob", "alice", "boone" },
                      { "chad", "dave", "elliot" } };

MY_OUTER_LOOP: for(int i = 0; i<groups.length; i++){
    for(String name : groups[i]){
        System.out.println("Checking "+name);
        if("bob".equals(name)) {
            System.out.println("Found bob in Group "+i);
            break MY_OUTER_LOOP;
        }
    }
}
```

The following is the output of the above code:

```
Checking ally
Checking bob
Found bob in Group 0
```

I just "labeled" the outer for loop with `MY_OUTER_LOOP` and used the same label as the target of the break statement. This is known as a "**labeled break**". A "**labeled continue**" works similarly.

Let me give a quick overview of labels and then I will get back to the rules of using labeled break and continue.

What is a label? ↗

A **label** is nothing but a name that you generally give to a **block of statements** or to statements that allow block of statements inside them (which means if, for, while, do-while, enhanced for, and

switch statements). The exam does not test you on the precise rules for applying labels or naming of labels, but here are few examples:

```
SINGLE_STMT: System.out.println("hello");

HELLO: if(a==b) callM1(); else callM2();

COME_HERE : while(i>=0) {
    System.out.println("hello");
}

SOME_LABEL: { //ok, because this is a block of statements
    System.out.println("hello1");
    System.out.println("hello2");
}
```

Here are some examples of invalid usage of labels:

```
BAD1 : int x = 0; //can't apply a label to declarations

BAD2 : public void m1() { } //can't apply a label to methods
```

Although not necessary, the convention is to use capital letters for label names. Also, applying a label to a statement doesn't necessarily mean that you have to use that label as a target of any break or continue. You can label a statement just for the sake of it :)

Rules for labeled break and continue

Getting back to using labeled break and continue, you need to remember that if you use a labeled break or a labeled continue, then that label must be present on one of the nesting loop statements within which the labeled break or continue exists. Thus, the following code will fail to compile:

```
LABEL_1 : for(String s : array) System.out.println(s); //usage of LABEL_1 is valid here
for(int i = 0; i<10; i++){
    if(i ==2) continue LABEL_1; //usage of continue is invalid because LABEL_1 does not
        appear on a loop statement that contains this continue.
}
```

The following is invalid as well:

```
for(int i = 0; i<10; i++){
    LABEL_1 : if(i ==2) System.out.println(2); //usage of LABEL_1 is valid here
    for(int j = 0; j<10; j++){
        if(i ==2) continue LABEL_1; //usage of continue is invalid because LABEL_1 does
            not appear on a loop statement that contains this continue.
    }
}
```

But the following is valid because the continue statement uses a label that nests the continue statement.

```
LABEL_1 : for(int i = 0; i<10; i++){
    if(i ==2) System.out.println(2);
    for(int j = 0; j<10; j++){
        if(i ==2) continue LABEL_1; //usage of continue is valid because it refers to an
        outer loop
    }
}
```

You can use labeled break and continue for non-nested loops as well but since unlabeled break and continue are sufficient for breaking out of and continuing with non-nested loops, labels are seldom used in such cases.

It is actually possible to use a labelled break (but not a labelled continue) inside any block statement. The block doesn't necessarily have to be a loop block. For example, the following code is valid and prints 1 3 .

```
public static void main (String[] args) {
    MYLABEL: {
        System.out.print("1 ");
        if( args != null) break MYLABEL;
        System.out.print("2 ");
    }
    System.out.print("3 ");
}
```

However, you need not spend time in learning about weird constructs because the exam doesn't test you on obscure cases. The `break` and `continue` statements are almost always used inside loop blocks and that is what the exam focuses on.

To answer the questions on break and continue in the exam correctly, you need to practice executing simple code examples on a piece of paper. The following is one such code snippet:

```
public static void doIt(int h){
    int x = 1;
    LOOP1 : do{
        LOOP2 : for(int y = 1; y < h; y++ ) {
            if( y == x ) continue;

            if( x*x + y*y == h*h){
                System.out.println("Found "+x+" "+y);
                break LOOP1;
            }
        }
        x++;
    }while(x<h);
}
```

Execute the above code mentally or on a piece of paper and find out what will it print when executed with 5 as an argument and then 6 as an argument.

8.8 Comparing loop constructs

8.8.1 Comparison of loop constructs

As you saw before, there are only a few technical differences between the four kind of loops. The **for**, **foreach**, and **while** loops are conceptually a little different than the **do-while** loop due to the fact that a **do-while** loop always executes at least one iteration. The **foreach** loop is a little different than other loops due to the fact that it can be used only for arrays and for classes that implement `java.lang.Iterable` interface.

Other than these differences, they are all interchangeable. For example, you can always replace a **while** loop with a **for** loop as follows:

```
while( booleanExpression ){
}
for( ; booleanExpression ; ){
}
```

You can also replace a **foreach** loop with a **for** loop as shown below:

```
for( Object obj : someIterable){
}

for( Iterator it = someIterable.iterator() ; it.hasNext() ; ){
    Object obj = it.next();
}
```

Of course, the **foreach** version is a lot simpler than the **for** version and that is the point. While you can use any of the loops for a given problem, you should use the one that results in code that is most understandable or intuitive.

A general rule of thumb is to use a **for** loop when you know the number of iterations before hand and use a **while** loop when the decision to perform the next iteration depends on the result of the operations performed in the prior iteration. For example, if you want to print "hello" ten times, use a **for** loop, but if you want to keep printing "hello" until the user enters a secret code as a response, use a **while** loop!

8.9 Exercise ↗

Note: Since most of the questions on loops involve arrays, do the following questions after going through the chapter on Arrays.

1. Initialize an array of 10 ints using a for loop such the each element contains an value equal to the sum of its previous two values. Do the same using a while loop.
2. Write a method that determines whether a given number N is a prime number or not by dividing that number with all the numbers from 2 to N/2 and checking the remainder.
3. Use nested for loops to print a list of prime numbers from 2 to N.
4. The following code contains a mistake. Identify the problem, fix it and print out all the elements of chars array.

```
String[] chars = new String[4];
char cha = 97;
for(char c=1;c<=chars.length; c++){
    chars[c] = ""+cha;
    cha++;
}
}
```

5. To avoid the possibility of inadvertently introducing the mistake shown in the above code, a programmer decided to use for-each loops instead of the regular for loops:

```
String[] chars = new String[4];
char cha = 97;
for(String s : chars){
    s = ""+cha;
    cha++;
}
```

Will this work? Why/why not?

6. Use an enhanced for loop to print alternate elements of an array. Can you use an enhanced for loop to print the elements in reverse order?
7. Given two arrays of same length and type, copy the elements of the first array into another in reverse order.

Exam Objectives

1. Use a one-dimensional array
2. Create and manipulate an ArrayList
3. Traverse the elements of an ArrayList by using iterators and loops including the enhanced for loop
4. Compare an array and an ArrayList

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

9.1.1 Declaring array variables

An array is an object that holds a fixed number of values of a given type. You can think of an array as a carton of eggs. If you have a carton with 6 slots, then that carton can hold only six eggs. Each slot of the carton can either have an egg or can be empty. Observe that the carton itself is not an egg. Similarly, if you have an array of size six of `int` values (or ints, for short), then that array can hold six ints but the array itself is not an `int`.

An array of a given type cannot hold anything else except values of that type. For example, an array of ints cannot hold long or double values. Similarly, you can have an array of references of any given type. For example, if you have an array of Strings, this array can only hold references to String objects. An important point to note here is that even though we call it an "array of strings", it does not actually contain String objects. It contains only references to String objects. You cannot really have an array that contains actual objects.

Array declaration

When you declare an array variable, you are basically specifying the type of values that you want to deal with through that variable. The way to specify that in Java is to apply `[]`, i.e., square brackets to the type of the values. For example, if you want a variable through which you will deal with `int` values, you will write `int[]`. Arrays can be multi-dimensional and there will be one set of opening and closing brackets for each dimension. For example,

```
int i; //i is an int
int[] ia1, ia2; //ia1 and ia2 are one dimensional arrays of ints
int[][] iaa; //iaa is a two dimensional array of ints and so on
```

An array declaration can never include the size of the array. Thus, the following are declarations will not compile:

```
int[2] invalid1;
int[3][] invalid2;
int[] [4] invalid3;
```

Unfortunately, the above method is not the only way to declare arrays. Java allows you to apply square brackets to the variable name instead of type name as well. For example,

```
int i, ia[]; //i is an int but ia is a one dimensional array of int values
int[] ia, iaa[]; //ia is a one dimensional array of ints but iaa is a two dimensional
                 array of ints and so on
```

Observe that the rule of thumb of one set of square brackets per dimension still holds. In the case of `ia`, you have one set applied on the type and one set applied on the variable, therefore `ia` is a two dimensional array.

Arrays of objects are declared the same way. For example,

```
Object[] obja, objaa[]; //obja is a one dimensional array of Objects but objaa is a two
                        //dimensional array of Objects
String[] strA; //strA is a one dimensional array of Strings
```

Note that the statements shown above only declare array variables. They don't actually create arrays. Array creation and initialization is a topic in its own right and that is what I will discuss next.

9.1.2 Creating and initializing array objects ↗

Creating arrays using array creation expressions

You use the new keyword to create an array object. For example,

```
int[] ia = new int[10]; //an array of ints of size 10
boolean[] ba = new boolean[3]; //an array of booleans of size 3
String[] stra = new String[5]; //an array of Strings of size 5
MyClass[] myca = new MyClass[5]; //an array of MyClass of size 5
```

```
int[] invalid = new int[]; //missing size. will not compile
```

The parts on the right-hand side of = sign in the above statements are called "**array creation expressions**". These expressions merely allocate space to hold **10 ints**, **3 booleans**, **5 string references**, and **5 MyClass references** respectively. Every element of the array is also initialized to its default value automatically by the JVM. The default values of array elements are very straightforward - references are initialized to **null**, numeric primitives to **0**, and booleans to **false**. In the above lines of code, **ia** is set to point to an array of ten int values and each element of the array is initialized to 0, **ba** is set to point to an array of three boolean values and each element of the array is initialized to false, **stra** is set to point to an array of five **String** references and each element of the array is initialized to **null**, and finally, **myca** is set to point to an array of five **MyClass** references and each element of the array is initialized to **null**. Observe that all the elements of an array are initialized to the same value. This shows us another important aspect of arrays - that arrays can have **duplicate values**.

It is important to understand that, in the above statements, we are not creating instances of the class of the array elements. In other words, we are not creating instances of **String** or instances of **MyClass**. (We are not creating instances of ints or booleans either, for that matter, but since they are primitives, and since primitives are not objects, they don't have instances anyway.)

In Java, arrays, whether of primitives or objects, are objects of specific classes. In other words, an array object is an instance of some class. It is not an instance of Object class but since Object is the root of every class in Java, an array object is an Object and all methods of the Object class can be invoked on an array. Let us now look at the following program and its output to know more about the class of the above defined array objects -

```
public class TestClass{
    public static void main(String[] name){
        int[] ia = new int[10];
        boolean[] ba = new boolean[3];
```

```

String[] stra = new String[5];
TestClass[] ta = new TestClass[5];
System.out.println(ia.getClass().getName()+""
"+ia.getClass().getSuperclass().getName());
System.out.println(ba.getClass().getName()+""
"+ba.getClass().getSuperclass().getName());
System.out.println(stra.getClass().getName()+""
"+stra.getClass().getSuperclass().getName());
System.out.println(ta.getClass().getName()+""
"+ta.getClass().getSuperclass().getName());
}
}

```

Output:

```

[I , java.lang.Object
[Z , java.lang.Object
[Ljava.lang.String; , java.lang.Object
[LTestClass; , java.lang.Object

```

The output shows that `ia`, which is declared to be of type `int[]`, is not an instance of `int` but of a class named `[I`. `stra`, which is declared to be of type `String[]`, is not an instance of `String` but of a class named `[Ljava.lang.String` and so on. These names of the classes look weird. Actually, Java cooks up the name of the class of an array by looking at the number of dimensions and the type of the elements. For each dimension, there is one opening square bracket. This is followed by a letter for the class of the elements as per the following table and, if the array is not of a primitive, the name of the class followed by a semi-colon.

Type	Letter
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
any object	L

Table 9.1.1: Letters used for creating class names of arrays

Based on the above table, the name of the class for `long[][]` (i.e. a two dimensional array of longs) would be `[[J` and the name of the class for `mypackage.SomeClass[][][]` (i.e. a three dimensional array of `mypackage.SomeClass`) would be `[[[Lmypackage.SomeClass;`.

Although the above discussion about the class of arrays is not required for the JFCJA exam, it is good to know anyway.

Creating arrays using array initializers

In the previous section, we created array objects using the new keyword. It is possible to create array objects without using the new keyword. For example, the arrays that we created above can also be created as follows:

```
int[] ia = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; //an array of ints of size 10
boolean[] ba = {true, false, false}; //an array of booleans of size 3
String[] str = {"a", "b", "c", "d", "e"}; //an array of Strings of size 5
MyClass[] myca = { new MyClass(), new MyClass(), new MyClass(), new MyClass(), new
    MyClass() } ; //an array of MyClass objects of size 5
```

The parts on the right-hand side of the = sign in the above statements are called "**array initializers**". An array initializer is a shortcut that allows you to create as well as initialize each element of the array with the values that you want (instead of the default values that you get when you use array creation expressions).

Since the compiler can find the type of the elements of the array by looking at the declaration (i.e. the left-hand side of the statement), specifying the same on the right-hand side is not required. Similarly, the compiler finds out the length of the array as well by counting the number of values that are specified in the initializer.

Array initializers can also be mixed with array creation expressions. For example:

```
int[] ia = new int[]{ 1, 2, 3, 4, 5 };
```

Observe that size is missing from the expression on the right-hand side. Java figures out the size of the array by counting the number of elements that are specified in the initialization list. In fact, it is prohibited to specify the size if you are specifying individual elements. Therefore, the following is invalid:

```
int[] ia = new int[2]{ 1, 2 }; //will not compile.
```

9.2 Using arrays

9.2.1 Array indexing

In Java, array indexing starts with 0. In other words, the index of the first element of an array is 0. For example, if you have an array variable named `accounts` that refers to an array of 100 `Account` objects, then you can access the first object through `accounts[0]` and the 100th object through `accounts[99]`. `accounts[0]` or `account[99]` are like any other variable of type `Account`.

Similarly, if you have an array variable `ia` pointing an array of 5 ints, the first element can be accessed using `ia[0]` and the last element using `ia[4]`.

java.lang.ArrayIndexOutOfBoundsException

If you try to access any array beyond its range, JVM will throw an instance of `ArrayIndexOutOfBoundsException`. For example, if you have `int[] ia = new int[3];` the statements `int i = ia[-1];` and `int i = ia[3];` will cause an `ArrayIndexOutOfBoundsException` to be thrown. Do not worry if you don't know much about exceptions at this point. I will discuss exceptions in detail later.

Arrays of length zero

As strange as it may sound, it is possible to have an array of length 0. For example, `int[] ia = new int[0];` Here, ia refers to an array of ints of length 0. There are no elements in this array. It is important to understand that an array of length 0 is not the same as `null`. `ia = null` implies that ia points to nothing. `ia = new int[0];` implies that ia points to an array of ints whose length is 0.

A good example of an array of length 0 is the `args` parameter of the main method. If you run a class with no argument, `args` will **not** be `null` but will refer to an array of Strings of length 0.

Changing the size of an array

Java arrays are always of fixed size(or length). Once you create an array, you cannot change the number of elements that this array can have. So, for example, if you have created an array of 5 ints and if you have 6 int values to store, you will need to create a new int array. There is no way to increase or decrease the size of an existing array. You may change the array variable to point to a different array, and you can, of course, change the values that an array contains.

9.2.2 Members of an array object

We saw earlier that arrays are actually objects of specific classes. We also saw the names of some of these classes. But what do these classes contain? What are the members of these classes? What functionality do these classes provide? Let us take a look:

Array length

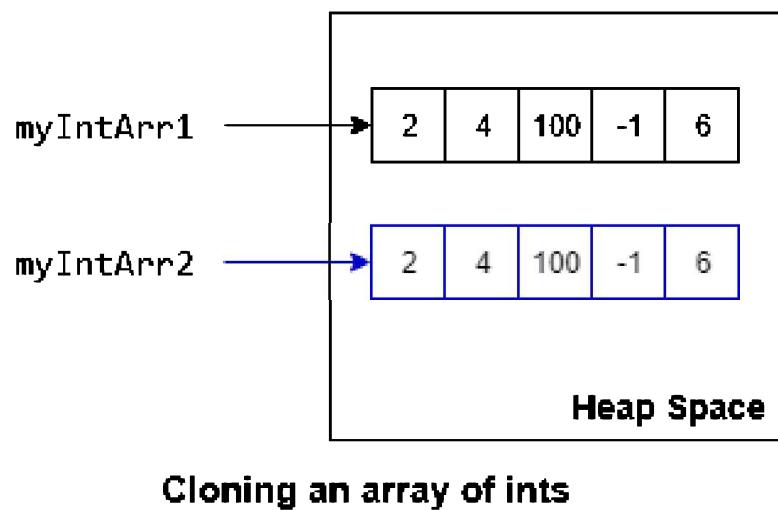
All array classes have one field named `length`, which is of type `int`. This field is **public** and it stores the length of the array. This field is also **final**, which reflects the fact that you cannot change the length of an array after its creation. Since the length of an array can never be less than 0, the value of this field can never be less than 0 either.

Array cloning

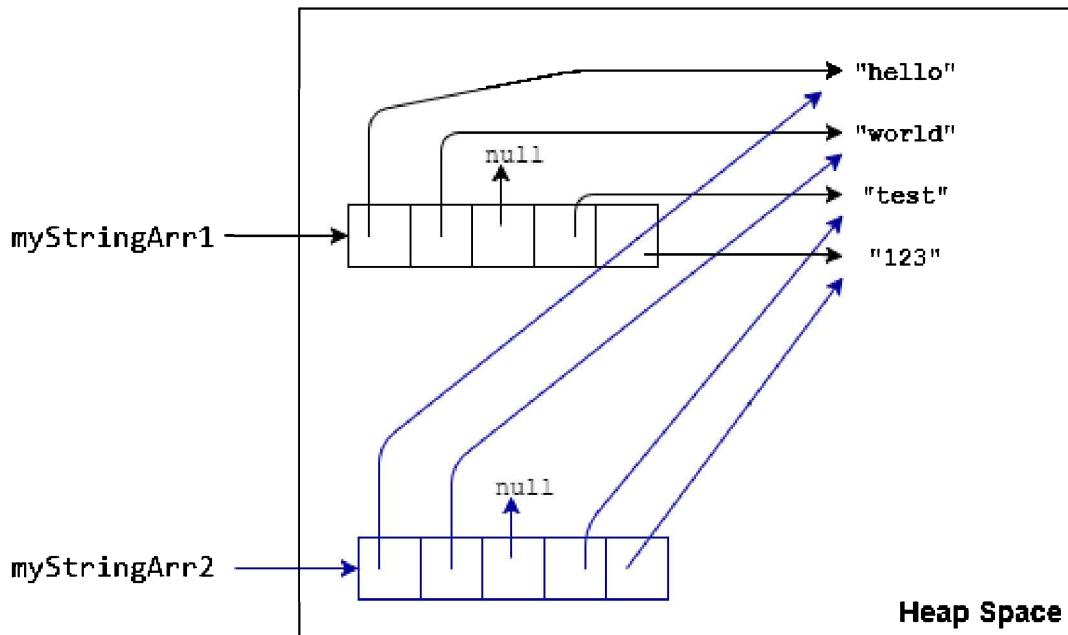
All array classes also have a public method named `clone`. This method creates a copy of the array object. Note that it doesn't create copies of the objects referred to by the array elements. It merely creates a new array object of the same length and copies the contents of existing array into the new array. Which means, if the existing array contained primitive values, those values will be copied to the elements of the new array. If the existing array contained references to objects, those references will be copied to the elements of the new array. Thus, the elements of the new array will also point to the same objects. This is also known as "**shallow copy**".

For example, the following figure shows exactly what happens when you clone an array of five ints referred to by the variable `myIntArr1` using the statement `int[] myIntArr2 = (int[])`

`myIntArr1.clone();` Don't worry about the explicit cast for now, but observe that a new array object with the same number of slots is created and the contents of the slots of the existing array are copied to the slots of the new array.



The following figure shows what happens when you clone an array of Strings referred to by the variable `myStringArr1` using the statement `String[] myStringArr2 = (String[]) myStringArr1.clone();` Observe that the references are copied into the new array and that no new String objects are created in the heap space.



Cloning an array of Strings

Shallow copying of an array of ints

Cloning is not required for the exam but it is an important aspect of arrays anyway.

Members inherited from Object class

Remember that since `java.lang.Object` is the root class of all classes, it is the root class of all array classes as well and therefore, array classes inherit all the members of the `Object` class. This includes `toString`, `equals`, and `hashCode` methods.

9.2.3 Runtime behavior of arrays

Although arrays are the simplest of data structures, they are not without their quirks. To use arrays correctly and effectively, you must be aware of the two most important aspects of arrays in Java.

The first is that they are **"reified"**. Meaning, the type checking of arrays and its elements is done at runtime by the JVM instead of at compile time by the compiler. In other words, the type information of an array is preserved in the compiled bytecode for use by the JVM during run time. The JVM knows about the type of an array and enforces type checking on arrays.

For example, if you have an array of Strings, the JVM will not let you set any element of that array to point to any object other than a String, even though the compiler may not notice such a violation as illustrated by the following code:

```
String[] sa = { "1", "2", "3" };
Object[] oa = sa;
oa[0] = new Object();
```

The above code will compile fine. The compiler has no objection when you try to assign an array of Strings to a variable of type array of Objects. It has no objection when you try to set an element of this array of Strings to point to an object that is not a String. But when you run it, the JVM will throw a `java.lang.ArrayStoreException` on the third line. This is because the JVM knows that the array pointed to by `oa` is actually an array of Strings. It will not let you corrupt that array by storing random objects in it.

It is important to understand this concept because it is diametrically opposite to how generics work. This is the reason why arrays and generics don't operate well with each other. You will realize this when you learn about generics later.

The second thing about arrays is that they are "**covariant**". Meaning, you can store a subclass object in an array that is declared to be of the type of its superclass. For example, if you have an array of type `java.lang.Number`, you can store `java.lang.Integer` or `java.lang.Float` objects into that array because both are subclasses of `java.lang.Number`. Thus, the following code will compile and run fine:

```
Number[] na = { 1, 2, 3 };
na[0] = new Float(1.2f);
```

I haven't seen anyone getting a question on array reification and covariance in the JFCJA exam. However, any discussion about arrays is incomplete without these two points and that is why I have talked about them here.

9.2.4 Uses of arrays ↗

Arrays are quite powerful as a **data structure** but they are somewhat primitive as a **data type**. As we saw earlier, arrays have only one field and merely a couple of methods. But because of their simplicity, arrays are used as building blocks for other data types and data structures. For example, the `String` data type is built upon an array of `chars`. So are `StringBuffer` and `StringBuilder`. These higher level classes really only wrap an array of characters and provide methods for manipulating that array.

Arrays are also used extensively for building higher level data structures such as `List`, `Stack`, and `Queue`. You may have come across the `ArrayList` class (this class is also on the exam, by the way, and I will discuss it later). Guess what, it is a `List` that manages its collection of objects using an array inside. Since these classes provide a lot of additional features on top of arrays, more often than not, it is these classes that get used in application programs rather than raw arrays.

java.util.Arrays class

Java standard library does include a utility class named **Arrays** in package **java.util** that makes working with raw arrays a little easier. **java.util.Arrays** class contains a large number of static utility methods for manipulating any given array object. Although you are not required to know about this class for the JFCJA exam, it is a good idea to browse through the JavaDoc API description of **Arrays** class to see what you can do with arrays.

9.3 Create and manipulate an ArrayList

9.3.1 Introduction to Collections and Generics

Processing multiple objects of the same kind in the same way is often a requirement in applications. Printing the names of all the Employees in a list of Employees, computing interest for a list of Accounts, or something as simple as computing the average of a list of numbers, require you to deal with a list of objects instead of one single object. You have already seen a data structure that is capable of holding multiple objects of the same kind - array. An array lets you hold references to multiple objects of a type and pass them around as a bunch. However, arrays have a couple of limitations. First, an array cannot grow or shrink in length after it is created. If you create an array of 10 Employees and later find out you have 11 Employees to hold, you will need to create a new array with 11 slots. You can't just add one more slot in an existing array. Second, inserting a value in the middle of an array is a bit difficult. If you want to insert an element just before the last element, you will have to first shift the last element to the next position and then put the value. Imagine if you want to insert an element in the middle of the list!

Although both of the limitations can be overcome by writing some extra code, these requirements are so common that writing the same code over and over again is just not a good idea. **java.util.ArrayList** is a class that incorporates these, and several other, features out of the box. The following is an example that shows how easy it is to manage a list of objects using an **ArrayList**:

```
import java.util.ArrayList;
public class TestClass{
    public static void main(String[] args){
        ArrayList al = new ArrayList();

        al.add("alice"); // [alice]
        al.add("bob"); // [alice, bob]
        al.add("charlie"); // [alice, bob, charlie]

        al.add(2, "david"); // [alice, bob, david, charlie]

        al.remove(0); // [bob, david, charlie]

        for(Object o : al){ // process objects in the list
            String name = (String) o;
            System.out.println(name+" "+name.length());
        }
    }
}
```

```
//dump contents of the list
System.out.println("All names: "+al);
}
}
```

You will get a few warning messages when you compile the above program. Ignore them for now. It prints the following output when run:

```
bob 3
david 5
charlie 7
All names: [bob, david, charlie]
```

The above program illustrates the basics of using an `ArrayList`, i.e., adding objects, removing, iterating over them, and printing the contents of the `ArrayList`. I suggest you write a program that does the same thing using an array of strings instead of an `ArrayList`. This will give you an appreciation for the value that `ArrayList` adds over a raw array.

I showed you the above code just to give you a glimpse of one of the readymade classes in the Java library. There is a lot more that you can do with it but before we get to that, you need to understand the bigger picture, the grand scheme of things, into which classes such as `ArrayList` fit.

Collections and Collections API

`ArrayList` belongs to a category of classes called "**collections**". A "**collection**" is nothing but a group of objects held up together in some form. You can visualize a collection as a bag in which you put several things. Just like you use a bag to take various grocery items from a store to your home, you put the objects in a collection so that you can pass them around to other objects and methods. A bag knows nothing about the items themselves. It merely holds the items. You can add items to it and take items out of it. It doesn't care about the order in which you put the items or whether there are any duplicates or how they are stored inside the bag. The behavior of a bag is captured by an interface called `Collection`, which exists in the `java.util` package.

Now, what if you want a collection that ensures objects can be retrieved from it in the same order in which they were added to it? Or what if you want a collection that does not allow duplicate objects? You can think of several features that can be added on top of a basic collection. As you have already learnt previously, subclassing/subinterfacing is the way you extend the behavior of an existing class or interface. The Java standard library takes the same route here and extends `java.util.Collection` interface to provide several interfaces with various features. In fact, the Java standard library provides a huge tree of interfaces along with classes that implement those interfaces. These classes and interfaces are collectively known as the "**Collections API**".

So what has `ArrayList` got to do with this, you ask? Well, `ArrayList` is a class that implements one of the specialized collection interfaces called `List`. `java.util.List` extends `java.util.Collection` to add the behavior of ordering on top of a simple collection. A `List`

is supposed to remember the order in which objects are added to the collection. Okay, so that takes care of the "List" part of ArrayList, what about the "Array" part? Again, as you have learnt in previous chapters, an interface merely describes a behavior. It doesn't really implement that behavior. You need a class to implement the behavior. In this case, `ArrayList` is the class that implements the behavior described by `java.util.List` and to implement this behavior, it uses an array. Hence the name `ArrayList`. Remember I talked about writing code to overcome the limitation of an array that it is not flexible enough to increase in size automatically? `ArrayList` contains that code. When you add an object to an `ArrayList`, it checks whether there is space left in the array. If not, it allocates a new array with a bigger length, copies the elements from the old array to this new array, and then adds the passed object to the new array. Similarly, it contains code for inserting an object in the middle of the array. All this is transparent to the user of an `ArrayList`. A user simply invokes methods such as `add` and `remove` on an `ArrayList` without having any knowledge of the array that is being used inside to store the object.

Generics

Now, regarding the warning messages that I asked you to ignore. Observe the `for` loop in the above code. The type of the loop variable `o` is `Object` (and not `String`). To invoke the `length` method of `String` on `o`, we need to cast `o` to `String`. This is because, in the same way that a bag is unaware of what is inside of it, so too does the `ArrayList` object have no knowledge about the type of the objects that have been added to it. It is the responsibility of the programmer to cast the objects that they retrieve from the list to appropriate types. In this program, we know that we have added `Strings` to this list and so we know that we can cast the object that we retrieve from this list to `String`. But what if we were simply given a pre-populated list as an argument? Since the list would have been populated by another class written by someone else, what guarantee would we have about the type of objects we find in the list? None, actually. And as you are aware, if we try to cast an object to an inappropriate type, we get a `ClassCastException` at run time. Getting a `ClassCastException` at run time would be a coding mistake that will be discovered only at run time. Discovering coding mistakes at run time is not a good thing and the compiler is only trying to warn us about this potential issue by generating warning messages while compiling our code.

The first warning message that it prints out is, "`Warning: unchecked call to add(E) as a member of the raw type java.util.ArrayList`" for the code `al.add("alice");` at line 6. It prints the same warning every time an object is added to the list. The compiler is trying to tell us that it does not know what kind of objects this `ArrayList` is supposed to hold and that it has no way of checking what we are adding to this list. By printing out the warning, the compiler is telling us that it will not be held responsible for the objects that are being put in this list. Whoever wants to retrieve objects from this list must already know what type of objects are present inside the list and must cast those objects appropriately upon retrieval at their own risk. In other words, this list is basically "**type unsafe**". It is unsafe because it depends on assumptions made by humans about the contents of the list. This assumption is not checked or validated by the compiler for its truthfulness. One can put any kind of object in the list and others will not realize until they are hit with a `ClassCastException` at run time when they try to cast the retrieved object to its expected type.

Java solves this problem by allowing us to specify the type of objects that we are going to store in a list while declaring the type of the list. If, instead of writing `ArrayList al = new ArrayList();`, we write `ArrayList<String> al = new ArrayList<String>();`; all the warning messages go away. The compiler now knows that the `ArrayList` pointed to by `al` is supposed to hold only `Strings`. Therefore, it will be able to keep a watch on the type of objects the code tries to add this list. If it sees any code that attempts to add anything that is not a `String` to the list, it will refuse to compile the code. The error message generated by the following code illustrates this point:

```
ArrayList<String> al = new ArrayList<String>();
al.add(new Object());
```

The error message is:

```
Error: no suitable method found for add(java.lang.Object)
method java.util.Collection.add(java.lang.String) is not applicable
```

The compiler will not let you corrupt the list by adding objects that this list is not supposed to keep. Similarly, the compiler will not let you make incorrect assignments either. Here is an example:

```
List<String> al = new ArrayList<String>(); //al points to a List of Strings
al.add("hello"); //valid
String s = al.get(0); //valid, no cast required
Object obj = al.get(0); //valid because a String is-a Object
Integer in = al.get(0); //Invalid. Will not compile
```

The error message is:

```
Error: incompatible types: java.lang.String cannot be converted to java.lang.Integer
```

Observe that we supplied information about the type of object the list is supposed to keep by appending `<String>` to the variable declaration and the `ArrayList` creation. This way of specifying the type information is a part of a feature introduced in Java 5 known as "generics".

The above discussion should tell you that the topics of List/ArrayList, Collections, and Generics are interrelated. All three of them combine together to provide you a powerful mechanism to write type safe and bug free code. However, the JFCJA exam requires you to know only the basic usage of the ArrayList class. You are not expected to know about Collections and Generics and so, I will not talk about them anymore in this book.

9.3.2 ArrayList API

As discussed earlier, ArrayList is one of the implementation classes of the List interface. It has three constructors:

1. `ArrayList()`: Constructs an empty list with an initial capacity of 10. Just like you saw with the `StringBuilder` class, capacity is simply the size of the initial array that is used to store

the objects. As soon as you add the 11th object, a new array with bigger capacity will be allocated and all the existing objects will be transferred to the new array.

2. `ArrayList(Collection c)`: Constructs a list containing the elements of the specified collection.
3. `ArrayList(int initialCapacity)`: Constructs an empty list with the specified initial capacity. This constructor is helpful when you know the approximate number of objects that you want to add to the list. Specifying an initial capacity that is greater than the number of objects that the list will hold improves performance by avoiding the need to allocate a new array every time it uses up its existing capacity. It is possible to increase the capacity of an `ArrayList` even after it has been created by invoking `ensureCapacity(int n)` on that `ArrayList` instance. Calling this method with an appropriate number before inserting a large number of elements in the `ArrayList` improves performance of the add operation by reducing the need for incremental reallocation of the internal array. The opposite of `ensureCapacity` is the `trimToSize()` method, which gets rid of all the unused space by reducing its capacity to the match the number of elements in the list.

Here are a few declarations that you may see on the exam:

1. `List list = new ArrayList(); //ok because ArrayList is a List`
2. `ArrayList<String> al2 = new ArrayList<String>(list); //copying an existing list, observe the diamond operator`
3. `List<Student> list1 = new ArrayList<>(); //ok, list1 is of type List<Student>`

The third declaration is interesting. We have used `<Student>` on the left side of `=` but only `<>` on the right side. `<>` is called the "diamond operator". It just means that the compiler should take type information from the left side of `=`. Since we have already declared `list1` to be a `List` of `Student` objects, there is no need for us to type this information again on the right side. Thus, all it does is save us from typing a few extra keystrokes.

Important methods of ArrayList

`ArrayList` has quite a lot of methods. However, since `ArrayList` implements `List` (which, in turn, extends `Collection`), several of `ArrayList`'s methods are declared in `List` and `Collection` interfaces. The exam does not expect you to make a distinction between the methods inherited from `List` / `Collection` and the methods declared in `ArrayList`.

The following are the ones that you need to know for the exam:

1. `String toString()`: Well, `toString` is not really the most important method of `ArrayList` but since we will be depending on its output in our examples, it is better to know about it anyway. `ArrayList`'s `toString` first gets a string representation for each of its elements (by invoking `toString` on them) and then combines into a single string that starts with `[` and ends with `]`. For example, the following code prints `[a, b, c]`:

```
var al = new ArrayList<String>();
al.add("a");
al.add("b");
al.add("c");
System.out.println(al);
```

Observe the order of the elements in the output. It is the same as the order in which they were added in the list. Calling `toString` on an empty `ArrayList` gets you `[]`. I will use the same format to show the contents of a list in code samples.

Methods that add elements to an `ArrayList`:

1. `boolean add(E e)`: Appends the specified element to the end of this list. As discussed in the Generics section, `E` is just a place holder for whichever type you specify while creating the `ArrayList`. For example, if you create an `ArrayList` of Strings, i.e., `ArrayList<String>`, `E` stands for `String`.

This method is actually declared in `Collection` interface and the return value is used to convey whether the collection was changed as a result of calling this method. In case of an `ArrayList`, the `add` method always adds the given element to the list (even if the element is null), which means it changes the collection every time it is invoked. Therefore, it always returns `true`.

2. `void add(int index, E element)`: Inserts the specified element at the specified position in this list. The indexing starts from `0` and the maximum value of `index` can be `size`. Therefore, if you call `add(0, "hello")` on an list of `Strings`, `"hello"` will be inserted at the first position.
3. `boolean addAll(Collection<? extends E> c)`: Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator. Again, for the purpose of the Part 1 exam, you don't need to worry about the `? extends E` or the Iterator part. You just need to know that you can add all the elements of one list to another list using this method. For example,

```
ArrayList<String> sList1 = new ArrayList<>(); //observe the usage of the diamond
operator
sList1.add("a"); // [a]
ArrayList<String> sList2 = new ArrayList<>();
sList2.add("b"); // [b]
sList2.addAll(sList1); // sList2 now contains [b, a]
```

4. `boolean addAll(int index, Collection<? extends E> c)`: This method is similar to the one above except that it inserts the elements of the passed list in the specified collection into this list, starting at the specified position. For example,

```
ArrayList<String> sList1 = new ArrayList<>();
sList1.add("a"); // [a]
```

```
ArrayList<String> sList3 = new ArrayList<>();
sList3.add("b"); // [b]
sList3.addAll(0, sList1); // sList3 now contains [a, b]
```

Methods that remove elements from an `ArrayList`:

1. `E remove(int index)`: Removes the element at the specified position in this list. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, c]
String s = list.remove(1); // list now has [a, c]
```

It returns the element that has been removed from the list. Therefore, `s` will be assigned the element that was removed, i.e., "b". If you pass an invalid int value as an argument (such as a negative value or a value that is beyond the range of the list, i.e., `size-1`), an `IndexOutOfBoundsException` will be thrown.

2. `boolean remove(Object o)`: Removes the first occurrence of the specified element from this list, if it is present. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, a]
list.remove("a"); // [b, a]
```

Observe that only the first `a` is removed.

You have to pay attention while using this method on an `ArrayList` of `Integers`. Can you guess what the following code will print?

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 } ));
list.remove(1);
System.out.println(list);
list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 } ));
list.remove(new Integer(1));
System.out.println(list);
```

The output is:

```
[1, 3]
[2, 3]
```

Recall the rules of method selection in the case of overloaded methods. When you call `remove(1)`, the argument is an `int` and since a `remove` method with `int` parameter is available, this method will be preferred over the other `remove` method with `Object` parameter because invoking the other method requires boxing `1` into an `Integer`.

This method returns `true` if an element was actually removed from the list as a result of this call. In other words, if there is no element in the list that matches the argument, the method will return `false`.

3. `boolean removeAll(Collection<?> c)`: Removes from this list all of its elements that are contained in the specified collection. For example the following code prints [c]:

```
ArrayList<String> al1 = new ArrayList<>(Arrays.asList( new String[]{"a", "b", "c", "a" } ));
ArrayList<String> al2 = new ArrayList<>(Arrays.asList( new String[]{"a", "b" } ));
al1.removeAll(al2);
System.out.println(al1); //prints [ c ]
```

Observe that unlike the `remove(Object obj)` method, which removes only the first occurrence of an element, `removeAll` removes all occurrences of an element. This method returns `true` if an element was actually removed from the list as a result of this call.

4. `void clear()`: Removes all of the elements from this list.

Methods that replace an element in an `ArrayList`:

1. `E set(int index, E element)`: Replaces the element at the specified position in this list with the specified element. It returns the element that was replaced. Example:

```
ArrayList<String> al = ... // create a list containing [a, b, c]
String oldVal = al.set(1, "x");
System.out.println(al); //prints [a, x, c]
System.out.println(oldVal); //prints b
```

Methods that read an `ArrayList` without modifying it:

1. `boolean contains(Object o)`: The object passed in the argument is compared with each element in the list using the `equals` method. A `true` is returned as soon as a matches is found, a `false` is returned otherwise. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c" } ));
System.out.println(al.contains("c")); //prints true
System.out.println(al.contains("z")); //prints false
System.out.println(al.contains(null)); //prints true
```

Observe that it does not throw a `NullPointerException` even if you pass it a `null`. In fact, a `null` argument matches a `null` element.

2. `E get(int index)`: Returns the element at the specified position in this list. It throws an `IndexOutOfBoundsException` if an invalid value (i.e. a value less than `0` or greater than `size-1`) is passed as an argument.
3. `int indexOf(Object o)`: The object passed in the argument is compared with each element in the list using the `equals` method. The index of the first element that matches is returned. If this list does not contain a matching element, `-1` is returned. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c", null } ));
System.out.println(al.indexOf("c")); //prints 3
System.out.println(al.indexOf("z")); //prints -1
System.out.println(al.indexOf(null)); //prints 1
```

Observe that just like `contains`, `indexOf` does not throw a `NullPointerException` either even if you pass it a `null`. A `null` argument matches a `null` element.

4. `boolean isEmpty()`: Returns `true` if this list contains no elements.
5. `int size()`: Returns the number of elements in this list. Recall that to get the number of elements in a simple array, you use the variable named `length` of that array.

The examples that I have given above are meant to illustrate only a single method. In the exam, however, you will see code that uses multiple methods. Here are a few points that you should remember for the exam:

1. **Adding nulls:** `ArrayList` supports `null` elements.
2. **Adding duplicates:** `ArrayList` supports duplicate elements.
3. **Exceptions:** None of the `ArrayList` methods `except toArray(T[] a)` throw `NullPointerException`. They throw `IndexOutOfBoundsException` if you try to access an element beyond the range of the list.
4. **Method chaining:** Unlike `StringBuilder`, none of the `ArrayList` methods return a reference to the same `ArrayList` object. Therefore, it is not possible to chain method calls.

Here are a few examples of the kind of code you will see in the exam. Try to determine the output of the following code snippets when they are compiled and executed:

1. --

```
var al = new ArrayList<Integer>(); //observe that the type specification is on
    the right side
al.add(1).add(2);
System.out.println(al);
```

2. --

```
ArrayList<String> al = new ArrayList<>(); //observe the usage of the diamond
operator
if( al.add("a") ){
    if( al.contains("a") ){
        al.add(al.indexOf("a"), "b");
    }
}
System.out.println(al);
```

3. --

```
ArrayList<String> al = new ArrayList<>();
al.add("a"); al.add("b");
al.add(al.size(), "x");
System.out.println(al);
```

4. --

```
var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>();
list1.add("a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.remove("b");
System.out.println(list1);
```

5. --

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add("a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
System.out.println(list1);
list1.remove("b");
System.out.println(list1);
```

6. --

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add("a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.removeAll("b");
System.out.println(list1);
```

Size vs Capacity

Size is the number of elements that are currently stored in the ArrayList. While **capacity** is the length of the internal array that an ArrayList uses to store its elements.

You know that in Java, arrays are of fixed length. You must specify the length of an array while creating it and you can never change this length. So, what happens when you add more elements to an ArrayList than its capacity? Simple. A new array with bigger length is allocated and elements are transferred from the old array to the new array. Since all this is managed internally by the ArrayList, you don't have to worry about it most of the time.

However, it has implications on the performance of an ArrayList. Since an ArrayList has no idea how many elements are going to be added to it, it starts with a capacity of 10 and keeps incrementing the capacity as soon as it is full (the exact algorithm for incrementing the capacity is not important here). Now, imagine if you wanted to add a thousand elements to an ArrayList. This could potentially require the ArrayList to perform the allocation and transfer operation, which is quite expensive, several times. This will reduce its performance.

This is where the `ArrayList(int capacity)` constructor and `ensureCapacity(int capacity)` method come in handy. If you know the number of elements you are going to add to the ArrayList, you should use this constructor/method to make sure that the internal array of the ArrayList is big enough to hold all the elements.

Remember that insertion of an element depends the **size** (and not **capacity**) of the ArrayList. For example, if the size of an ArrayList is 5, you can't insert an element at index 6 even if the capacity of that ArrayList is 10. It will throw an `IndexOutOfBoundsException`.

9.4 Traversing the elements of an ArrayList

In the chapter on Loops, we discussed all kinds of loops and how to use them to iterate over an array. The same techniques are applicable while iterating over an `ArrayList`. Let's see an example to make it clear:

```
import java.util.ArrayList; //importing the ArrayList class
class Student{
    String name;
    Student(String n){ name = n; }
}
class TestClass{
    public static void main(String[] args){
        List<Student> students = new ArrayList<>();
        //adding a few Students to the list
        students.add(new Student("alice")); students.add(new Student("bob"));
```

```
students.add(new Student("chad"));
int size = students.size();

//using a regular for loop
for(int i=0; i<size; i++){
    System.out.println(students.get(i).name);
}

//using the enhanced for loop
for(Student s : students){
    System.out.println(s.name);
}

//using java.util.Iterator
java.util.Iterator it = students.iterator();
while(it.hasNext()){
    Student s = it.next();
    System.out.println(s.name);
}

}
```

The first two are the most common ways to iterate through a list and are very straight forward. The third way, which uses an Iterator, is a little complicated and rarely used. Unfortunately, the JFCJA exam expects you to know about it even though the underlying concepts that it touches upon are not on the exam. So, just remember the following - `ArrayList` has a method named `iterator()`, which returns a `java.util.Iterator` object. The Iterator object has two methods - `hasNext()` and `next()`. The `hasNext` method returns a `boolean` that tells you whether there are more elements that haven't yet been iterated upon and the `next` method returns the next `Object` in line to be iterated upon.

So, basically, we use `it.hasNext()` as the while condition, and inside the loop we retrieve the next available element using `it.next()`. Let's see how our loop executes step by step:

1. At the beginning, `it.hasNext()` returns `true` (because we haven't processed any elements yet and there are three elements to go in the list) and we enter the loop body. Inside the body, we call `it.next()`, which returns the first element "`alice`". We print it out and loop back to the while condition.
2. `it.hasNext()` returns `true` again, and we enter the loop. Inside, `it.next()` returns the second element "`bob`". We print out the second element and loop back to the while condition.
3. Again, `it.hasNext()` returns `true` and `it.next()` returns the third element "`chad`". We print the third element out and then loop back to the while condition.

4. Now, `it.hasNext()` returns `false` because there are no more elements waiting to be iterated upon in the list. Thus, the loop ends.

9.5 ArrayList vs array

You may get a few theoretical questions in the exam about the advantages and disadvantages of an ArrayList over an array. You have already seen all that we can do with ArrayLists and arrays, so, I am just going to summarize their advantages and disadvantages here.

Advantages of ArrayList

1. **Dynamic sizing** - An ArrayList can grow in size as required. The programmer doesn't have to worry about the length of the ArrayList while adding elements to it.
2. **Type safety** - An ArrayList can be made type safe using generics.
3. **Readymade features** - ArrayList provides methods for searching and for inserting elements anywhere in the list.

Disadvantages of ArrayList

1. **Higher memory usage** - An ArrayList generally requires more space than is necessary to hold the same number of elements in an array.
2. **No type safety** - Without generics, an ArrayList is not type safe at all.
3. **No support for primitive values** - ArrayLists cannot store primitive values while arrays can. This disadvantage has been mitigated somewhat with the introduction of autoboxing in Java 5, which makes it possible to pass primitive values to various methods of an ArrayList. However, autoboxing does impact performance.

Similarities between ArrayLists and arrays

1. **Ordering** - Both maintain the order of their elements (meaning, an element stored at a given position aka index can be accessed by specifying that position) and allow retrieval of elements based on an index, which ranges from 0 (inclusive) to `length` (exclusive). Since elements in an array and an ArrayList can be accessed through an index, both can be traversed in either direction.
2. **Duplicates** - Both allow duplicate elements to be stored.
3. **nulls** - Both allows nulls to be stored.
4. **Performance** - Since an ArrayList is backed by an array internally, there is no difference in performance of various operations such as searching on an ArrayList and on an array.
5. **Thread safety** - Neither of them are thread safe. Don't worry, thread safety is not on the exam, but you should be aware of the fact that accessing either of them from multiple threads, may produce incorrect results in certain situations.

9.6 Exercise ↗

1. Create a array of booleans of length 3 inside the main method. Print the elements of the array without initializing the array elements explicitly. Observe the output.
2. Given `int[] first = new int[3];, int[] second = {};`, and `int[] third = null;`, print out the length of the three arrays and print out every element of the three arrays.
3. Create an array of chars containing four values. Write assignment statements involving the array such that the first element of the array will contain the value of the second element, second element will have the value that was there in the third element. and third element will contain the value of the fourth element.
4. Declare and initialize an array of length 4 of type array of Strings without using the new keyword such that no two arrays of Strings have the same length. Print the length of all of the arrays one by one (including the length of the two dimensional array).
5. Define a simple class named `Data` with a public instance field named `value` of type `int`. Create and initialize a `Data` variable named `d` in `TestClass`'s main. Create an array of `Data` of length 3 and initialize each of its elements with the same `Data` instance. Use any of the array elements to update the `value` field of the `Data` object. Print out the `value` field of the `Data` object using the three elements of the array. Finally, print the `value` field of the original `Data` using the variable `d`.
6. Create an array of `ints` and an `ArrayList` of Integers. Add alternate elements of the array to the list. Remove all even values from the list.
7. Create an `ArrayList` with some values. Print the first and the last element of the list. Insert a new value in the middle of the list.
8. Create an `ArrayList` and add all the arguments given on the command line to that list. Search for a particular argument in the list and print its index. Check if a particular argument is present in the list and if present, remove it from the list.

Exam Objectives

1. Create a new class including a main method (covered in the "Basic Java Elements" chapter)
2. Use the private modifier
3. Describe the relationship between an object and its members
4. Describe the difference between a class variable, an instance variable, and a local variable
5. Develop code that creates an object's default constructor and modifies the object's fields
6. Use constructors with and without parameters
7. Develop code that overloads constructors

10.1 Relationship between an object and its members

10.1.1 Accessing object fields

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

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

For example, consider the following code:

```
class Student{
    String name;
}

public class TestClass{
    public static void main(String[] args){
        Student s1 = new Student();
        Student s2 = new Student();

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

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

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

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

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

```

class Calculator{
    public int calculate(int[] iArray){
        int sum = 0;
        for(int i : iArray){ //this is a for-each loop, we'll cover it later
            sum = sum+i;
        }
        return sum;
    }
}

public class TestClass {
    public static void main(String[] args) {
        int result = new Calculator().calculate( new int[]{1, 2, 3, 4, 5} );
        System.out.println(result);
    }
}

```

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

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

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

10.1.2 What is "this"?

Let us modify our `Student` class a bit:

```

class Student{
    String name;

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

    public void printName(){
        System.out.println(name); //5
    }
}

```

```
}
```

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

Observe that at //3, we are calling the `printName` method using the reference variable `s1`. Therefore, when the JVM invokes this method, it already knows the instance on which it is invoking the method. It is the same instance that is being pointed to by `s1`. Now, in Java, if you don't specify any reference variable explicitly within any instance method, the JVM assumes that you mean to access the same object for which the method has been invoked. Thus, within the `printName` method, the JVM determines that it needs to access the `name` field of the same `Student` instance for which `printName` method has been invoked. You can also explicitly use this reference to the same object by using the keyword "`this`". For example, //5 can be written as: `System.out.println(this.name);`

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

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

When is "this" necessary?

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

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

In the `setName` method, four variables are accessible: two method parameters - `code` and `name`, and two instance variables - `id` and `name`. Now, within the method code, when you do `id = code;` the compiler knows that you are assigning the value of the method parameter `code` to the instance field `id` because these names refer to exactly one variable each. But when you do `name = name;`, the compiler cannot distinguish between the two `name` variables. It thinks that `name` refers to the method parameter and assigns the value of the method parameter to itself, which is

basically redundant and is not what you want. There is nothing wrong with it from the compiler's perspective but from a logical perspective, the above code has a serious bug. In technical terms, this is called "**shadowing**". A variable defined in a method (i.e. either in parameter list or as a local variable) shadows an instance or a static field of that class. It is not possible to access the shadowed variable directly using a simple name. The compiler needs more information from the programmer to disambiguate the name.

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

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

```
class Student{  
    static int count = 0;  
    public void doSomething(){  
        int count = 10;  
        count = count; //technically valid but logically incorrect  
        Student.count = count; //works fine in instance as well as static methods  
        this.count = count; //works fine in an instance method  
    }  
}
```

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

Exam Tip

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

Here are a few quick facts about `this`:

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

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

10.2 Apply access modifiers

10.2.1 Accessibility

One of the objectives of object-oriented development is to encourage the users of a component (which could be a class, interface, or an enum) to rely only on the agreed upon contract between the user and the developer of the component and not on any other information that the component is not willing to share.

For example, if a component provides a method to compute taxes on the items in a shopping cart, then the user of that component should only pass the required arguments and get the result. It should not try to access internal variables or logic of that class because using such information will tie the user of that component too tightly to that component. Imagine a developer writes the following code for the `TaxCalculator` class:

```
class TaxCalculator{
    double rate = 0.1;
    double getTaxAmount(double price){
        return rate*price;
    }
}
```

Ideally, users of the above class should use the `getTaxAmount` method but let us say they do not and access the `rate` variable instead, like this:

```
//code in some other class
double price = 95.0;
TaxCalculator tc = new TaxCalculator();
double taxAmt = price*tc.rate;
```

Later on, the developer realizes that "tax rate" cannot be hardcoded to 0.1. It needs to be retrieved from the database and so, the developer makes the following changes to the class:

```
class TaxCalculator{
    //double rate = 0.1; //no more hardcoding
    double getTaxAmount(double price){
        return getRateFromDB()*price;
    }

    double getRateFromDB(){
```

```
//fetch rate from db using jdbc
}
}
```

Since there is no `rate` variable present in this class anymore, all other code that is accessing this variable will now fail to compile. Had they stuck to using the `getTaxAmount` method, they would not have had any problem at all. To prevent compilation failure, the developer will now be forced to maintain the `rate` variable in their new code even though this variable is not required by the class anymore. What if the tax rate changes in the database but is not updated in the `TaxCalculator` class's `rate` variable? What if one rogue user updates the rate variable at an inopportune time while another user is using it to compute taxes? Well, in both the cases the users will be computing taxes incorrectly. There will be no error message to make the developer aware of the problem either. This is a serious matter.

But the problem is not just with the users relying on internal details of a class. The `TaxCalculator` class doesn't give any clue as to what features it supports. How are the users supposed to know that they should be using only the `getTaxAmount` method and not the `rate` variable? In other words, the `TaxCalculator` class doesn't make its public contract clear and therefore, it would be unfair to blame only the users of this class. This is where Java's **access modifiers** come into picture.

10.2.2 Access modifiers

Java allows a class and members of a class to explicitly specify who can access the class and the members using three accessibility modifiers: **private**, **protected**, and **public**. Besides these three, the absence of any access modifier is also considered an access modifier. This is known as "**default**" access. These access modifiers make your intention about a member very clear not only to the users of your class but also to the compiler. The compiler then helps you enforce your intention by refusing to compile code that violates your intention. Here is how these modifiers affect accessibility:

private - A private member is only accessible from within that class. It cannot be accessed by code in any other class.

For example, the problem that I showed you with the `TaxCalculator` class could have been easily avoided if the developer had simply declared the `rate` variable as private. That would make the variable inaccessible from any other class. The compiler would prevent the users of this class from using the `rate` variable by refusing to compile the code that tried to use it. Since there would have been no dependency on `rate` variable, the developer could have easily removed this variable without any impact on anyone.

"default" - A member that has no access modifier applied to it is accessible to all classes that belong to the same package. This is irrespective of whether the class trying to access a default member of another class is a sub class or a super class of the other class. If two classes belong to the same package, then they can access each other's default members. This is also called "**package private**" or simply "**package access**".

protected - A protected member is accessible from two places - if the accessing class belongs to the same package or if the accessing class is a subclass irrespective of the package to which the subclass belongs. The first case is simple because it is exactly the same as default access. All classes

belonging to the same package can access each other's protected members. The second case is not so, simple and requires a deeper understanding of the concepts of inheritance and polymorphism, which are not covered in this exam.

public - A public member is accessible from everywhere. Any code from any class can make use of a public member of another class. For example, the `getTaxAmount` method could have been made public. That would give a clear signal to the users to use this method if they want to compute the tax amount.

If you order the four access modifiers in terms of how restrictive they are, then **private** is **most restrictive** and **public** is **least restrictive**. The other two, i.e., **default** and **protected**, lie between these two. Thus, the order of access modifiers from the most restrictive to the least would be: **private > default > protected > public**.

Exam Tip

In the exam, watch out for non-existent access modifiers such as "friend", "private protected", and "default".

10.3 Apply encapsulation principles to a class

10.3.1 Encapsulation

Encapsulation is considered as one of the three pillars of Object-Oriented Programming. The other two being **Inheritance** and **Polymorphism**. In the programming world, the word **encapsulation** may either refer to a language mechanism for restricting direct access to an object's data fields **or** it may refer to the features of a programming language that facilitate the bundling of data with the methods operating on that data. For the purpose of the exam, we will be focusing on the first meaning, i.e., restricting direct access to an object's fields.

As I explained in the previous section on **access modifiers**, letting other classes directly access instance variables of a class causes **tight coupling** between classes and that reduces the maintainability of the code. While designing a class, your goal should be to present the functionality of this class through methods and not through variables. In other words, a user of your class should be able to make use of your class by invoking methods and not by accessing variables. There are two advantages of this approach:

1. You give yourself the freedom to modify the implementation of the functionality without affecting the users of your class. In fact, users of a well encapsulated class do not even become aware of the internal variables that the class uses for delivering that functionality because the implementation details of the functionality are hidden from the users. In that sense, **encapsulation** and **information hiding** go hand in hand.
2. You can ensure that the value of a variable is always consistent with the business logic of the class. For example, it wouldn't make sense for the `age` variable of a `Person` class to have a negative value. If you make this variable public, anyone could mess with `Person` objects by setting their ages to a negative value. It would therefore, be better if the `Person` class has a public setter method instead, like this:

```
class Person{
    private int age;

    public void setAge(int yrs){
        if(yrs<0) throw new IllegalArgumentException();
        else this.age = yrs;
    }

    public int getAge(){ return age; }

    //other code
}
```

Using access modifiers in professional code

Since the only way to restrict access to the variables of a class from other classes is to make them private, a well encapsulated class defines its variables as private. The more you relax the access restrictions on the variables, the less encapsulated a class gets. Thus, a class with public fields is not encapsulated at all. The visibility of the methods, on other hand, can be anything ranging from private to public, depending on the business purpose of the class.

In your code, you should always act miserly while giving access rights to other classes, i.e., always try to make your class members as restricted (private) as possible.

Accessibility of accessor methods

There is almost never a need to make data fields non-private. If you want other classes to access the data members of your class, you should provide non-private methods that get and set values of those variable. For example in the above code, the `age` field of Person class is private. The `getAge` and `setAge` are accessor methods and are public.

Make methods non-private only if you are sure that other classes need to access them. Be very careful while make making classes and method public. As explained before, making something public means that you want to allow any class from any application to access and thus, depend on your class and you don't want to create any unnecessary dependencies.

Encapsulation of static members

Encapsulation is an OOP concept and generally applies to the instance members of a class and not to the static members. However, if you understand the spirit of encapsulation, you will notice that all we want to do is to prevent others from inadvertently or incorrectly accessing stuff that they should not be accessing. This is true of static variables also. Thus, it is better to have even the static variables as private. Ideally, the only variables that deserve to be public are the ones that are constants.

10.4 Create and overload constructors

10.4.1 Creating instance initializers

When you ask the JVM to create a new instance of a class, the JVM does four things:

1. First, it checks whether that class has been initialized or not. If not, the JVM loads and initializes the class first.
2. Second, it allocates the memory required to hold the instance variables of the class in the heap space.
3. Third, it initializes these instance variables to their default values (i.e. numeric and char variables to **zero**, boolean variables to **false**, and reference variables to **null**).
4. And finally, the JVM gives that instance an opportunity to set the values of the instance variables as per the business logic of that class by executing code written in special sections of that class. These special sections are: **instance initializers** and **constructors**.

It is only after these four steps are complete that the instance is considered "ready to use". Remember the use of the **new** operator to create instances of a class? The JVM performs all of the four activities mentioned above and only then returns the reference of the newly created and initialized object to your code.

Out of the four activities listed above, you have already seen the details of the first one in the "Declare and initialize variables" section of "Working with Java Data Types" chapter. The second two activities are performed transparently by the JVM. They don't require the programmer to do anything. The fourth activity, which is the subject of this section, depends on the programmer because it involves the code written by the programmer.

Although **instance initializers** are not on the exam, let us take a brief look at them anyway because these are the ones that are executed by the JVM first.

Creating instance initializers

Instance initializers are blocks of code written directly within the scope of a class. Here is an example:

```
class TestClass{  
    {  
        System.out.println("In instance initializer");  
    }  
}
```

Observe that there is no method declaration or anything but just a line of code nested inside the opening and closing curly braces. Code inside an instance initializer block is regular code. There is

no limitation on the number of statements or the kind of statements that an instance initialize can have. You may have any number of such instance initializer blocks in a class. The JVM executes them in the order that they appear in the class. The following code, for example, prints **Hello World!** using two instance initializers in different places in a class:

```
class TestClass{  
  
    //first instance initializer  
    {  
        System.out.print("Hello ");  
    }  
  
    public static void main(String[] args){  
        new TestClass();  
    }  
  
    //second instance initializer  
    {  
        System.out.print("World!");  
    }  
}
```

Observe that the main method does nothing except create an instance of TestClass. As a result of this creation, the JVM executes each of the two instance initializers and then returns the newly created instance to the main method. Of course, the main method does not assign the reference of the newly created instance to any variable but that is okay.

Accessing members from instance initializers

Instance initializers have access to all of the members of a class. This includes static as well as instance fields and methods. Just like the instance methods, instance initializers have access to the implicit variable "**this**".

10.4.2 Creating constructors

Constructor of a class

A constructor of a class looks very much like a method of a class but there are two things that make a method a constructor:

1. **Name** - The name of a constructor is always exactly the same as the name of the class.
2. **Return type** - A constructor does not have a return type. It cannot even say that it returns void.

The following is an example of a class with a constructor.

```
class TestClass{

    int someValue;
    String someStr;

    TestClass(int x) //No return type specified
    {
        this.someValue = x; //initializing someValue
        //return; //legal but not required
    }
}
```

Observe that the name of the constructor is the same as that of the class and that there is no return type in the declaration of the constructor. Also observe that there is no return statement because a constructor cannot return anything, not even void. However, it is permissible to write an empty return statement as shown in the code above.

It is interesting to note that a class can have a method with the same name as that of the class. For example, consider the following class:

```
class TestClass{

    int someValue;
    String someStr;

    void TestClass(int x) //<-- observe the return type void
    {
        this.someValue = x;
    }
}
```

In the above code, `void TestClass(int x)` is not a constructor but it is a valid method nevertheless and so the code will compile fine.

Besides the above two rules there are several other rules associated with constructors. Some of them are related to inheritance and exception handling and I will cover them later. Here, I will talk about rules that are applicable to constructors in general.

The default constructor

If I tell you that every class must have a constructor, you might have trouble believing it because so far, I have shown you so many classes that had no constructor at all! Well, here is the deal - it is true that every class must have at least one constructor but the thing is that the programmer doesn't necessarily have to provide one. If the programmer doesn't provide any constructor, then the compiler will add a constructor to the class on its own. This constructor, that is, the one provided by the compiler, is called the "**default**" constructor. This default constructor takes no argument and has no code in its body. In other words, it does absolutely nothing. For example, if I write and compile this class, `class Account { }`, the compiler will automatically add a constructor to this class which looks like this:

```
class Account{  
    Account(){ } //default constructor added by the compiler  
}
```

This sounds simple but the cause of confusion is the fact that if you write any constructor in a class yourself, the compiler will not provide the default constructor at all. So, for example, consider the following class:

```
class Account{  
    int id;  
    Account(int id){  
        this.id = id;  
    }  
    public static void main(String[] args) {  
        Account a = new Account(); //trying to use the no-args constructor  
    }  
}
```

I have provided a constructor explicitly in the above class. Can you guess what will happen if I try to compile this class? It will not compile. The compiler will complain that `Account` class does not have a constructor that takes no arguments. What happened to the default constructor, you ask? Well, since this class provides a constructor explicitly, the compiler feels no need to add one on its own.

The discussion on default constructors also gives me an opportunity to talk about one misconception that I have often heard, which is that constructors must initialize all instance members of the class. This is not true. A constructor is provided by you, the programmer, and you can decide which instance members you want to initialize. For example, the constructor I showed earlier for `TestClass` doesn't touch the `someStr` variable. The default constructor provided by the compiler also doesn't assign any values to the instance members. Remember that the JVM always provides default values to static and instance members anyway.

Exam Tip

You will most certainly get a question that tests your knowledge about the default constructor. Watch out for code that assumes the existence of the default constructor while the class provides a constructor explicitly.

Difference between default and user defined constructors

There are only two things that you need to remember about a default constructor:

1. **When is it provided** - It is provided by the compiler **only** when a class does not define **any** constructor explicitly.
2. **What does it look like** - The default constructor is the simplest constructor that you will ever see. It does not take any argument, does not have any throws clause, and does not

contain any code.

But one peculiar thing about the default constructor is its accessibility. The accessibility of the default constructor is the same as that of the class. That means, if the class is public, the default constructor will also be public and if the class has default accessibility, the default constructor will also have default accessibility. If you have no idea about accessibility, don't worry, I will talk about it in the next topic.

The default constructor is also sometimes called the "**default no-args**" constructor because it does not take any arguments. But this is technically imprecise because the default constructor is always a no-args constructor. There is nothing like a default constructor with args!

You can, on the other hand, write a no-args constructor explicitly in a class with a throws clause and with a different accessibility. For example, it is quite common to have a class with private no-args constructor if you don't want anyone to create instances of that class. I will talk more about it in the next topic.

Benefit of constructors

Anything that you can do in a method, you can do in a constructor and vice-versa. There is no limitation on what a method or a constructor can do. Why not just have a method and name it as init() or something? Indeed, you will encounter Java frameworks such as servlets that do exactly that. Then why do you need a constructor? Well, a detailed discussion of the pros and cons of constructors is beyond the scope of this book, but I will give you a couple of pointers.

As I mentioned earlier, an instance is not considered "ready to use" until its constructor finishes execution. Thus, a constructor helps you make sure that an instance of your class will be initialized as per your needs before anyone can use it. Here, "use" essentially means other code accessing that instance through its fields or methods.

It is possible to do the same initialization in a method instead of a constructor but in that case the users of your class would have to remember to invoke that method explicitly after creating an instance. What if a user of your class fails to invoke that special method after creating an instance of your class? The instance may be in a logically inconsistent state and may produce incorrect results when the user invokes other methods later on that instance. Therefore, a constructor is the right place to perform all initialization activities of an object. It is a place where you make sure that the instance is ready with all that it needs to perform the activities it is supposed to perform in other methods.

Constructors also provide thread safety because no thread can access the object until the constructor is finished. This protection is guaranteed by the JVM to the constructors and is not always available to methods.

10.4.3 Overloading constructors

A class can have any number of constructors as long as they have different signatures. Since the name of a constructor is always the same as that of the class, the only way you can have multiple constructors is if their parameter type list is different.

Just like methods with same names, if a class has more than one constructor, then this is called "**constructor overloading**" because the constructors are different only in their list of parameter types.

There are a couple of differences between method overloading and constructor overloading though. Recall that in case of method overloading, a method can call another method with the same name just like it calls any other method, i.e., by using the method name and passing the arguments in parenthesis. In case of constructor overloading, when a constructor calls another constructor of the same class, it is called "**constructor chaining**" and it works a bit differently. Let me show you how.

Constructor chaining

A constructor can invoke another constructor using the keyword **this** and the arguments in parenthesis. Here is an example:

```
class Account{  
    int id;  
    String name;  
    Account(String name){  
        this(111, name); //invoking another constructor here  
        System.out.println("returned from two args constructor");  
    }  
    Account(int id, String name){  
        this.id = id;  
        this.name = name;  
    }  
    public static void main(String[] args) {  
        Account a = new Account("amy");  
    }  
}
```

Observe that instead of using the name of the constructor, the code uses 'this'. That is, instead of calling `Account(111, name);`, the code calls `this(111, name);` to invoke the other constructor. It is in fact against the rules to try to invoke another constructor using the constructor name and it will result in a compilation error.

Invoking another constructor from a constructor is a common technique that is used to initialize an instance with different number of arguments. As shown in the above example, the constructor with one argument calls the constructor with two arguments. It passes one user supplied value and one default value to the second constructor. This helps in keeping all initialization logic in one constructor, while allowing the user of the class to create instances with different parameters.

The only restriction on the call to the other constructor is that it must be the first line of code in a constructor. This implies that a constructor can invoke another constructor only once at the most. Thus, the following three code snippets for constructors of `Account` class will not

compile:

```
Account(String name){  
    System.out.println("calling two args constructor");  
    this(111, name); //call to another constructor must be the first line  
}  
  
Account(){  
    this(111);  
    this(111, "amy"); //call to another constructor must be the first line  
}  
  
Account(String name){  
    Account(111, name); //incorrect way to call to another constructor but if the  
    // Account class had a method named Account, this would be a valid call to that method  
}
```

Invoking a constructor

It is not possible to invoke the constructor of a class directly. It is invoked only as a result of creation of a new instance using the "new" keyword. The only exception to this rule is when a constructor invokes another constructor through the use of "this" (and "super", which I will talk about in another chapter) as explained above. In other words, if you are given a reference to an object, you cannot use that reference to invoke the constructor on that object.

Now, consider the following program. Can you tell which line out of the four lines marked LINE A, LINE B, LINE C, and LINE D should be uncommented so that it will print **111, dummy?**

```
class Account{  
    int id;  
    String name;  
    public Account(){  
        id = 111;  
        name = "dummy";  
    }  
  
    public void reset(){  
        //this(); //<-- LINE A  
        //Account(); //<-- LINE B  
        //this = new Account(); //<-- LINE C  
        //new Account(); //<-- LINE D  
    }  
  
    public static void main(String[] args) {  
        Account a = new Account();  
        a.id = 2;  
        a.name = "amy";  
        a.reset();  
    }  
}
```

```
    System.out.println(a.id+" "+a.name);
}

}
```

The answer is none of these. First three are invalid attempts to invoke the constructor - `this()` can only be used within a constructor, `Account()` is interpreted as method call to a method named `Account`, but such a method doesn't exist in the given code, and `this = new Account()` attempts to change "`this`", which is a final variable, to point to another object. Thus, none of these three statements will compile.

`new Account()` is a valid statement but it creates an entirely new `Account` object. The instance variables of this new `Account` object are indeed set to `111` and `"dummy"`, but this doesn't change the values of the current `Account` object.

10.5 Understanding the scopes of class, instance, and local variables

10.5.1 Scope of variables

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

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

10.5.2 Scope and Visibility

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

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

At the same time, you can certainly have two presidents **in** a country - the President of the country, and the President of a basketball association within that country! If you are in your basketball association meeting, and if you talk about the president in that meeting, it will be interpreted as person who is the president of the association and not the person who is the president of your country. But if you do want to mean the president of the country, you will have to clearly say something like the "President of our country". Here, "of our country" is the referencing mechanism that removes the ambiguity from the word "president".

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

This is exactly how "scope" in Java (or any other programming language, for that matter) works. For example, if you declare a static variable in a class, the **visibility** of that variable is the class in which you have defined it. The visibility of an instance variable is also the class in which it is

defined. Since both have same visibility, you cannot have a static variable as well as an instance variable with the same name in the same class. It would be like having two presidents of a country. It would be absurd and therefore, invalid.

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

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

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

For example, consider the following code:

```
public class Area{
    public static String UNIT="sq mt"; //UNIT is visible all over inside the class Area
    public void printUnit(){
        System.out.println(UNIT); //will print "sq mt" because UNIT is visible here
    }
}

public class Volume{
    //Area's UNIT is accessible in this class but not visible to the compiler directly
    public static String UNIT="cu mt";

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

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

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

10.5.3 Scope and Lifespan

Scope and Lifespan

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

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

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

```
public class TestClass
{
    public static void main(String[] args){
        {
            int i = 0; //i exists in this block only
            System.out.println(i); //OK
        }
        System.out.println(i); //NOT OK because i has already gone out of scope
    }
}
```

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

```
public class TestClass
{
    public static void main(String[] args){
        for(int i = 0; i<10; i++)
        {
            int k = 0; //k is block scoped. It is reset to 0 in each iteration
            System.out.println(i); // i retains its value from previous iteration
        }
    }
}
```

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

}
```

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

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

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

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

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

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

10.5.4 Scopes Illustrated ↗

The following code shows various scopes in action.

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

    public static void method1(int param1){ //visible throughout the method
        int local1 = 0; //visible throughout the method

        {
            int anonymousBlock = 0; //visible in this block only
        }

        anonymousBlock = 1; //compilation error

        for(int loop1=0; loop1<10; loop1++){
    }
```

```

        int loop2 = 0;
        //loop1 and loop2 are visible only here
    }
    loop1 = 0; //compilation error
    loop2 = 0; //compilation error
    if(local1==0){
        int block1 = 0; //visible only in this if block
    }

    block1 = 7; //compilation error

    switch(param1){
        case 0:
            int block2 = 10; //visible all over case block
            break;
        case 1:
            block2 = 5; //valid
            break;
        default:
            System.out.println(block2); //block2 is visible here but compilation
            error because block2 may be left uninitialized before access
    }
    block2 = 9; //compilation error
    int loop1 = 0, loop2 = 0, block1 = 0, block2 = 8; //all valid
}
}

```

10.5.5 Scope for the Exam ✎

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

```

class Person{
    private String name; //class scope

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

    public static void main(String[] args){
        for(int i = 0; i<10; i++){
            String name = "john"; //OK. name is scoped only within this for loop block
        }
        String name = "bob"; //OK. name is method scoped
        System.out.println(name); //will print bob
    }
}

```

```
}
```

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

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

```
class Person{
    private String name; //name is class scoped

    public static void main(String[] args){
        String name = "bob"; //method scope. OK. Overlaps with the instance field name
        defined in the class
        int i = -1; //method scope

        for(int i = 0; i<10; i++){ //i has for loop scope. Not OK
            String name = "john"; //block scope. Not OK.
        }

        { //starting a new block here
            int i = 2; //block scope. Not OK.
        }
    }
}
```

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

10.5.6 Quiz

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

```
public class TestClass {
    public static void main(String[] args) {

        {
            int x = 10;
        }
        System.out.println(x);
    }
}
```

Select 1 correct option

- A. It will not compile.
- B. It will print 10

C. It will print an unknown number

The correct answer is A.

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

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

```
java ScopeTest hello world
```

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

Select 1 correct option

A. It will not compile.

B. It will print:

```
null  
null  
null
```

C. It will print:

```
null  
null  
hello
```

Answer is B.

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

The original string array passed to the program is lost.

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

10.6 Exercise ↗

1. Define a reference type named `Bird`. Define an instance method named `fly` in `Bird`. Define a few instance as well as static variables of type `int`, `float`, `double`, `boolean`, and `String` in `Bird`.
2. Create a `TestClass` that has a static variable of type `Bird`. Initialize this variable with a valid `Bird` object. Print out the default values of static and instance variables of `Bird` from the `main` method of `TestClass`. Also print out the static variable of `TestClass` from `main`. Observe the output.
3. Create and initialize one more instance variable of type `Bird` in `TestClass`. Assign values to the members of the `Bird` instance pointed to by this instance variable in `TestClass`'s `main`. Assign values to the members of first `Bird` using the second `Bird`. Print the values of the members of both the `Bird` objects.
4. Write code in `fly` method to print out the values of all members of `Bird`. Alter `main` method of `TestClass` to invoke `fly` on both the instances of `Bird`. Observe the values printed for static variables of `Bird`.
5. Add an instance variable of type `Bird` in `Bird`. Initialize this variable on the same line using "`new Bird()`" syntax. Instantiate a `Bird` object in `TestClass`'s `main` and execute it. Observe the output.
6. Remove the initialization part of the variable that you added to `Bird` in previous exercise. Initialize it with a new `Bird` object in `TestClass`'s `main`. Identify how many `Bird` objects will be garbage collected when the `main` method ends.
7. Add a parameter of type `Float` to `Bird`'s `fly` method. Return an `int` value from `fly` by casting the method parameter to `int`. Invoke `fly` multiple times from `TestClass`'s `main` by passing a `float` literal, a `Float` object, a `double` literal, an `int`, an `Integer`, and a `String` containing a `float` value. Observe which calls compile.
8. Assign the return value of `fly` to an `int` variable, a `float` variable, a `String` variable, and `boolean` variable. Observe which assignments compile. Try the same assignments with an explicit cast. Print these variables out and observe the output.

Exam Objectives

1. Describe and create a method
2. Create and use accessor and mutator methods
3. Create overloaded methods
4. Describe a static method and demonstrate its use within a program

11.1 Create methods with arguments and return values

11.1.1 Creating a method

We have already seen methods in previous chapters. Indeed, we have been using the "main" method to run our test programs all along. But we haven't discussed them formally yet.

A **method** is what gives **behavior** to a type. Java allows classes, interfaces, and enums to have a behavior. A behavior is nothing but a high level action performed by a piece of code. If you expect this action to be performed as and when required, you put this piece of code in a method and give it a name. This lets you invoke that action whenever you need by using the name. It is like a black box to which you give some input and get back some output in return.

Thus, the basic structure of a method is as follows:

```
returnType methodName(parameters) {  
    methodBody  
}
```

A **ReturnType** specifies what the method returns as a result of its execution. For example, a method that returns the sum of two integers may specify that it returns an `int` as a result. A method that returns an `Account` object may specify `Account` as the return type. If a method doesn't return anything, it must specify so, using the keyword `void`.

It is important to know that a method can return one thing at the most.

A **MethodName** is the name given to this method. It must be a valid Java identifier. Remember that an identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter. It cannot be a reserved word or a keyword.

The **Parameters** of a method are specified by a list of comma-separated parameter declarations. Each parameter declaration consists of a type and an identifier that specifies the name of the parameter. If a method does not take any parameter, the parameters list will be empty. The following are a few examples of a valid parameter list:

```
void save() //no parameters  
void saveAccount(Account acct) //takes an Account as a parameter  
void add(int a, int b) //takes two ints as parameters
```

Note that, unlike regular declarations, each parameter in a parameter list must be specified along with its type individually. Thus, while `int a, b;` is a valid statement on its own that declares two `int` variables, `add(int a, b)` is an invalid declaration of method parameter `b`.

Each parameter in the parameter list may also be declared as final if the code in the method does not change its value. For example:

```
void add(int a, final int b) //b is final
```

If a parameter is declared final and if the method body tries to change its value, the code will fail to compile.

The **MethodBody** contains the code that is to be executed upon invocation of that method. It must be contained within curly braces. Observe that this is unlike other block statements such as if/else, for, and while, where you can omit the curly braces if there is just one statement in their body. In the case of methods, curly braces are required even if the body consists of just one statement.

There are several other bells and whistles associated with methods such as var-args, accessibility, static, abstract, final, and exception handling. I will discuss all of these as we go along.

11.1.2 Returning a value from a method

A method must always return a value of the type that it promises to return in its declaration after successful completion. This means that it is not possible to return a value conditionally as shown in the following code:

```
public int get2X(int x){  
    if(x>0) return 2*x;  
}
```

The compiler will generate an error message saying, "error: missing return statement" because it notices that the method will not return anything if the if condition is false. This makes sense if you consider what will happen to the caller of the get2X method if the method doesn't return anything. For example, what value should be assigned to `y` by this statement - `int y = get2X(-1);`? There is no good answer. Thus, the method must return an `int` in all situations (except when it throws an exception, but you can ignore that for now).

This rule applies even to methods that return a reference type (and not just to method that return a primitive). But if the return type of a method is a reference type, it is ok for the method to return `null` because `null` is a valid value for a reference. In other words, returning `null` is not the same as returning nothing. Thus, the following method is fine:

```
String getValue(int x){  
    if(x > 0) return "good day!";  
    else return null; //this is ok  
}
```

The only situation where a method can avoid returning a value is if it ends up throwing an exception, which means that the method didn't really finish successfully and therefore, it cannot be expected to return a value!

Method returning void

If a method says that it doesn't return anything (i.e. its return type is specified as `void`), then it must not return anything in any situation. Obviously, it cannot even return `null` because as we

saw above, `null` is not the same as nothing. But the interesting thing is that it cannot even return `void`. Thus, the following method will not compile.

```
void doSomething(){
    System.out.println("hello");
    return void; //invalid
}
```

There are only two options in this case - do not have any return statement at all or have an empty return statement, i.e., `return;` For example:

```
void doSomething(){
    System.out.println("hello");
    return; //empty return
}
```

or

```
void doSomething(){
    System.out.println("hello");
    //no return statement at all
}
```

Returning values of different types from a method

The general rule is that Java does not allow a method to return a value that is of a different type than the one specified in its declaration. This means that if a method says that it returns an `int`, it cannot then return a `boolean` value. However, there are three exceptions to this rule. Two are about primitives and one is about references.

- Numeric promotion** - If the return type of a method is a numeric type (i.e. `byte`, `char`, `short`, `int`, `long`, `float`, and `double`), then the return value can be of any other numeric type as long as the type of the return value is smaller than the type of the declared return type. For example, if a method says it returns an `int`, it is ok for that method to return a `byte`, `short`, or `char` value. But it cannot return a `long`, `float`, or `double` value. Thus, the following code will compile fine:

```
public int getVal(int x){
    char ch = 'a';
    byte b = 0;
    if(x>0) return ch;
    else return b;
}
```

This is allowed because a smaller type can easily be promoted to a larger type without any loss of information.

- Autoboxing/Unboxing** - Java allows a return value to be a reference to a wrapper class if the return type is of a primitive type of the same or a larger type. Thus, the following code is ok:

```
public int getVal(){
    return new Short((short)10); //wrapper object will be unboxed into a
    primitive
    //return new Long(10); //will not compile, Long cannot be converted to int
}
```

The reverse is also allowed but only when the method's return type is a wrapper of the same type as the type of the primitive value being returned:

```
public Integer getVal(){
    byte b = 10;
    return 10; //ok, int 10 will be boxed into an Integer object
    //return b; //will not compile, byte cannot be converted to Integer
}
```

3. The third exception is related to inheritance. Since I haven't discussed this topic yet, I will only mention it briefly here. Don't worry if you don't understand it completely at this point. It is ok for a method to return a reference of a subtype of the type declared as its return type. This is called "covariant return types." For example, if a method declares that it returns an Object, it is ok for the method to return a String because a String is also an Object. The following code shows some valid possibilities:

```
Object getValue(){
    return "hello";
//return 10; //This is ok, 10 will be boxed into an Integer object, which is an
//Object
//return null; //This is ok too
}
```

This is the same as promising someone that you will give them a fruit, and then give them a banana. This is ok because a banana is also a type of fruit. But the reverse is not true. If you promise someone that you will give them a banana, you cannot then give any other fruit. Thus, the following will fail to compile:

```
String getValue(){
    return new Object(); // will not compile because an Object is not a String
}
```

It will become clearer after you learn about inheritance.

These rules ensure that only those values that are "assignable" to the declared return type are returned by a method. In other words, if you can assign a particular value to a variable of the declared return type directly, then you can return that value from that method. For example, you can assign a `char` value to an `int` variable directly, therefore, you can return a `char` value from a method that declares that it returns an `int`.

Returning multiple values from a method

Java does not allow a method to return more than one value. Period. This seems like difficult restriction to overcome if you want to return multiple values from a method. For example, what if your getName method wants to return first name and last name separately? Well, the way to do that in Java is to use a class to capture multiple values and then return a reference to an object of that class. Here is what your getName method may look like:

```
Name getName(){
    Name n = new Name(); //capture two values in a Name object
    n.firstName = "Ann";
    n.lastName = "Rand";
    return n; //return a reference to the Name object
}
```

This code assumes the existence of a separate class called Name that can capture the two components of the name:

```
class Name{
    String firstName, lastName;
}
```

You may also use arrays to overcome this restriction. For example, the getName method can also be coded as follows:

```
String[] getName(){
    return new String[]{"ann", "rand"};
}
```

Generally, classes are designed to capture values that are related to each other. For example, you might have a Student class that captures a student's id, name, and address. You may, however, encounter situations where you want to return multiple unrelated values from the same method. You can use the same approach to return these values. Classes that are used to capture unrelated values are called holder classes. If you encounter such situations too often in your code then it is a symptom of a bad design. A detailed discussion on this is beyond the scope of the exam.

11.1.3 Varargs

If you know the type of the arguments but don't know the exact number of arguments that the caller is going to pass to a method, you can put the arguments in an array and pass the array to the method. For example, if you want to write a method that computes the average of any number of integer values, you could do something like this:

```
public double average(int[] values){
    /* by the way, can you tell what will happen if sum is declared as int?
       Expect questions in the exam that seem to be about one topic
```

```
    but are actually about something entirely different.  
*/  
double sum = 0;  
for(int i=0; i<values.length; i++) sum += values[i];  
return values.length==0?0 : sum/values.length;  
}
```

The caller of this method can put all the integer values in an array and call it as follows:

```
int[] values = { 1, 2, 3, 4 };  
double average = average(values);
```

This approach works fine but is a little tedious to write. Java 5 introduced a new syntax called "**varargs**" that makes passing a variable number of arguments to a method a little easier. Instead of using an array parameter in the method declaration, you use a varargs parameter by appending three dots to **int** like this:

```
public double average(int... values){  
    //same code as shown earlier goes here  
}
```

There is absolutely no change in the method body. Within the method body, **values** remains an array of integers like before. The caller, however, does not need to create an array explicitly. It can simply pass any number of int arguments in the method invocation. Thus, the following are all valid method invocations of the new average method.

```
double average = average(); //no argument  
double average = average(1); //one argument  
double average = average(1, 2, 3, 4); //multiple arguments
```

An important point to understand here is that the varargs syntax is just a syntactic sugar for the developer. It saves a few keystrokes while typing the code but makes absolutely no difference to the resulting bytecode generated by the compiler. When the compiler sees the invocation of a method with varargs parameter, it simply wraps the arguments into an array and passes the array to the method. Indeed, the old code that used an array to call the average method still works with the new average method. If you update the method code to print the number of elements in the **values** array (just add `System.out.println(values.length);`), you will see it print 0, 1, and 4 for the above three invocations respectively.

Observe that if you don't pass any argument, the compiler will create an array with a length of zero and pass that array. The method will not receive a **null** but an array of length zero in this case. This is unlike the other ways that you invoke the array version of the method. It is not possible to invoke the array version without any argument. If you don't want to pass any value, you will have to pass **null**. Try it out and see what happens.

You can apply the varargs approach to a parameter of any type and not just primitives.

Restrictions on varargs

Let us try to expand the usage of varargs from one parameter to two. What if we define a method as follows:

```
public double test(int... p1, int... p2){
    System.out.println(p1.length+" "+p2.length);
}
```

and call this method like as follows?

```
test(1, 2);
```

This poses a problem for the compiler. It has three equally valid possibilities for initializing `p1` and `p2`. It can create two int arrays containing `{1}` and `{2}`, or it can create `{1, 2}` and `{ }`, or it can create `{ }` and `{ 1, 2 }` and pass them as arguments for `p1` and `p2` respectively.

Resolving this ambiguity through complicated rules would make the varargs feature too confusing to use and so, Java imposes the following two restrictions on varargs:

1. A method cannot have more than one varargs parameter.
2. The varargs parameter, if present, must be the last parameter in the parameter list of a method.

With the above two rules in place, it is easy to understand what will happen if you have a method as follows:

```
void test(int x, int... y){
    //some code here
}
```

and call it as follows:

```
test(1, 2); //x is assigned 1, y is assigned {2}
test(1); //x is assigned 1, y is assigned { }
test(1, 2, 3, 4); //x is assigned 1, y is assigned { 2, 3, 4 }
```

Note that since `x` is a non-varargs parameter, any invocation of the method `test` must include a value for `x`. Therefore, a call to `test()` with no argument will not compile.

11.2 Create overloaded methods

11.2.1 Method signature

Before learning about method overloading, you need to know about something called "method signature".

A **method signature** is kind of an "id" of a method. It uniquely identifies a method in a class. A class may have several methods with the same name but it cannot have more than one method with the same **signature**. When you call a method of a class, you basically tell the compiler which

method you mean to call by mentioning its signature. If a class has multiple methods with the same signature, the compiler will not be able to determine which method you mean and will raise an error.

The question then is: what exactly constitutes a method signature? Simple. Method signature includes just the **method name** and its **ordered list of parameter types**. Nothing else. For example, all of the following method declarations have the same signature:

1. void process(int a, String str);
2. public void process(int value, String name);
3. void process(int a, String str) throws Exception;
4. String process(int a, String str);
5. private int process(int a, String data);
6. static void process(int a, String str);

Observe that in all of the above cases, the method name (i.e. `process`) and the ordered list of parameter types (i.e. `int, String`) are exactly the same and their access types, static/instance types, return types, parameter names, and the throws clauses are all different. However, since these attributes are not part of the method signature, they do not make the methods different. The compiler will consider all of the above methods as having the same signature and will complain if you have any two of the above methods in the same class.

11.2.2 Method overloading

Now that you know about method signature, method overloading is a piece of cake. You know that a class cannot have more than one method with the same signature. But it can certainly have multiple methods with the same name and different parameter types because having different parameter types would make their signatures different. In such a situation where a class has multiple methods with same name, it is said that the class has "**overloaded**" the method name. The method name is overloaded in the sense that there are multiple possibilities associated with that name.

From the compiler's perspective, method overloading is nothing special. Since the compiler cares only about the method signatures, it makes no difference to the compiler whether two methods are different because of a difference in their method names, or their method parameters, or both. To the compiler, overloaded methods are just different methods.

However, method overloading does hold importance for the developer and the users of a class because it can either make the code more intuitive to use or make it totally confusing. For example, you have been using `System.out.println` methods to print out all sorts of values to the console. You are able to use the same method named `println` for printing an `int` as well as a `String` only because there are multiple `println` methods that take different parameter types. Take a look at the JavaDoc API description of `java.io.PrintStream` class to see how many `println` methods it has. Wouldn't you be frustrated if you had to use `printlnByte` to print a `byte`, `printlnShort` to print a `short`, `printlnString` to print a `String` and so on? In this case, method overloading has certainly made your life easier.

Now, consider the following code:

```
public class TestClass{
    static void doSomething(Integer i, short s){
        System.out.println("1");
    }

    static void doSomething(int in, Short s){
        System.out.println("1");
    }

    public static void main(String[] args){
        int b = 10;
        short x = 20;
        doSomething(b, x);
    }
}
```

Any guess on what the above code prints? Don't worry, even experienced programmers will scratch their heads while figuring this one out. The answer is that it will not compile. But not because of the presence of two `doSomething` methods. The methods are fine. They have different signatures. The problem is with the call to `doSomething(b, x)`. The compiler is not able to determine which one of the two `doSomething` methods to use because both of them are equally applicable.

You will not get questions this hard in the exam. I showed you the above code only to illustrate how overloading can be misused to create horrible code.

11.2.3 Method selection

The code that I showed you in the previous section illustrates a problem with overloaded methods. When their parameter lists are not too different, it is very difficult to figure out which of the overloaded methods will be picked up for a method call. If the parameters are too similar, it may even become an impossible task.

Remember that even though the actual method is selected and executed at runtime, it is the compiler's job to validate the code and make sure that a particular method call can be bound unambiguously to a specific method out of several available methods. Since actual objects are created by the JVM at run time, the compiler does not always know the exact type of the object pointed to by a reference variable. Therefore, the compiler considers only the declared type of the variable on which the method is invoked and the declared type of the variables passed as arguments while determining which method should be invoked. Once it determines the method that should be invoked, it binds the method call to that particular method signature. This part is important because if the compiler allowed any method to be called on any type of reference, the call may fail at run time. By making the compiler perform this analysis upfront at compile time, we reduce the chances of coding mistakes getting discovered at runtime. If you have written any JavaScript code, you know what I am talking about. This ties back to the strongly typed nature of the Java language.

At runtime, the JVM picks up the method signature that was selected by the compiler, finds

out the actual object pointed to by the reference variable, and invokes the method on that object. This is the part where inheritance plays a huge role and where the magic of polymorphism happens. I will talk about it in the next chapter but for now, I will just focus on overloading without the complication of inheritance.

While you will not get questions that are as complicated as shown above in the exam, you still need to know a few basic rules to figure out the simple cases. So, here they are:

1. The first rule is that having overloaded methods does not cause a compilation error by itself. In other words, as long as their method signatures are different, the compiler doesn't care whether they are too similar or not. Compilation error occurs only if the compiler is not able to successfully disambiguate a particular method call. Java specifies precise rules that are used by the compiler to disambiguate such method calls.
2. **Exact match** - If the compiler finds a method whose parameter list is an exact match to the argument list of the method call, then it selects that method. For example, consider the following two methods:

```
void processData(Object obj){ }
void processData(String str){ }
```

and the method call `processData(myString);`, where `myString` is a `String` variable. Since `String` is an `Object`, the `String` argument matches the parameter list of both the methods and so, both the methods are capable of accepting the method call. However, `String` is an **exact match** to the declared type of `myString` and so, the compiler will select the second method.

This rule applies to primitives as well. Thus, out of the following two methods, the first method is selected when you call `processData(10);` because 10 is an `int`, which matches exactly to the parameter type of the first method even though the long version of the method is also perfectly capable of accepting the value.

```
void processData(int value){ }
void processData(long value){ }
```

3. **Most specific method** - If more than one method is capable of accepting a method call and none of them is an exact match, the one that is "**most specific**" is chosen by the compiler. For example, consider the following two methods:

```
void processData(Object obj){ }
void processData(CharSequence str){ }
```

and the method call `processData("hello");`. Remember that `String` extends `CharSequence` and `CharSequence` extends `Object`. Thus, a `String` is a `CharSequence` and a `String` is also an `Object`. So here, neither of the methods has a parameter list that is an exact match to the type of the argument but both the methods are capable of accepting a `String`. However, between `Object` and `CharSequence`, `CharSequence` is more specific and so the compiler will select the second method.

It is important to understand what "**most specific**" means. It really just means closer or more similar to the type of the parameter being passed. A `String` is closer to a `CharSequence` than it is to an `Object`. Technically, a subclass (or a subtype) is always more specific than a super class (or supertype). If you have trouble understanding this, remember that `Cat` is more specific than `Animal`, `Apple` is more specific than `Fruit`.

Since primitives are not classes, there is no subclass/superclass kind of relation between them as such but Java does define the subtype relation for them explicitly, which is as follows:

```
double > float > long > int > char  
and  
int > short > byte
```

It means, `float` is a subtype of `double`, `long` is a subtype of `float`, `int` is a subtype of `long`, and `char` is subtype of `int`. And also, `short` is a subtype of `int` and `byte` is a subtype of `short`.

Based on the above, you can easily determine which of the following two methods will be picked if you call `processData(byteVar);` (assume that `byteVar` is declared as `byte`).

```
void processData(int value){ }  
void processData(short value){ }
```

The `short` version will be picked because `short` is a subtype of `int` and is therefore, more specific than an `int`.

Here is another interesting situation that can be explained easily based on the above subtype hierarchy of primitives. What will the following code print when compiled and run?

```
public class TestClass{  
    public void m(char ch){  
        System.out.println("in char");  
    }  
    public static void main(String[] args) {  
        byte b = 10;  
        new TestClass().m(b);  
    }  
}
```

The above code will cause a compilation error saying, "error: incompatible types: possible lossy conversion from byte to char". Observe that `char` is not a supertype of `byte`. Therefore, the method `m(char ch)` is not applicable for the method call `m(b)`. This makes sense because even though `char` is a larger data type than `byte`, `char` cannot store negative values, while `byte` can.

4. **Consider widening before autoboxing** - Since autoboxing only came into existence when Java 5 was released, it is necessary to give higher priority to the primitive versions if the argument can be widened to the method parameter type so that existing code will keep working as before. Therefore, out of the following two methods, the `short` version will be picked instead of the `Byte` version if you call `processData(byte b)`; even though the `Byte` version is an exact match with `byte` after autoboxing.

```
void processData(short value){ }

void processData(Byte value){ }
```

5. **Consider autoboxing before varargs** - This rule mandates that if an argument can be autoboxed into a method parameter type then that method be considered even if a method with varargs of the same type is available. This explains why if you call `processData(10)`; then the `Integer` version (and not the `int...` version) is picked out of the following two.

```
void processData(int... values){ }

void processData(Integer value){ }
```

Let us now try to figure out what happens if you call `processData((byte) 10)`; with the same two methods available.

```
void processData(int... values){ }

void processData(Integer value){ }
```

Just apply the rules one by one:

1. **Applying rule 1** - Requires no application.
2. **Applying rule 2** - There is no method available whose parameter type matches exactly to `byte` because `byte` doesn't match exactly to `int...` i.e. `int[]` or `Integer`.
3. **Applying rule 3** - There is no method available whose parameter type matches to `byte` after widening a `byte` to any of the types a `byte` can be widened to (i.e. `short`, `int`, `long` etc.). Note that a `byte` cannot be widened to `int[]`.
4. **Applying rule 4** - There is no method available whose parameter type matches to `byte` after autoboxing it to `Byte`. Remember that a primitive type can only be autoboxed to the same wrapper type. So a `byte` can only be autoboxed to `Byte` and not to anything else such as `Short` or `Integer`.
5. **Applying rule 5** - A `byte` can be widened to `int` and an `int` can be accepted by a method that takes `int...` Therefore, the varargs version of the method will be invoked

Let me show you another example. What will happen if we call `processData(10)`; with the following two methods available:

```
void processData(Long value){ }

void processData(Long... values){ }
```

Observe that 10 is an `int`. Do we have a `processData` method that takes an `int`? No. So Rule 2 about exact match doesn't apply. `int` is not a subtype of `Long` or `Long[]` so Rule 3 doesn't apply either. Let's see if Rule 4 is of any help. 10 can be widened to a `long`, `float`, or `double` but we don't have any method that takes any of these types. Finally, as per Rule 5, 10 can be boxed into an `Integer` but that won't work either because there is no method that takes an `Integer`. Remember that `processData(Long)` cannot accept an `Integer` because `Integer` is not a subtype of `Long`.

Well, we don't have anymore options left to try. This means the compiler can't tie this call to any method and will therefore, raise an error saying, "Error: no suitable method found for `processData(int)`".

Method selection is not a trivial topic. The Java language specification spends a considerable number of pages defining the rules of method selection. Since it is easy to get bogged down with all those rules, I have tried to simplify them so they are easy to remember. You will be able to answer the questions in the exam and will also be able to figure out what a piece of code does in real life using the four points I have mentioned above.

Just be aware that there are several situations that cannot be explained by these points. Most of them involve Generics, a topic that is not on the exam. You should go through relevant sections of JLS if you are interested in exploring this topic further.

Exam Tip

You will most likely get only a single question on this topic. If you forget the above rules and find yourself taking too much time in figuring out the answer, I suggest you leave that question and move ahead. There is no point in wasting too much time on a question that is very easy to answer incorrectly.

11.3 Passing object references and primitive values into methods

11.3.1 Passing arguments to methods ↗

This section requires that you have a clear understanding of the difference between an object and a reference. I suggest you to go through the section 3.6.5, "Relation between Class, Object, and Reference" and section 5.2, "Difference between reference variables and primitive variables" to refresh your memory before proceeding with this topic.

Pass by value

To understand how Java passes arguments to methods, there is just one rule that you need to remember - Java uses **pass-by-value** semantics to pass arguments to methods. There are no exceptions. I mentioned this rule right at the beginning because if you keep this rule in your mind, this topic will feel like a piece of cake to you. You will never get confused with any code that the exam throws at you.

Let me start with the following simple code:

```
public class TestClass {  
  
    public static void main(String[] args){  
        int a = doubleIt(100);  
        System.out.println(a); //prints 200  
    }  
  
    public static int doubleIt(int x){  
        return 2*x;  
    }  
  
}
```

In the above code, the main method invoke the `doubleIt` method and passes the value `100` to the method. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `x` with the value that was passed, i.e., `100`. Thus, `x` now contains the value `100`. The return statement returns the value generated by the statement `2*x` back to the caller. The JVM assigns this value to the variable `a`. Thus, `a` now contains `200`. This is what is printed out.

Quite simple so far, right? Let me modify the code a bit now:

```
public class TestClass {  
  
    public static void main(String[] args){  
        int a = 100;  
        int b = doubleIt(a);  
        System.out.println(a+" "+b);  
    }  
  
    static int doubleIt(int x){  
        return 2*x;  
    }  
  
}
```

The only thing that I have changed is that instead of passing the value `100` directly to the `doubleIt` method, I am passing the variable `a`. The JVM notices that the method call is using a variable as an argument. It takes the value contained in this variable (which is `100`) and passes it to the method. Rest is exactly the same as before. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `x` with the value that was passed, i.e., `100`. Thus, `x` now contains

the value `100`. The return statement returns the value generated by the statement `2*x` back to the caller. The JVM assigns this value to the variable `b`. Thus, `b` now contains `200`. Finally, `100, 200` is printed.

Observe that `doubleIt` has no knowledge of the variable `a`. It only gets the value contained in the variable `a`, i.e., `100`, which is assigned to `x`. Are you with me so far? Alright, here comes the twist. Take a look at the following code:

```
public class TestClass {

    public static void main(String[] args){
        int a = 100;
        int b = doubleIt(a);
        System.out.println(a+" , "+b);
    }

    static int doubleIt(int x){
        x = 200;
        return 2*x;
    }
}
```

If you have understood the logic I explained earlier, you should be able to figure out what the above code prints. Let's follow the same process for analyzing what is happening here. The JVM notices that the method call is using a variable as an argument. It takes the value contained in this variable (which is `100`) and passes it to the `doubleIt` method. Just before starting the execution of the `doubleIt` method, the JVM initializes the parameter `x` with the value that was passed, i.e., `100`. Thus, `x` now contains the value `100`. Next, `x` is assigned a new value `200` by the statement `x = 200`; In other words, the `x` is overwritten with a new value `200`. The point to understand here is that this assignment has no effect of the variable `a` that was used in the calling method because `doubleIt` has absolutely no idea where the original value of `100` that it was passed as an argument came from. It merely gets the value `100` as an argument. In other words, the JVM passed only the value `100` to the method and not the variable `a`.

Next, the return statement returns the value generated by the statement `2*x` back to the caller, which is `400`. The JVM assigns this value to the variable `b`. Thus, `b` now contains `400`. At this time, you should realize that nothing was done to change the value of the variable `a` at all. It still contains the same value as before, i.e., `100`. Therefore, the print statement prints `100, 400`. The above example also shows that it is not possible for a method to change the value of the variable that was passed as an argument by the caller because that variable is never sent to the method. Only its value is sent. That is why the term "**pass by value**" is used to describe parameter passing in Java.

11.3.2 Passing objects to methods

In the previous lesson, you saw how passing a primitive variable to a method works. Passing a reference variable works exactly the same way. Check out the following code:

```
class Data{
```

```
    int value = 100;
}

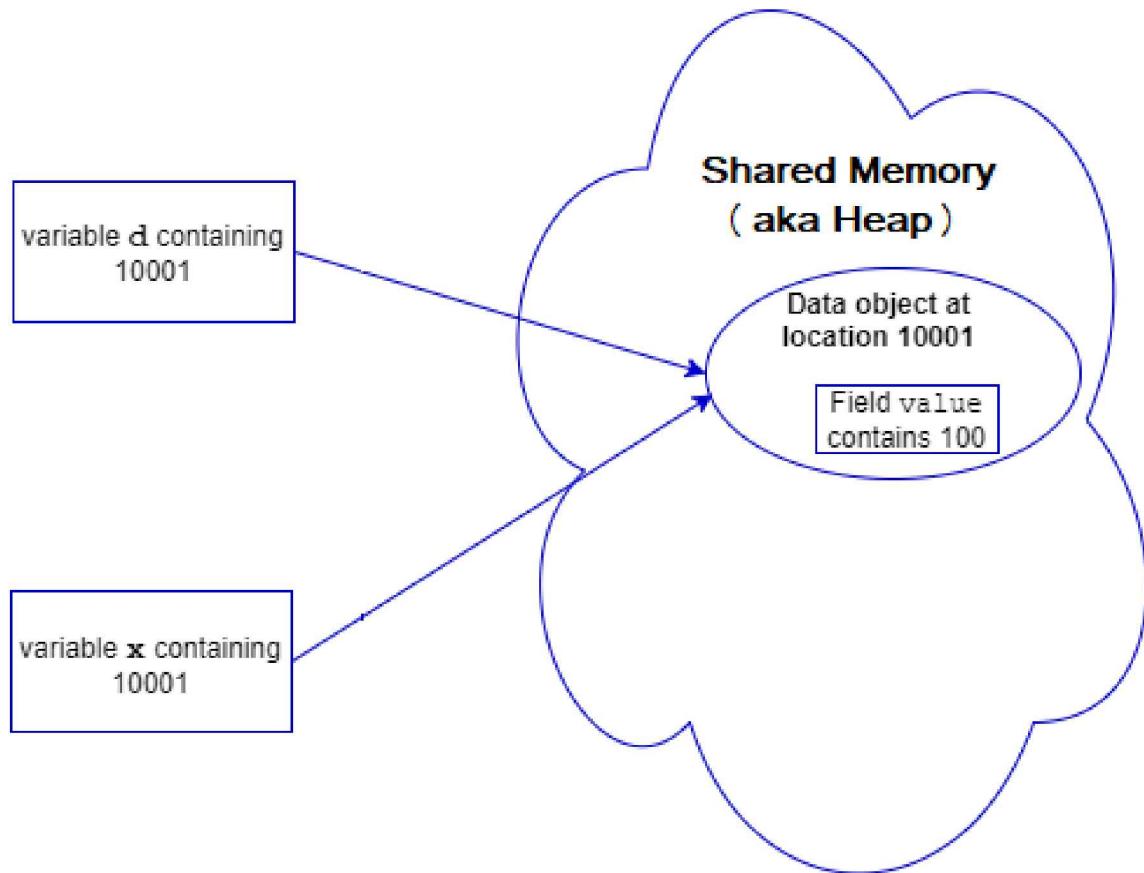
public class TestClass {

    public static void main(String[] args){
        Data d = new Data();
        modifyData(d);
        System.out.println(d.value); //prints 200
    }

    public static void modifyData(Data x){
        x.value = 2*x.value;
    }

}
```

In the above code, the main method creates a `Data` object and saves its reference in the variable `d`. It then calls the `modifyData` method. During execution, the JVM notices that the call to `modifyData` method uses a reference variable. Recall our discussion on reference variables in "Working with data types" chapter that a reference variable doesn't actually hold an object. It just holds the address of the memory location where the object resides. While invoking the `modifyData` method, the JVM copies the value stored in the variable `d` into the method parameter `x`. So, for example, if the `Data` object is stored at the memory address `10001`, the variable `d` contains `10001` and the JVM copies this value into the variable `x`. Thus, the variable `x` also now contains the same address as `d` and thus, in a manner of saying, variable `x` also starts pointing to the same object as the one pointed to by the variable `d`. This situation is illustrated in figure 1 below.



Passing an object to a method 1

The point to note here is that even though it looks as if we passed the `Data` object created in `main` to `modifyData`, all we actually passed was the value stored in variable `d` (which was nothing but the address of the `Data` object) to `modifyData`. The `Data` object itself remained exactly at the same memory location where it was. We didn't move it, pass it, or copy it, to anywhere.

The `modifyData` method uses the reference variable `x` to modify the `value` field of `Data` object pointed to by `x` to `200`. When the control goes back to the main method, it prints `d.value.`, which is now `200`.

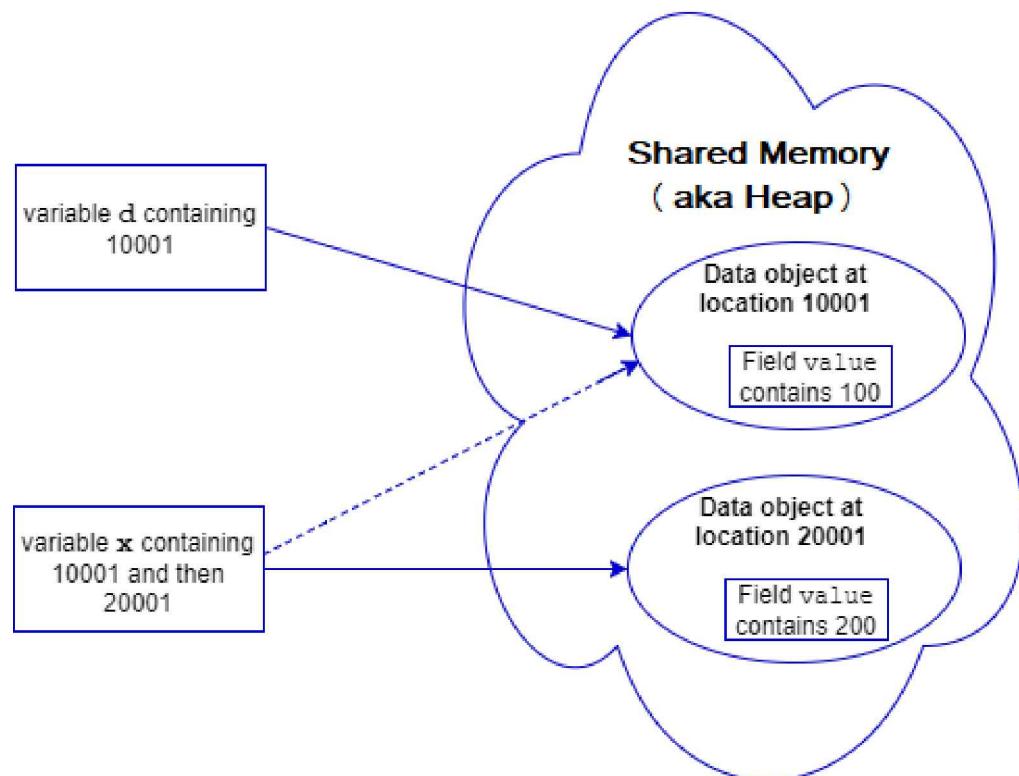
Let us make a small modification to the `modifyData` method of the above code:

```
public static void modifyData(Data x){
    x = new Data();
    x.value = 2*x.value;
}
```

All we have done is inserted the statement `x = new Data();`. Can you tell what the main method

will print now?

When the JVM invokes `modifyData` method, it makes `x` point to the same `Data` object as the one pointed to by variable `d`. This part is the same as before. But the first statement of the updated method creates a new `Data` object and makes `x` point to this new object. For example, if this new `Data` object resides at address 20001, then `x` now contains 20001 (instead of 10001). The next statement updates the value field using the variable `x`. Since `x` now points to the new `Data` object, this update statement updates the value field of the new `Data` object instead of the old `Data` object. When the control goes back to the main method, it prints `d.value`. This prints 100 because we never modified the variable `d` and so, `d` still points to the original `Data` object. This is illustrated in figure 2 below.



Passing an object to a method 2

In the two examples above, we passed a variable (the local variable `d`) as an argument to the `modifyData` method. Now, take a look at the following code:

```
class Data{
    int value = 100;
}
```

```

public class TestClass {

    public static void main(String[] args){
        modifyData(new Data());
    }

    public static void modifyData(Data x){
        x.value = 2*x.value;
    }

}

```

Observe the statement `modifyData(new Data());`. We are not passing any variable to the `modifyData` method here. So, are we really passing the `Data` object directly to the method? No, although we do not have any explicit variable to store the address of the `Data` object, the compiler creates a temporary reference variable implicitly. It creates the `Data` object in the heap space and assigns it to this temporary variable. While making the method call, it is the value of this variable that is passed to the `modifyData` method. Of course, since we have not saved a reference to this `Data` object in the `main` method, we won't be able to refer to this object after the `modifyData` method returns. This technique is used all the time while printing messages to the console using the `print/println` method. For example, when you do `System.out.println("hello world")`; the compiler passes the value of the temporary variable that points to the `String` object containing "hello world" to the `println` method.

11.3.3 Returning a value from a method

Just like parameters, returning a value from a method also uses "pass by value" semantics.

In the case of primitives, a return statement can return either the value directly (e.g. `return 100; return true; return 'a';` etc.) or the value of a variable (e.g. `return i;` where `i` is an `int` variable).

In the case of objects, a return statement can either return the value of an explicit variable (e.g. `return str;` where `str` is a `String` variable) or the value of an implicit temporary variable that references the object (e.g. `return "hello"; return new Student();` etc.). In both cases, it is really the address where the object is stored that is returned.

What the caller does with the return value is irrelevant to the method that returns a value. The caller can either use the return value further in the code or ignore the return value altogether.

11.4 Apply the static keyword to methods and fields

11.4.1 Apply the static keyword to methods and fields

I have already discussed the meaning of the word `static` and what it implies when applied to a method or a field in the "Kickstarter for Beginners" chapters.

In this section, I will dig a little deeper and explain the nuances of `static` from the perspective of the exam. Let us start with the syntax. You can declare any member of a class (i.e. a method,

a field, or any nested type definition) as static using the keyword static. In the case of a method or a field, this keyword must appear before the return type of the method or the type of the field respectively. For example:

```
class ConnectionHelper{  
    static int idle_timeout;  
    static String url;  
    private static void ping(){ } //order of modifiers doesn't matter  
    static public final void check(){ } //order of modifiers doesn't matter  
}
```

11.4.2 Accessing static members

A static member exists as a member of the owning class and not as a member of an instance of the owning class. In other words, a static member does not require an instance of the owning class to exist. A static member, therefore, can be accessed by specifying the name of the owning class. For example, the static members of `ConnectionHelper` and `Request` can be accessed from another class like this:

```
class TestClass{  
    public static void main(String[] args){  
        System.out.println(ConnectionHelper.idle_timeout); //prints 0 (why?)  
        System.out.println(ConnectionHelper.url); //prints null (why?)  
        Request.Data rc = new Request.Data();  
    }  
}
```

Accessing a static member using the name of the owning class is the standard and recommended way of accessing static members. However, Java allows a static member to be accessed through a variable as well. For example, the static variable `idle_timeout` of class `ConnectionHelper` can also be accessed like this:

```
class TestClass{  
    public static void main(String[] args){  
        ConnectionHelper c = null;  
        System.out.println(c.idle_timeout); //prints ConnectionHelper's idle_timeout  
    }  
}
```

Observe that `c` is `null`. But the compiler doesn't care about `c` being `null` because it notices that `idle_timeout` is a static variable and an instance of `ConnectionHelper` is not needed to access `idle_timeout`. The compiler knows what `idle_timeout` in the statement `c.idle_timeout` implies and effectively translates it to `ConnectionHelper.idle_timeout`.

This simple example highlights an important aspect of accessing static members: access to static members is decided by the compiler at compile time by checking the declared type of the variable. It is not decided by the JVM at run time. The compiler knows that the type of the variable `c` is `ConnectionHelper` and that `idle_timeout` is a static member of `ConnectionHelper`,

that is why the compiler binds the call to `c.idle_timeout` to `ConnectionHelper`'s `idle_timeout`. The compiler doesn't care what `c` may point to at run time. Now, armed with this knowledge, can you tell what the following code will do?

```
class TestClass{
    public static void main(String[] args){
        ConnectionHelper c = null;
        c.ping();
    }
}
```

That's right. It will compile and run fine (of course, without any output). But the interesting thing is that there will be no `NullPointerException`. Since the type of the reference variable `c` is `ConnectionHelper` and `ping()` is a static method of `ConnectionHelper`, the compiler binds the call to `c.ping()` to `ConnectionHelper`'s `ping()`.

This is called **static binding** or **compile time binding** because the compiler doesn't leave the decision of binding a call to the JVM. This concept will play an important role when we discuss inheritance and polymorphism later.

11.4.3 Accessing static members from the same class ↗

A static member of a class can also be accessed directly from static as well as instance members of the same class without the need to use the class name. For example, the following class makes use of a static variable to count the number of instances of that class:

```
class InstanceCounter{
    static int count;
    InstanceCounter(){
        //directly accessing count from a constructor
        count++;
    }

    static void printCount(){
        //directly accessing count from a static method
        System.out.println(count);
    }

    void reduceCount(){
        //directly accessing count from an instance method
        count--;
    }
}
```

The following class shows various ways in which you can access static members of a class from another class:

```
class TestClass {
    public static void main(String[] args){
```

```
InstanceCounter ic = new InstanceCounter();
ic.printCount(); //accessing static method through a reference to an instance

new InstanceCounter().printCount(); //accessing static method through an implicit
reference to an instance

InstanceCounter.printCount(); //accessing static method using the class name
System.out.println(InstanceCounter.count); //accessing static field using the
class name

}

}
```

You should be able to tell what the above code will print.

11.4.4 Accessing instance members from a static method

Since a static method belongs to a class and not to an object of that class, a static method does not execute within the context of any instance of that class. On the other hand, an instance method is always invoked on a specific instance of a class and so, it executes within the context of the instance upon which it is invoked. An instance method can access this instance using the **implicit** variable "**this**". Since there is no instance associated with a static method, the variable **this** is not available in a static method.

The reason why the "**this**" variable is called an implicit variable becomes important here. This variable is not declared explicitly anywhere and the compiler assumes its existence in an instance method. Whenever the compiler sees an instance member being accessed from within a method directly, the compiler uses the **this** variable to access that member even when you don't type it explicitly in your code. However, when a static method tries to access an instance member, **this** is not available and so, the compiler complains that a non-static variable cannot be referenced from a static context.

The following code illustrates this point:

```
class Book{
    int name;
    static void printName1(){
        System.out.println(this.name); //will not compile
        System.out.println(name); //same as above. will not compile
    }
    void printName2(){
        System.out.println(this.name); //this is fine
        System.out.println(name); //same as above. this is fine
    }
}
```

In the above code, the compiler realizes that `name` is an instance variable and so, tries to access it through `this`, i.e., `this.name`. But since `this` is not available in `printName1`, it generates an error. There is no issue with `printName2` because `printName2` is an instance method and `this` is available in an instance method.

Remember that you don't always need to explicitly use the variable `this` to access instance members. You need to use it only if there is also a local variable with the same name declared in the method and you want to refer to the instance variable instead of the local one. Technically, `this` is used to "unshadow" an instance variable if it is shadowed by a local variable of the same name.

A common misunderstanding amongst beginners is that a static method cannot access instance fields of a class. This misunderstanding exists because they see or hear this statement in many places. However, it is an incomplete statement. The correct statement is that a static method cannot access an instance member without specifying the instance whose member it wants to access. It will be clear when you see the following code:

```
class Book{  
    int name;  
    static void printName(){  
        Book b1 = new Book();  
        Book b2 = new Book();  
        System.out.println(b1.name); //this will compile fine  
    }  
}
```

In the above code, we are accessing the instance variable `name` from within a static method. Notice that there are two instances of `Book`. Each instance has its own copy of the variable `name`. But because we are using the reference `b1` to access `name`, the compiler knows exactly whose `name` we intend to access. It knows that we want to access the `name` field of the `Book` instance pointed to by the reference variable `b1`. That is why there is no problem. This shows that an instance field can indeed be accessed from a static method as long as we specify exactly the instance whose field we want to access.

11.5 Exercise

1. Create a method named `add` that can accept any number of `int` values and returns the sum of those values.
2. Create another method named `add` in the same class that can accept any number of `int` values but returns a `String` containing concatenation of all those values. What can you do to resolve compilation error due to the presence of the two methods with same signature? Invoke these methods from the main method of this class.
3. Create a class named `Student` with a few fields such as `studentId`, `name`, and `address`. Should these fields be static or non-static? Add the main method to this class and access the fields from the main method.

4. Add a static field to **Student** class. Access this field from another class. Use appropriate import statement to access the field directly. Change accessibility of the field and see its impact on the code that tries to access it.
5. Create a method named **method1** in **TestClass** that accepts a **Student** object and updates the static as well as instance fields of this object. Pass the same **Student** object to another method named **method2** and print the values. Assign a new **Student** object to the **Student** variable of **method2** and set its fields to different values. After returning back to **method1**, print the values again. Explain the output.
6. Add a constructor in **Student** class that accepts values for all of its instance fields. Add a no-args constructor in **Student** class that makes use of the first constructor to set all its instance fields to dummy values.
7. Create a class named **Course** in different package. Add a static method named **enroll** in this class that accepts a **Student**. Use different access modifiers for fields of **Student** class and try to access them from the **enroll** method.

Exam Objectives

1. Use the Math class
2. Use the Random class

12.1 Using the Math class

The Java standard library contains `java.lang.Math` class, which contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Observe that the Math class belongs to the `java.lang` package and since classes from this package are always accessible to all other class, there is no need to import this class in your code.

All the methods of this class are static. Thus, you don't need to instantiate an object of this class to use those method. In fact, its constructor is private, so, you can't really create an instance using `new Math()`.

This class has a huge number of methods but that is only because it contains overloaded methods that are merely copies for various input types. For example, there are four overloaded `abs` methods. One each for `int`, `long`, `float`, and `double` parameter types - `int abs(int i)`, `long abs(long l)`, `float abs(float f)`, and `double abs(double d)`. Another characteristic that you may observe in these four methods is that their return types match with their parameter types. All methods except the `round` methods follow this practice.

You need to be aware of the following methods of this class for the purpose of this exam:

1. **abs**: This method returns the positive value of the given argument. For example:

```
double d = Math.abs(-1.1); //assigns 1.1 to d
float f = Math.abs(-1.1); //will not compile because -1.1 is a double and so,
the return type will also be a double and a double cannot be assigned to a
float variable without a cast.
```

2. **ceil**: This method returns the smallest (closest to negative infinity) value that is greater than or equal to the argument and is equal to a mathematical integer. For example:

```
double d = Math.ceil(1.1); //assigns 2.0 to d
double d = Math.ceil(-1.1); //assigns -1.0 to d
```

3. **floor**: This method returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer. For example:

```
double d = Math.floor(1.1); //assigns 1.0 to d
double d = Math.floor(-1.1); //assigns -2.0 to d
```

4. **long round(double) and int round(float f)**: Returns the closest `long` (or `int`, in the case of the `float` version) to the argument, with ties rounding to positive infinity. For example:

```
long d = Math.round(1.1); //assigns 1 to d
long d = Math.round(1.5); //assigns 2 to d
long d = Math.round(-1.1); //assigns -1 to d
long d = Math.round(-1.5); //assigns -1 to d
```

Observe the difference in the return values of `round(1.5)` and `round(-1.5)`.

The next method in the Math class that you need to know about is `double random()`.

The Math.random() method

This method returns a random double value greater than or equal to `0.0` and less than `1.0`. For example:

```
double d = Math.random(); //assigns some value between 0.0 to less than 1.0 to d
```

Observe that it returns a `double`. So, if you want to use this method to get a random integer value between 0 to less than 10), you can do as follows:

```
int i = (int) (Math.random()*10); //assigns a random int value between 0 to 9 (both inclusive) to i
long l = Math.round(Math.random()*10); //Math.round(double) returns a long
```

Observe that multiplication by `10` gives you a double value between `0.0` to less than `10.0` and the cast simply truncates the decimal part. Also observe the parentheses around `Math.random()*10`. They are very important here. Recall from the Operators chapter that the cast operator has a higher precedence than the multiplication operator. Therefore, without the parentheses, return value of `Math.random()` will be cast to `int` first and that will always result in a zero because the value returned by the random method is always less than `1.0`! Therefore, you need to first multiply by `10` to get a random double value between `0.0` to less than `10.0` and then truncate the decimal part by casting to get an `int` value between `0` to less than `10`.

It is interesting that the exam asks you questions on the `Math.random()` method because Java has a dedicated feature rich class `java.util.Random` to generate random numbers, which is also on the exam.

12.2 Using the Random class

The `java.util.Random` class provides a more comprehensive, albeit somewhat complicated, solution to the problem of generating random numbers. Even the JavaDoc API description for this class accepts that many applications will find the `Math.random()` method simpler to use. However, for some reason, Oracle deems it important enough to test Java beginners on it.

The `Random` class lets you generate a stream of "pseudorandom" numbers, which means that the values generated by this class do not qualify as truly random but they are random enough for most practical purposes. The following example shows the basic usage of this class:

```
class TestClass{
    public static void main(String[] args){
        java.util.Random r = new java.util.Random(); //create an instance first
        int i = r.nextInt(); //get a random int value
        System.out.println(i);
        double d = r.nextDouble(); //gets a random double value
        System.out.println(d);
    }
}
```

Observe that, unlike with the `Math.random()` method, you need to first create an instance of the `Random` class and then call an appropriate `getXXX` method to get a random `int`, `long`, `float`, or `double` value.

Besides the no args `getInt` method, it also has `getInt(int bound)` method that takes a "bound". It returns a random integer between 0 to less than `bound`. For example, if you want to get a random integer between 0 to 9, you can do, `r.nextInt(10);` Compare this with `Math.random()`, where you had to do `(int) (Math.random()*10);` or `Math.round(Math.random()*10);.`

Using seed to create a Random instance

The `Random` class has two constructors - one that takes no argument, which you saw above and one that takes a `long` argument called "seed". Seed is a number that is required for the internal algorithm of the `Random` class to function. The random numbers generated by the algorithm depend on the seed. If two `Random` instances are created with the same seed value, then they will generate the same series of random numbers. For example, the following code prints the same two random numbers:

```
java.util.Random r1 = new java.util.Random(10);
    java.util.Random r2 = new java.util.Random(10);
    System.out.println(r1.nextInt()); //prints a random int
    System.out.println(r2.nextInt()); //prints the same random int as above
```

The no-args constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. Therefore, unless you know what you are doing, it is best to use the no-args constructor instead of the one with the seed parameter.

The `Random` class also has a `setSeed(long)` method that resets the seed used to generate new random values. Although the exam doesn't test you on the details of the algorithm that generates random numbers, it may test you on the usage of this method. So, here is how it can be used:

```
class TestClass{
    public static void main(String[] args){
        java.util.Random r = new java.util.Random();
        int i = r.nextInt();
        r.setSeed(10000); //set the new seed
        i = r.nextInt(); //gets another
        System.out.println(d);
    }
}
```

The `Random` class shines when you need not just one or two random numbers but a series of random numbers. It has several methods that return a `Stream` of random numbers. The concept of streams is too advanced for this exam and so, I will not talk about in this book.

12.3 Exercise ↗

1. Define a few `double` and `int` variables. Initialize them with positive as well as negative values. Assign the values of the double variables to the int variables after rounding them with the `round`, `ceil`, and `floor` methods. Observe which assignments require a cast and which do not.
2. Generate 10 random double values between 5 and 15(excluding) using `Math.random()`.
3. Generate 10 random double values between 1 and 10 (both including) using `Math.random()` and the `Random` class.
4. Create a array with a few elements. Shuffle the elements of the array in a random fashion.

Exam Objectives

1. Develop code that uses methods from the String class
2. Format Strings using escape sequences including %d, %n, and %s

13.1 Creating and manipulating Strings

13.1.1 What is a "string"?

In Java, a "string" is an object of class `java.lang.String`. It represents a series of characters. Strings such as `"1234"` or `"hello"` are really just objects of this class.

Not important for the JFCJA exam, but it is good to know that `String` is a **final** class. It extends `Object` and implements `java.lang(CharSequence` interface.

In the Java world, `String` objects are usually just called "**strings**". Although a string is just like any other regular object, it is such a fundamental object that Java provides special treatment to strings in terms of how they are created, how they are managed, and how they are used. Let's go over these aspects now.

13.1.2 Creating Strings

Creating strings through constructors

The `String` class has several constructors but for the purpose of the exam, you only need to be aware of the following:

1. `String()` - The no-args constructor creates an empty String.
2. `String(String str)` - Create a new String by copying the sequence of characters currently contained in the passed String or `StringBuilder` objects.
3. `String(byte[] bytes)` - Creates a new String by decoding the specified array of bytes using the platform's default charset.
4. `String(char[] value)`- Creates a new String so that it represents the sequence of characters currently contained in the character array argument.

Note that a string is composed of an array of `chars`. But that does not mean a string is the same as a `char` array. Therefore, you cannot apply the array indexing operator on a string. Thus, something like `char c = str[0];`, where `str` is a `String`, will not compile.

Creating strings through concatenation

The second common way of creating strings is by using the concatenation (, i.e., `+`) operator:

```
String s1 = "hello ";
String s12 = s1 + " world"; //produces "hello world"
```

The `+` operator is overloaded in such a way that if either one of its two operands is a string, it converts the other operand to a string and produces a new string by joining the two. There is no restriction on the type of operands as long as one of them is a string.

The way `+` operator converts the non-string operand to a string is interesting:

1. If the non-string operand is a reference variable, the `toString()` method is invoked on that reference to get a string representation of that object.
2. If the non-string operand is a primitive variable or a primitive literal value, a wrapper object of the same type is created using the primitive value and then a string representation is obtained by invoking `toString()` on the wrapper object.
3. If the one of the operands is a `null` literal or a null reference variable, the string "null" is used instead of invoking any method on it.

The following examples should make this clear:

```
String s1 = "hello ";
String s11 = s1 + 1; //produces "hello 1"

String s12 = 1 + " hello"; //produces "1 hello"

String s2 = "" + true; //produces "true";

double d = 0.0;
String s3 = "-" +d +"-"; //produces "-0.0-"

Object o = null;
String s4 = "hello "+o; //produces "hello null". No NullPointerException here.
```

Just like a mathematical expression involving the `+` operator, string concatenation is also evaluated from left to right. Therefore, while evaluating the expression `"1"+2+3`, `"1"+2` is evaluated first to produce `"12"` and then `"12"+3` is evaluated to produce `"123"`. On the other hand, the expression `1 + 2 +"3"` produces `"33"`. Since neither of the operands to `+` in the expression `1 + 2` is a `String`, it will be evaluated as a mathematical expression and will therefore, produce integer `3`. `3 + "3"` will then be evaluated as `"33"`.

Remember that to elicit the overloaded behavior of the `+` operator, at least one of its operands must be a `String`. That is why, the following statements will not compile:

```
String x = true + 1; //Will not compile. First operand is boolean and second is int

Object obj = "string"; //OK
String y = obj + obj; //Will not compile. Even though obj points to a String at
                      runtime, as far as the compiler is concerned, obj is an Object and not a String
```

Since the `toString` method is defined in the `Object` class, every class in Java inherits it. Ideally, you should override this method in your class but if you do not, the implementation provided by the `Object` class is used. Here is an example that shows the benefit of overriding `toString` in a class:

```
class Account{
    String acctNo;
    Account(String acctNo){
        this.acctNo = acctNo;
    }
}
```

```

//overriding toString
public String toString(){
    return "Account["+acctNo+"]";
}

public class TestClass{
    public static void main(String[] args){
        Account a = new Account("A1234");
        String s = "Printing account - "+a;
        System.out.println(s);
    }
}

```

The above code produces the following output with and without overriding `toString`:

Printing account - Account[A1234]

and

Printing account - Account@72bfaced

Observe that when compared to `Object`'s `toString`, `Account`'s `toString` generates a meaningful string. Since the `Object` class has no idea about what a class represents, it just returns a generic string consisting of the name of the class of the object, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. Don't worry, you will not be asked to predict this value in the exam. Just don't get scared if you see such a value in the exam.

On a side note, the `print/println` methods that we have often used also behave just like the `+` operator with respect to generating a string representation of the object that is passed to them. For example, when you call `System.out.print(acct);` where `acct` refers to an `Account` object, the `print` method invokes `toString` on that `Account` object and prints the returned string.

The `+=` operator

In the chapter on operators, we saw that `+=` is a compound operator. It applies the `+` operator on the two operands and then assigns the result back to the variable on the left side. As is the case with the `+` operator, the string concatenation behavior of `+=` is triggered when the type of any one of its operands is `String`. Here is an example:

```

String s = "1";
s += 2; //expanded to s = s + 2;
System.out.println(s); //prints "12"

```

Furthermore, if the result of the `+=` operator is a string, the type of the operand on the left must be something that can refer to a string, otherwise, the expression will not compile. There are only 4 such types other than `String` - the super classes of `String`, i.e., `CharSequence` and `Object` and, the interfaces that `String` implements, i.e., `Serializable` and `Comparable`. Here is an example:

```
int x = 1;
x += "2"; //will not compile
```

Since the type of one of the operands in the above expression is `String`, the String concatenation behavior of `+=` will be triggered. However, the expression will not compile because you can't assign the resulting object of type `String` to a variable of type `int`.

Observe that if the type of the left operand is `String`, the type of the right operand can be anything because in that case, even if the type of the right operand is not `String`, it will be converted to a string as per the rules discussed above.

Now, can you tell what the following code will print?

```
Object m = 1;
m += "2";
System.out.println(m);
```

It will compile fine and print "12". First, `1` will be boxed into an `Integer` object, which will be assigned to `m`. This assignment is valid because an `Integer` "is-a" `Object`. Next, the expression `m += "2"` will be expanded to `m = m + "2"`. Since one of the operands of `+` in this expression is a string, a string concatenation will be performed, which will produce the string "12". This string will be assigned to `m`. The assignment is also valid because a `String` is an `Object`.

Okay, how about this?

```
Object m = "Hello ";
m += 1;
System.out.println(m);
```

It will fail to compile because as far as the compiler is concerned, type of `m` is `Object` and type of `1` is `int`. Therefore, when `m+=1` is expanded to `m = m + 1`, neither of the operands of `+` is a `String`!

Strings and Character Encodings

The following discussion is not required for the JFCJA exam but is important to know for any Java developer

You know that computers deal with just raw bytes containing zeros and ones. They have no idea about human readable characters. Thus, every human readable character needs to be converted into a series of zeros and ones, aka a binary representation, before it can be stored. Similarly, every stored character needs to be converted back from the series of zeros and ones to a human readable character. But what should a character, say 'A', look like in the storage? What series of zeros and ones should be used to represent 'A' or any other character? If you consider the number of languages that people speak, there are literally thousands of characters that need to have binary representations. Who decides the binary representation of these characters? It is not a trivial task. Because whatever representation you use to store your character data, must also be understood by other people who are trying to read the data that you have stored. The answer to these questions is "character encoding" or "Charsets". Many computer scientists, groups, consortiums, and companies at various times have created "standards" that define how a particular character should be represented in the binary format. These standards are known as character encodings. One of the oldest and simplest standards is the ASCII standard which defines binary representation for all English alphabets and a few special characters using just 8 bits. So, for example, the character 'A' in ASCII can be stored as '1000001', which, in decimal is the number 65. So, if you store your text in ASCII format, you just need to tell the person reading that data that your data is in the ASCII format. They will then easily figure out that '1000001' means 'A'. However, the ASCII standard can store only 127 characters, which is enough to represent all English lower case and upper case characters but way too less than the total number of characters of all the languages that need to be represented!

Java uses a newer UTF-16 standard to store character data in Strings by default. UTF-16 uses up to 4 bytes to represent a character and is capable of representing a large number of characters. A good thing about this encoding is that its binary representations for all English letters matches with the ASCII representation, so, you will not notice a difference while dealing with English letters. You should be aware though that whenever you create a String, UTF-16 encoding is used to store the characters if you don't specify any encoding. Furthermore, if you want to read or write character data in any other charset, String class allows you to do that using the following two constructors - `String(byte[] bytes, String charsetName)` and `String(byte[] bytes, Charset charset)`. Similarly, you can use String's `getBytes(String charsetName)` and `getBytes(Charset charset)` methods to get a binary representation of text data in any other charset.

Strings containing special characters

You may sometimes need to store special characters such as new line, carriage return, and backslash characters in a String. Since these characters have a special meaning for a Java compiler, if you

try to put them in the String, they will cause compilation failure. For example, you can't write the following in your code:

```
String s = "first line
second line"; //compilation failure
```

Java provides "escape sequences" to put such special characters in a String. An escape sequence starts with a backslash (\) followed by a letter. For example, if you want to write a new line character in a string, you should write \n, i.e., "first line\nsecond line". Other commonly used escape sequences are \r for carriage return character, \t for tab character, and \\ for the backslash character. Note that '\n' does not constitute two characters. It is just one character. So, "1\n2".length() will return 3 and not 4. You can use these escape sequence for specifying char values as well. For example, char tab = '\t';

13.1.3 String interning

Since strings are objects and since all objects in Java are always stored only in the **heap space**, (heap space is just a common space for storing all objects created by a program) all strings are stored in the heap space. However, Java keeps compile time constant strings (i.e., strings that the compiler knows to be constants) in a special area of the heap space called "**string pool**". Java keeps all other strings (including the ones created using the new keyword) in the regular heap space.

String interning

The purpose of the string pool is to maintain a set of unique strings. Any time you create a new string using a constant expression (i.e. using hardcoded String literals and String concatenation operations on constant String variables), Java checks whether the same string already exists in the string pool. If it exists, Java returns a reference to the same String object and if it does not, Java creates a new String object in the string pool and returns its reference. So, for example, if you use the string "hello" twice in your code as shown below, you will get a reference to the same string. We can actually test this theory out by comparing two different reference variables using the == operator as shown in the following code:

```
String str1 = "hello"; //1
String str2 = "hello"; //2
System.out.println(str1 == str2); //prints true //3
String str3 = "hell"+ "o";
System.out.println(str1 == str3); //prints true //5
String str4 = new String("hello");
System.out.println(str2 == str4); //prints false //7
String str5 = "world";
String str6 = str2+str5; //creates "helloworld"
String str7 = str2+str5; //creates "helloworld"
System.out.println(str6 == str7); //prints false! //11
```

Recall from the Operators chapter that the == operator simply checks whether two references point to the same object or not and returns true if they do. In the above code, str2 gets the reference to the same String object which was created earlier at //1 because both str1 and str2

are assigned a compile time constant string "hello". Even `str3` gets the same reference because all the components of the expression "`hell`"+"`o`" (i.e., "`hell`" and "`o`") are compile time constants themselves. That is why, `str1 == str2` and `str1 == str3` produce `true`. On the other hand, the string referred to by `str4` is a new string created at run time and so, `str4` gets a reference an entirely different String object. That is why, `str2 == str4` returns `false`.

In fact, when you do `new String("hello")`; two String objects are created instead of just one if this is the first time the string "hello" is used anywhere in program - one in the string pool because of the use of a quoted string, and one in the regular heap space because of the use of new keyword.

Similarly, `str6` and `str7` get references to two different String objects because `str2` and `str5` are not constants. They are variables and so, the value of the expression `str5+str6` can only be computed at runtime. That is why `str6 == str7` returns `false`, even though both the variables point to strings containing the same data.

String pooling is Java's way of saving program memory by avoiding creation of multiple String objects containing the same value. It is possible to get a string from the string pool for a string created using the new keyword by using String's `intern` method. It is called "**interning**" of string objects. For example,

```
String str1 = "hello"; //str1 points to a String created in the string pool
String str2 = new String("hello"); //str2 points to a String created in the heap
String str3 = str2.intern(); //get an interned string referred to by str2
System.out.println(str1 == str3); //prints true
```

Generally, it is not possible to tell how the string returned by a method was created unless you go through the method code. The `intern` method is useful in such cases. If you want to work with an interned string only, you may get the interned version by invoking the `intern` method on the `String` reference returned by the method.

13.1.4 Quiz ↗

Q. How many string objects are created in the following lines of code?

```
String a, b, c;
a = new String("Good Morning!");
b = a;
c = a + b;
```

Select 1 correct option.

- A. one
- B. two
- C. three
- D. four

Correct answer is C.

The statement `a = new String("Good Morning!");` creates two String objects - one containing "Good Morning!" in the string pool and one containing the same string data in the regular heap space. The statement `b = a;` does not create any new String object. The statement `c = a + b;` creates a new String object containing "Good Morning!Good Morning!". Thus, a total of three String objects are created.

13.1.5 String immutability

Strings are immutable. It is impossible to change the contents of a string once you have created it. Let me show you some code that looks like it is altering a string:

```
String s1 = "12";
s1 = s1+"34";
System.out.println(s1); //prints 1234
```

The output of the above code indicates that the string pointed to by `s1` has been changed from "12" to "1234". However, in reality, the original string that `s1` was pointing to remains as it is. A new string containing "1234" is created instead and its address is assigned to `s1`. So, after the last line of the above code, the JVM would have created three different strings - "12", "34", and "1234".

There are several methods in the `String` class that may make you believe that they change a string but just remember that a string cannot be mutated. Ever. Here are some examples:

```
String s1 = "ab";
s1.concat("cd");
System.out.println(s1); //prints ab
s1.toUpperCase();
System.out.println(s1); //prints ab
```

In the above code, `s1.concat("cd")` does create a new string containing "abcd" but this new string is not assigned to `s1`. Therefore, the first `println` statement prints "ab" instead of "abcd". The same thing happens with `toUpperCase()`. It does produce a new string containing "AB" but since this string is not assigned to `s1`, the second `println` statement prints "ab" instead of "AB".

13.1.6 Using the methods of the String class

The `String` class contains several methods that help you manipulate strings. To understand how these methods work, you may think of a string as an object containing an array of chars internally. These methods simply work upon that array. Since array indexing in Java starts with 0, any method that deals with the locations or positions of characters in a string, also uses the same indexing logic. Furthermore, any method that attempts to access an index that is beyond the range of this array throws `IndexOutOfBoundsException`.

Here are the methods that belong to this category with their brief JavaDoc descriptions:

1. `int length()` - Returns the length of this string.

For example, `System.out.println("0123".length());` prints 4. Observe that the index of the last character is always one less than the length.

2. `char charAt(int index)` - Returns the char value at the specified index. Throws `IndexOutOfBoundsException` if the index argument is negative or is not less than the length of this string.

For example, `System.out.println("0123".charAt(3));` prints 3.

3. `int indexOf(int ch)/indexOf(String s)/lastIndexOf(int ch)/lastIndexOf(String s)` - Returns the index within this string of the first (or last, in case of `lastIndexOf` methods) occurrence of the specified character(or String). Returns -1 if the character (or String) is not found.

Examples:

```
System.out.println("0123".indexOf('2')); //prints 2
System.out.println("Hi, How are you?".indexOf("are")); //prints 8
System.out.println("/user/ceo/MyClass.java".indexOf('/')); //prints 0
System.out.println("/user/ceo/MyClass.java".lastIndexOf("//")); //prints 9
System.out.println("/user/ceo/MyClass.java".lastIndexOf(".class")); //prints -1
```

The above code illustrates the use of the all four variations of the `indexOf` methods.

A design philosophy followed by the Java standard library regarding methods that deal with the starting index and the ending index is that the starting index is always inclusive while the ending index is always exclusive. `String's substring` methods works accordingly:

1. `String substring(int beginIndex, int endIndex)` - Returns a new string that is a substring of this string.

Examples:

```
System.out.println("123456".substring(2, 4)); //prints 34.
```

Observe that the character at index 4 is not included in the resulting substring.

```
System.out.println("123456".substring(2, 6)); //prints 3456
System.out.println("123456".substring(2, 7)); //throws StringIndexOutOfBoundsException
```

2. `String substring(int beginIndex)` - This method works just like the other substring method except that it returns all the characters from beginIndex (i.e. including the character at the beginindex) to the end of the string (including the last character).

Examples:

```
System.out.println("123456".substring(2)); //prints 3456.
```

```
System.out.println("123456".substring(7)); //throws StringIndexOutOfBoundsException.
```

The rule about not including the element at the ending index is followed not just by the methods of the String class but also by methods of other classes that have a concept of element positions such as java.util.ArrayList.

The following methods return a new **String** object with the required changes:

1. **String concat(String str)** - Concatenates the specified string to the end of this string.

Example - `System.out.println("1234".concat("abcd")); //prints 1 234abcd`

2. **String toLowerCase()/toUpperCase()** - Converts all of the characters in this String to lower/upper case. Example - `System.out.println("ab".toUpperCase()); //prints AB`

3. **String replace(char oldChar, char newChar)** - Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

Example: `System.out.println("ababa".replace('a', 'c'));` //prints cbcbc

4. **String trim()** - Returns a copy of the string, with leading and/or trailing white space omitted.

Example:

```
System.out.println(" 123 ".trim()); //prints "123" (without the quotes)
```

One interesting thing about the String manipulation methods detailed above is that they return the same string if there is no change in the string as a result of the operation. Thus, all of the following print statements prints **true** because all of these operations return the same String object:

```
String s1 = "aaa"; //size of this string is 3
System.out.println(s1.substring(0,3) == s1); //prints true because the resulting
     substring is the same as the original string
System.out.println(s1.substring(0) == s1); //prints true because the resulting
     substring is the same as the original string
System.out.println(s1.replace('b', 'c') == s1); //nothing is replaced because there is
     no b in the string
System.out.println(s1.strip() == s1); //there is nothing to strip at the ends of the
     original string
```

It is very common to invoke these methods in the same line of code by chaining them together:

```
String str = " hello ";
str = str.concat("world ").trim().concat("!").toUpperCase();
System.out.println(str);
```

The above code prints **HELLO WORLD!**. Note that such chaining is possible only because these methods return a String. You will see a similar chaining of methods in **StringBuilder/StringBuffer** classes as well.

Finally, here are a few methods that let you inspect the contents of a string:

1. `boolean startsWith(String prefix)`: Returns `true` if this string starts with the specified prefix.
2. `boolean endsWith(String suffix)`: Returns `true` if this string ends with the specified suffix.
3. `boolean contains(CharSequence s)`: Returns `true` if and only if this string contains the specified sequence of char values.
4. `boolean equals(Object anObject)`: Returns `true` if the contents of this string and the passed string are exactly same. Observe that the type of the parameter is `Object`. That's because this method is actually defined in the `Object` class and the `String` class overrides this method. So, you can pass any object to this method, but if that object is not a string, it will return `false`.
5. `boolean equalsIgnoreCase(String anotherString)`: Compares this `String` to another `String`, ignoring case considerations.

The above methods are fairly self-explanatory and work as one would expect after looking at their names, so, I am not going to talk about them in detail here but I suggest you take a look at their JavaDoc descriptions and write a few test programs to try them out. You will not get any trick questions on these methods in the exam.

The last method of the `String` class that you need to be aware of for the JFCJA exam is `String[] split(String regex)`. It splits this string into an array of strings around matches of the input. For example, `"/user/ceo/MyClass.java".split("/")` will return a `String` array containing three strings - `"user"`, `"ceo"`, and `"MyClass.java"`. This method is very powerful because the argument is actually a "regular expression". Regular expressions is an advanced topic and is not on the JFCJA exam. The exam just expects you to know about the basic usage of this method.

13.1.7 Comparing strings

There are three ways you can compare strings - using the `==` operator and using the `equals` and the `compareTo` methods of the `String` class.

Comparison using the `==` operator

As I discussed earlier, the `==` operator, when applied to references, checks whether two references point to the same object or not. You can, therefore, use this operator on string references to check whether they point to the same `String` object or not. If the two references point to the same string, they are obviously "equal". The following code, for example, prints `true` for this reason:

```
String str = "hello";
System.out.println(str == "hello"); //prints true
```

Since creating strings in Java is quite easy, it is very tempting to use the `==` operator for testing their equality. However, it is very dangerous to do so as illustrated by the following code. All the `checkCode` method below wants to do is to check whether the string passed to it matches with the string referenced by the static variable named `code`.

```
public class TestClass{  
    static String code = "1234";  
    public static void checkCode(String str){  
        System.out.println(code == str);  
    }  
  
    public static void main(String[] args){  
        checkCode("1234");  
        checkCode(new String("1234"));  
    }  
}
```

This code prints `true` for the first comparison and `false` for the second. Ideally, it should have printed `true` for both. The problem is that the `checkCode` method has no knowledge of how the string that was passed to it was created. As discussed earlier, when you create a string using the `new` keyword, an entirely new `String` object is created. That is why the second check returns `false`. It shows that `==` operator cannot guarantee you the right result if you want to compare the character data of two strings.

Comparison using the `equals` method

`String` class has an `equals` method that compares the actual character data of two strings to determine whether they are equal or not. This method returns `true` if the data matches and `false` otherwise. Since it compares the actual characters contained in two strings, it doesn't matter whether the two string references are references to the same `String` object or are references to two different `String` objects. Thus, `new String("1234").equals("1234")` will always return `true` even though both are two different `String` objects. The `equals` method, therefore, is a better way to compare two strings.

To fix the code shown above, just replace `System.out.println(code == str);` with `System.out.println(code.equals(str));` It is common to invoke `equals` on a `String` literal and pass a `String` variable instead of the other way round., i.e., `"1234".equals(str);` is preferred to `str.equals("1234")` because if `str` is `null`, the first style returns `false` but the second style throws a `NullPointerException`.

Comparison using the `compareTo` method

While `==` and `equals` tell you only about the equality of two `Strings`, the `compareTo` method tells you about their lexicographical order, also known as dictionary order, as well. The `compareTo` method has the following signature:

```
int compareTo(String anotherString)
```

The `int` value that it returns tells you whether `anotherString` is "lessor" or "greater", i.e., whether

anotherString should appear before or after this String in a dictionary. If the two Strings are equal, the return value is 0. For example:

```
String a = "alice";
String b = "bob";
String c = "charlie";
System.out.println(a.compareTo(b)); //prints -1
System.out.println(b.compareTo(a)); //prints 1
System.out.println(a.compareTo(c)); //prints -2
System.out.println(a.compareTo(new String("alice"))); //prints 0
```

A negative value indicates that this `String` is lesser than `anotherString`. Furthermore, the magnitude of the returned number tells you the difference of the two character values at the earliest position where the two Strings differ. For example, `"alice"` and `"charlie"` differ at the first position itself and the difference between the two characters at the first position, i.e., `'a'` and `'c'` is 2. Since `'a'` comes before `'c'`, the return value of `a.compareTo(c)` is -2.



13.1.8 Quiz

Q. What will the following code print?

```
String a1 = "alice";
String a2 = new String(a1);
System.out.format( (a1==a2)+" "+a1.equals(a2)+" "+a1.compareTo(a2)); //5
```

Select 1 correct option.

- A. false false false
- B. true false 1
- C. false true true
- D. false true 0 **Correct answer is D.**

`new String(a1)` creates a new `String` object. Although the new object will contain the same characters as the `String` pointed to by `a1`, it is, nevertheless, a different object. Therefore, `a1==a2` will return `false`.

Since `equals` and `compareTo` compare the contents of the two `String` objects and since the contents of the `Strings` pointed to be `a1` and `a2` are the same, `a1.equals(a2)` will return `true` and `a1.compareTo(a2)` will return 0.

13.2 Formatting Strings

So far you have used `System.out.print` and `System.out.println` methods to print messages to the console. For example, if you want to print something like `"Hello, Bob! How are you? Today is 1st Oct 2020."`, you could do - `System.out.print("Hello, "+name+"! How are you? Today is "+dateString+".")`; where the value for name would be taken from the `name` variable and the value for date would be taken from the `dateString` variable. The problem with this style

is quite apparent. It is fine for printing short messages during development but it is too primitive when you need to print large preformatted messages with lots of parameterized data.

A better way to achieve the same is: `System.out.format("Hello, %s! How are you? Today is %s.", name, dateString);` Observe that the message body is now separated from the customized values appearing inside the message. The resulting string is produced by replacing the first `%s` with the value of `name` variable and the second `%s` with the value of the `dateString` variable.

Recall that `out` is a static variable in the `java.lang.System` class of type `java.io.PrintStream`. `PrintStream` has the following `format` method, which makes this possible:

PrintStream format(String format, Object... args) : It writes a formatted string to this output stream using the specified format string and arguments. The first parameter is the main body of the text that you want to print (also called "template") and the second is a varargs parameter for the values that you want to embed into the message. Each value is formatted and inserted according to the format specifications contained within the template to create the resulting String.

`PrintStream` also has a `printf` method, which works exactly the same as `format`. You may see the usage of both of them in the exam questions.

The format specification itself is quite powerful and has several bells and whistles for generating complicated messages. However, for the purpose of the exam, you need to know only the basic technique using a few format specifiers: `%s`, `%d`, and `%n`.

1. `%s` : Used to embed String values.
2. `%d` : Used to embed numeric values in the format.
3. `%n` : Used to embed the line character

Remember that it must be possible for the values to be converted into the given format. For example, if your format specifier is `%d`, then the corresponding value must be an integral value (byte, char, short, int, and long). If you try to pass any other type of value, such as a `double` or a `String`, a `java.util.IllegalFormatConversionException` will be thrown. However, any type of value can be converted into a `String`, therefore, you can use any type of value with `%s`. Furthermore, the number of format specifiers used in the format String must not be more than the number of arguments in the varargs array, otherwise a `java.util.MissingFormatArgumentException` will be thrown. Extra values are fine.

The following example and its output should make this clear:

```
String name = "bob";
int acctNo = 1234;
int points = 100;
System.out.format("Hello, %s!\nThe point balance in your account %s is %d.%n",
    name, acctNo, points); //4
System.out.format("Great Job!\n", name); //5
System.out.format("These points are worth %d USD.%nThank you!", points*10.0); //6
```

It produces the following output:

```
The point balance in your account 1234 is 100.  
Great Job!  
These points are worth Exception in thread "main"  
java.util.IllegalFormatConversionException: d != java.lang.Double
```

You should observe the following points about the above code:

1. The line at //4 shows that integral values can be embedded in the message with `%d` as well as `%s`.
2. The line at //5 shows that extra values is not a problem.
3. The line at //6 shows that a double value cannot be converted to an integer using `%d`.

The `String.format` method

Sometimes you may not necessarily want to print the formatted string but use it for a different purpose. For example, you may want to use them as the body of an email or you may just want to store them in the database. The `String` class has a static method named `format`, which generates formatted Strings using the same formatting principles explained above. But instead of printing the resulting string, it simple returns the generated string. For example: `String message = String.format("Hello, %s!", name);`

13.3 Exercise

1. Create a few strings that contain special characters such as new line and backslash. Print their lengths.
2. Use the `indexOf` method to print the positions at which the strings created in the previous task contain the special characters.
3. Write a method that accepts a `String` as input and count the number of spaces in the string from start to until it finds an '`x`', or if there is no '`x`' in the string, till end.
4. Write code to determine whether the `toString` and `substring` methods of `String` class return an interned string or not. Confirm your results by checking the JavaDoc API descriptions of these methods.
5. Write a method that takes a `String` and returns a `String` of the same length containing the '`X`' character in all positions except the last 4 positions. The characters in the last 4 positions must be the same as in the original string. For example, if the argument is "`12345678`", the return value should be "`XXXX5678`".
6. Write a method that takes a `String[]` as an argument and returns a `String` containing the concatenation of all the strings in the input array. Invoke your method with different arguments. Make sure that your code handles the cases where the argument is null, contains a few nulls, or contains only nulls.

7. What will the following code print and why?

```
String s = "a";
String[] sa = { "a", s, s.substring(0, 1), new String("a"), ""+'a' };
for(int i=0; i<sa.length; i++){
    System.out.println(i);
    System.out.println(s == sa[i]);
    System.out.println(s.equals(sa[i]));
}
```

8. Create two arrays. One containing String values: Amy, Boyd, and Cathy, and another one containing int values: 10, 11, and 12. Use `printf` and `format` methods to create an HTML table showing these values as name value pairs.

Exam Objectives

1. Identify syntax and logic errors
2. Use exception handling
3. Handle common exceptions thrown
4. Use try and catch blocks

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

14.1.1 Java exception handling ↗

Exceptions are for managing exceptional situations. For a file copy program, the normal course of action could be - open file A, read the contents of file A, create file B, and write the contents to file B. But what if the program is not able to open file A? What if the program is not able to create or write to file B? There could be many reasons for such failure such as a typo in the file name, lack of permission, no space on disk, or even disk failure. Now, you don't expect these problems to occur all the time but it is reasonable to expect them some times. Since we don't expect these problems to occur regularly in normal course of operation, we call them "exceptional".

In exceptional situations, you may want to give the user a feedback about the error or you may even want to take an alternative course of action. For example, you may want to let the user to input another source file name if the input file is not found or another target location if the given location is out of space. Even if you don't want to take any special action upon such situations, you should at least want your program to end gracefully instead of crashing unexpectedly at run time. This means, you should provide a path for the program to take in such situations. One way would be to check for each situation before proceeding to copy. Something like this:

```
if(checkFileAccess(file1)){
    if(checkWritePermission(targetDirectory){
        //code for normal course of action
    }else{
        System.out.println("Unable to create file2");
    }
}else{
    System.out.println("Unable to read file1");
}
```

You can see where this is going. You will end up having a lot of if-else statements. Not only is it cumbersome to code, it will be a nightmare to read and maintain later on.

There is another serious problem with the above approach. It doesn't provide way to write code for situations that you don't even know about at the time of writing the code. For example, what if the user runs this code in an environment that requires file names to follow a particular format? So, now, you have two kinds of "exceptional situations". One that you know about at the time of writing the code, and one that you don't know anything about.

Java exception mechanism is designed to help you write code that covers all possible execution paths that a program may take - 1. path for normal operation 2. paths for known exceptional situations, and 3. path for unknown exceptional situations. Here is how the above mentioned program can be written:

```
try{
```

```
//code for normal course of action
}catch(SecurityException se){
    //code for known exceptional situation
    System.out.println("No permission!");
}
catch(Throwable t){
    //code for unknown exceptional situations
    System.out.println("Some problem in copying: "+t.getMessage());
    t.printStackTrace();
}
```

Observe that the perspective is reversed here. In the if/else approach, you check for each exceptional situation and proceed to copy if everything is good, while in the try/catch approach, you assume everything is good and only if you encounter a problem, you decide what to do depending on the problem. The benefits of the try/catch approach are obvious. It provides a clean separation between code for normal execution and code for exceptional situations, which makes the code easier to read and maintain. It allows an alternative path to the program to proceed even in cases where the programmer has not anticipated the problem.

Exception handling in Java is not without its criticism. There has been a good amount of debate on what constitutes "exceptional situations" and what should be the right approach to handle such situations. My objective in this chapter is to teach you Java's approach to exception handling and so, I will not go into an academic discussion on whether it is good or bad as compared to other languages. However, it is a very important topic of discussion in technical interviews. I suggest you google criticism of Java's exception handling and compare it with C#'s.

14.1.2 Fundamentals of the try/catch approach

When you think of developing code as developing a component, you will realize that there are always two stakeholders involved - the provider/developer of the component and the client or the user of the component. For the client to be able to use the component, it is imperative for the provider to tell the client about "how" his component works. The how not only includes the input/output details of the component but also the information about exceptional situations, i.e., about situations the component knows may occur but does not deal with.

For example, let's say you are developing a method that copies the contents of one file to another and that this method is to be used by a developer in another team. You must convey to the other developer that you are aware of the Input/Output issues associated with reading and writing a file but you won't do anything about them. In other words, you must convey that your method will try its best to copy the file but if there is an I/O issue, it will abandon the attempt and let her know about the failure so that she can deal with it however she wants to. This is done through the use of a "throws" clause in method declaration:

```
public void copyFile(String inputPath, String outputPath) throws IOException {
    //code to copy file
```

```
}
```

In the above code, if the `copyFile` method encounters any I/O issue while copying a file, it will simply abort the copying and throw an `IOException` to the caller. If it does not encounter any I/O issue, the method will end successfully.

On the other side of the component is the user of that component, who uses the information provided by the component provider to develop her code. The user has to decide how she wants to handle the exceptional situation. If she believes that she has the ability to "resolve" the situation, she will put the usage of that component in a "`try`" block with an associated "`catch`" block that contains code for "resolution" of the problem. When that exceptional situation actually occurs, the control goes to the catch block instead of proceeding to the next statement after the method call that threw the exception.

If the user decides that she cannot handle the exceptional situation either, she propagates the exception to the caller of her component.

For example, a program that creates backup of a file may use the file copy program internally to copy a file. If the file copy program throws an `IOException`, the backup creator program may catch that exception and show a message to the user.

```
public void createBackup(String input) {
    String output = input + ".backup";
    try{
        copyFile(input, output);
        System.out.println("backup successful");
    }catch(IOException ioe){
        System.out.println("backup failure");
    }
}
```

If the `copyFile` method completes without any issue, the control will go to the `println` statement. If the `copyFile` method throws an `IOException`, the control will go to the catch block, thus, providing an opportunity to take a different course of action. Here, the code may show a failure message to the user. From the perspective of the developer of the backup program, showing the error message to the user is the resolution of the I/O problem. One can also write code to have the user specify another file or directory to create a backup.

An exception is considered "handled" when it is caught in a catch block. If the backup creator method doesn't know what to do in case the `copyFile` is not successful, then it should let the exception propagate to its caller by using a `throws` clause of its own:

```
public void createBackup(String input) throws IOException{
    String output = input + ".backup";
    copyFile(input, output);
    System.out.println("backup successful");
}
```

In the above code, if the `copyFile` method completes without any issue, the control will go to the `println` statement. But if the `copyFile` method throws an `IOException`, the `createBackup` method will end immediately and the caller will receive an `IOException`. Note that this will be

the same `IOException` that it receives from the `copyFile` method. This is how an exception propagates from one method to the other. There could be a long chain of method calls through which an exception bubbles up before it is handled. If an exception is not caught anywhere while it is bubbling up the call chain, it ultimately reaches the JVM code. Since the JVM has no idea about the business logic of the program that it is executing, it "handles" the exception by killing(aka terminating) the program (actually, the JVM kills only the thread that was used to invoke the chain of method calls but threading is not on the JFCJA exam, so you can assume for now that a program is composed of only one thread and killing of that thread is the same as killing the program).

14.1.3 Pieces of the exception handling puzzle

Java exception handling mechanism comprises several moving parts. What makes this mechanism a bit complicated is that you need to put each part in its right place to make the whole thing work. So, let me introduce these parts first briefly.

The `java.lang.Throwable` object -

`Throwable` is the root class of all exceptions. It captures the details of the program and its surroundings at the time it is created. Among other things, a `Throwable` object includes the chain of the method calls that led to the exception (known as the "stack trace") and any informational message specified by the programmer while creating that exception. This information is helpful in determining the location and the cause of the exception while debugging a program. You may have seen crazy looking output on the console containing method names upon a program crash. This output is actually the stack trace contained in the `Throwable` object.

`Throwable` has two subclasses - `java.lang.Error` and `java.lang.Exception`, and a huge number of grand child classes. Each class is meant for a specific situation. For example, a `java.lang.NullPointerException` is thrown when the code tries to access a null reference or an `java.lang.ArrayIndexOutOfBoundsException` is thrown when you try to access an array beyond its size. You can also create your own subclasses by extending any of these classes if the existing classes don't represent the exceptional situation appropriately. For example, an accounting application could define a `LowBalanceException` for a situation where more money is being withdrawn from an account than what the account has.

What is called "**exception**" (i.e. exception with a lower case 'e') in common parlance, is in reality is an object of class `java.lang.Throwable` or one of its subclasses.

One subclass of `java.lang.Exception` that is particularly important is `java.lang.RuntimeException`. `RuntimeException` and its subclasses belong to a category of exception classes called "**unchecked exceptions**". You will see their significance soon.

The `throw` statement

The `throw` statement is used by a programmer to "throw an exception" or "raise an exception" explicitly. A programmer may decide to throw an exception upon encountering a condition that makes continuing the execution of the code futile. For example, if, while executing a method, you

find that the value of a required parameter is invalid, you may throw an `IllegalArgumentException` using a `throw` statement like this:

```
public double computeSimpleInterest(double p, double r, double t){
    if( t<0) {
        IllegalArgumentException iae = new IllegalArgumentException("time is less than
0");
        throw iae;
    }
    //other code
}
```

Usually, an exception object is created only to be "thrown" and so, there is no need to store its reference in a variable. That is why it is often created and thrown in the same statement as shown below:

```
public double computeSimpleInterest(double p, double r, double t){
    if( t<0) throw new IllegalArgumentException("time is less than 0");
    //other code
}
```

Throwing an exception implies that the code has encountered an unexpected situation with which it does not want to deal. The code shown above, for example, expects the time argument to be greater than zero. For this code, time being less than zero is an unexpected situation. It does not want to deal with this situation (probably because the programmer is not sure what to do in this case) and so it throws an exception in such a situation. This also means that this method passes on the responsibility of determining what should be done in case time is less than zero to the user of this method.

Note that only an instance of `Throwable` (or its subclasses) can be thrown using the `throw` statement. You cannot do something like `throw new Object();` or `throw "bad situation";`

Explicitly throwing an exception using the `throw` statement is not the only way an exception can be thrown. JVM may also decide to throw an exception if the code tries to do some bad thing like calling a method on a null reference. For example:

```
public void printLength(String str){
    System.out.println(str.length());
}
```

If you pass `null` to the above method, the JVM will create a new instance of `NullPointerException` and throw that instance when it tries to execute `str.length()`.

The `throws` clause

Java requires that you list the exceptions that a method might throw in the `throws` clause of that method. This ties back to Java's design goal of letting the user know of the complete behavior of a method. It wants to make sure that if the method encounters an exceptional situation, then the method either deals with that situation itself or lets the caller know about that situation. The

throws clause is used for this purpose. It conveys to the user of a method that this method may throw the exception mentioned in the **throws** clause. For example:

```
public double computeSimpleInterest(double p, double r, double t) throws Exception{  
    if( t<0) throw new Exception("time is less than 0");  
    //other code  
}
```

Now, anyone who uses the above method knows that this method may throw an exception instead of returning the interest. This helps the user write appropriate code to deal with the exceptional situation if it arises.

The **try** statement

A **try statement** gives the programmer an opportunity to recover from and/or salvage an exceptional situation that may arise while executing a block of code. A try statement consists of a **try block**, zero or more **catch blocks**, and an optional **finally block**. Its syntax is as follows:

```
try {  
    //code that might throw exceptions  
}catch(<ExceptionClass1> e1){  
    //code to execute if code in try throws exception 1  
}  
catch(<ExceptionClass2> e2){  
    //code to execute if code in try throws exception 2  
}  
catch(<ExceptionClassN> en){  
    //code to execute if code in try throws exception N  
}  
finally{  
    //code to execute after the try block and catch block finish execution  
}
```

Note that curly braces for the **try**, **catch**, and **finally** blocks are required even if there is a single statement in these blocks (compare that to **if**, **while**, **do/while** and **for** blocks where curly braces are not required if there is only one statement in the block). Furthermore, a **try block** must follow with at least one **catch block** or the **finally block**. A try block that follows with neither a catch block nor a finally block will not compile.

The **try** statement is the counterpart of the **throw** statement because putting a piece of code within a try block signifies that the programmer wants do something in case that piece of code throws an exception. In other words, a try block lets the programmer deal with an exceptional situation as opposed to the **throw** statement, which lets the programmer avoid dealing with it.

While a try block contains code for normal operation of the program, a **catch block** is the location where the programmer tries to recover from an exceptional situation that arises in the try block. If the code in the try block throws an exception, the normal flow of execution in that try block is aborted (i.e. no further code in the try block is executed) and the

control goes to the catch block. Here, the programmer can take alternate approach to finish the processing of the method. For example, the programmer may decide to just show a popup message to the user about the exception and move on to the next statement after the try statement.

A catch block is associated with a **catch clause**, which specifies the exception that the catch block is meant to handle. For example, a catch block with the catch clause as `catch(IllegalArgumentException e)` is meant to handle `IllegalArgumentException` and its subclasses. Thus, this catch block will be executed only if the code in the try block throws an `IllegalArgumentException` or its subclass exception. You can specify any valid exception class (including `java.lang.Throwable`) in the catch clause.

A **finally block** is the location where the programmer tries to salvage the situation or control the damage so to say, without attempting to recover from an exception. For example, a program that tries to copy a file may want to close any open files irrespective of whether the copy operation is successful or not. The programmer can do this in the finally block. You may think of a finally block as the step where a car mechanic reassembles the parts back irrespective of whether he was able to fix the car or not!

Here is a method that calls the `computeSimpleInterest` method shown above within a try statement:

```
public void doInterest(){

    try{
        double interest = computeSimpleInterest(1000.0, 10.0, -1);
        System.out.println("Computed interest "+interest);
    }catch(Exception e){
        System.out.println("Problem in computing interest:"+e.getMessage());
        e.printStackTrace();
    }finally{
        System.out.println("all done");
    }

}
```

In the above code, the call to `computeSimpleInterest` throws an `IllegalArgumentException` because `t` is negative. Thus, the `println` statement after the method call is not executed. The exception is caught in the catch block because its catch clause, i.e., `catch(IllegalArgumentException)` matches with the exception that is thrown by the try block and the control goes to the first statement in this catch block. It prints the exception's message and the stack trace on the console. Finally, the control goes to the code in the finally block, where it prints "all done". If you omit the catch block, the control will go directly to the finally block after the invocation of `computeSimpleInterest` method. After the execution of the code in the finally block, the caller of `doInterest` method will receive the same `IllegalArgumentException`.

Note that a finally block, if present, always executes irrespective of what happens in the

try block or the catch block. Even if the try block throws an exception and there is no catch block to catch that exception, the JVM will execute the finally block. It will throw the exception to the caller only after the finally block finishes. The only case where the finally block is not executed is if the code in the try or the catch block forces the JVM to shut down using a call to `System.exit()` method.

The following is a complete program that illustrates how an exception alters the normal program flow:

```
public class TestClass{  
  
    public static void main(String[] args){  
        TestClass tc = new TestClass();  
        tc.doInterest();  
    }  
  
    public double computeSimpleInterest(double p, double r, double t){  
        if( t<0 ) throw new IllegalArgumentException("time is less than 0");  
        return p*r*t/100;  
    }  
  
    public void doInterest(){  
        try{  
            double interest = computeSimpleInterest(1000.0, 10.0, -1);  
            System.out.println("Computed interest "+interest);  
        }catch(IllegalArgumentException iae){  
            System.out.println("Problem in computing interest:"+iae.getMessage());  
            iae.printStackTrace();  
        }finally{  
            System.out.println("all done");  
        }  
    }  
}
```

It generates the following output on the console :

```
Problem in computing interest:time is less than 0  
java.lang.IllegalArgumentException: time is less than 0  
    at TestClass.computeSimpleInterest(TestClass.java:8)  
    at TestClass.doInterest(TestClass.java:14)  
    at TestClass.main(TestClass.java:4)  
all done
```

You should observe the following points in the above code:

1. The `return` statement in `computeSimpleInterest` is not executed because the previous statement throws an exception.

2. The `println` statement in the try block of `doInterest` is not executed because the call to `computeSimpleInterest` ends with an exception instead of a return value. Control goes to the `catch` block directly after the method call.
3. The `catch` block prints the details captured in the exception object. It shows the sequence of the method invocations in reverse order from the point where the `IllegalArgumentException` object was created. You should try removing the catch block and see the output.
4. Once the `catch` block is finished, the control goes to the `finally` block.
5. The `doInterest` method returns after the execution of the `finally` block.
6. There is no throws clause in `computeSimpleInterest` method even though it throws an exception. The reason will be clear in the next section where I talk about checked and unchecked exceptions.

14.2 Differentiate among checked, unchecked exceptions, and Errors

14.2.1 Checked and Unchecked exceptions

As discussed earlier, exceptions thrown by a method are part of the contract between the method and the user of that method. If the user is not aware of the exceptions that a method might throw, she will be blindsided during run time because her code would not be prepared to handle that exception. The `throws` clause of a method is meant to list all exceptions that the method might throw. If an exception is listed in the throws clause, the user of the method will know that she needs to somehow handle that situation.

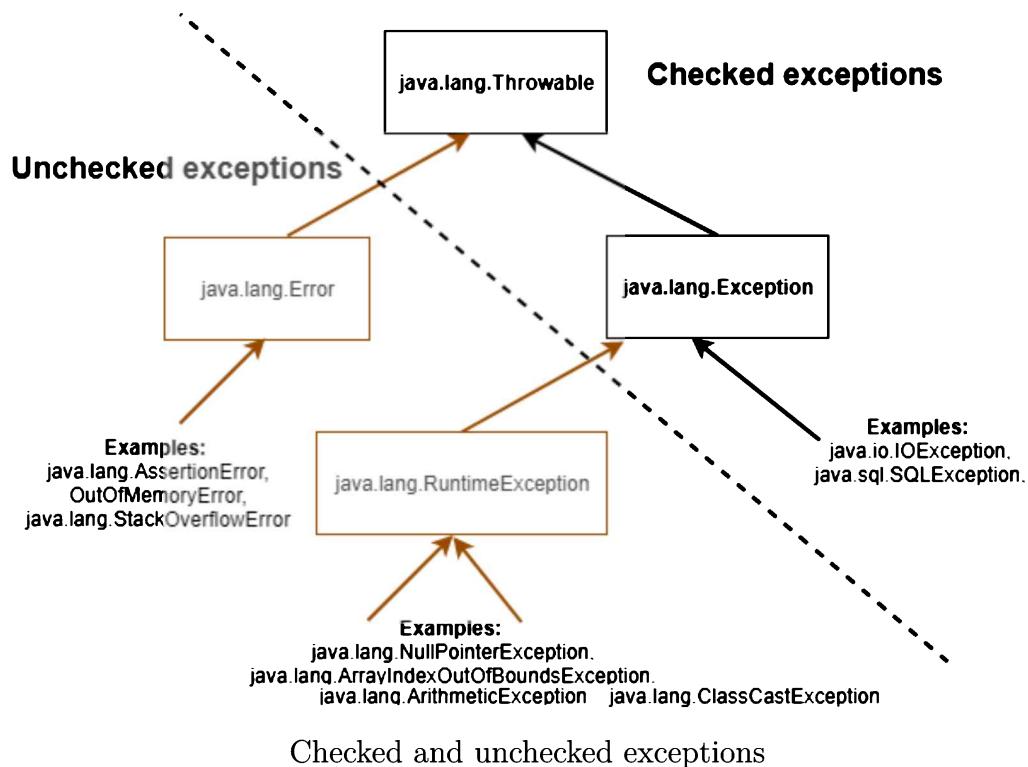
But what if a developer forgets to list an exception in the throws clause of a method? In that case, we are back to the same problem of blindsiding the user of that method at run time. This is where the compiler plays an important role. While compiling a method, the compiler checks for the possibility of any exception that might get thrown by the method to the caller. If that exception is not listed in the throws clause of that method, it refuses to compile the method.

This sounds like a good solution, but the problem here is that the compiler does not check for all kinds of exceptions thrown by a piece of code. It checks for only a certain kind of exceptions called "**checked exceptions**" and forces you to list the only those exceptions that belong to this category of exceptions in the throws clause.

Exceptions that do not belong to the category of checked exceptions are called "**unchecked exceptions**". They are called unchecked because the compiler doesn't care whether a piece of code throws such an exception or not. Listing unchecked exceptions in the throws clause is optional.

Finding out whether an exception is a checked exception or not is easy. Java language designers have postulated that any exception that extends `java.lang.Throwable` but does not extend `java.lang.RuntimeException` or `java.lang.Error` is a checked exception. The rest (i.e. `java.lang.Error`, `java.lang.RuntimeException`, and their subclasses) are unchecked.

The following figure illustrates this grouping of exceptions.



Checked and unchecked exceptions

Note that Java has a deep rooted exception class hierarchy, which means there are several classes, subclasses, and subclasses of subclasses in the tree of exceptions. So just because an exception is a `RuntimeException`, does not mean that that exception directly extends `RuntimeException`. It could even be a grand child of `RuntimeException`. For example, `ArrayListOutOfBoundsException` actually extends `IndexOutOfBoundsException`, which in turn extends `RuntimeException`. Similarly, just because an exception is a checked exception does not mean that it directly extends `Exception`. It may extend any subclass of `Exception` (except `RuntimeException`, of course).

Rationale behind checked and unchecked exceptions

Recall our discussion on exceptional situations where I talked about two kinds of exceptional situations - ones that a developer knows about and ones that a developer doesn't expect to occur at all. While a developer may want to provide an alternate path of execution in case of a situation that is known to occur but there is no point in providing an alternate path of execution for a situation that is never expected to occur.

For example, if a piece of code tries to write to a file, the developer may want to take a different approach if he is not able to write to the file system. But if the data array that it is trying to write is null, there is nothing much he can do. Such unexpected situations occur mostly due to badly written code. In other words, if a code gets itself into an unexpected situation, it is most likely because of a programming error, i.e., a bug in the code. Such issues should be fixed during development itself. But if they do occur in production, they should rather be handled at a much higher level than at the component level.

Unchecked exceptions are for such **unexpected** situations. Java language designers believe that unchecked exceptions need not be declared in a method declaration because there is nothing to gain by forcing the caller to catch them. Only checked exceptions need to be declared because the caller of the method may have a plan to recover from them. There are two kinds of unchecked exceptions - exceptions that extend `java.lang.RuntimeException` (aka **runtime exceptions**) and exceptions that extend `java.lang.Error` (aka **errors**).

Runtime exceptions and errors

Characterizing a situation as expected or unexpected is a design decision that depends on the business purpose of the method. One method may expect to receive a null argument and may work well if it gets a null argument, while another may not expect a null argument and may end up throwing a `NullPointerException` when it tries to access that null. In the second case, passing a null to that method would be considered a bug in the code, which must be identified and fixed during testing. Such exceptions that signify the presence of a bug in the code are categorized as **runtime exceptions**.

Here, the word runtime in `RuntimeException` does not imply that only exceptions that extend `RuntimeException` can be thrown at run time. All exceptions are thrown only when the program is executed, i.e., at run time. It refers to the fact that the developer comes to know of the occurrence of the situation that results in a `RuntimeException` only when the program is run, i.e., during run time. Had the programmer anticipated the occurrence of that situation during compile time, he would have fixed the code, and in which case the exception would not have been thrown upon running the program. For the same reason, runtime exceptions are usually not thrown explicitly using the `throw` statement. Indeed, why would you write the code to throw an exception if you don't even expect that situation to occur. The JVM throws runtime exceptions on its own when it encounters an unexpected situation. For example, if you try to access a null reference, the JVM will throw a `NullPointerException`, or if you try to access an array beyond its size, the JVM will throw an `ArrayIndexOutOfBoundsException`. It is possible to recover from runtime exceptions but ideally, since they indicate bugs in the code, you should not attempt to catch them and recover from them. A well written program should not cause the JVM to throw runtime exceptions.

The case of **errors** is similar to **runtime exceptions**. The difference is that errors are reserved for situations where the operation of the JVM itself is in jeopardy. For example, a badly written code may consume so much memory that there is no free memory left. Once that happens, the JVM may end up throwing `OutOfMemoryError`. Similarly, a bad recursion may cause `StackOverflowError` from which no recovery is possible. Errors signify serious issues in the interaction between the code and the JVM and are thrown exclusively by the JVM. It is never a good idea to throw them explicitly or to try to recover from them because the code will likely not work as expected anyway once the JVM starts throwing Errors.

Although the exam does not focus on the reason for categorizing exception between checked and unchecked exceptions, it is actually a very important topic to understand for a professional developer. You should also compare Java's exception handling with C#'s.

Identifying exceptions as checked or unchecked

Java follows a convention in naming exception classes. This convention is sometimes helpful in determining the kind of exception you are dealing with. The name of any class that extends **Error** ends with **Error** and the name of any class than extends **Exception** ends with **Exception**. For example, **OutOfMemoryError** and **StackOverflowError** are Errors while **IOException**, **SecurityException**, **IndexOutOfBoundsException** are Exceptions.

However, there is no way to distinguish between unchecked exceptions that extend **RuntimeException** and checked exceptions just by looking at their names. It is therefore, important to memorize the names of a few important runtime exception classes, namely - **NullPointerException**, **ArrayIndexOutOfBoundsException**, **ArithmetricException**, **ClassCastException**, and **SecurityException**.

14.2.2 Commonly used exception classes

The Java standard library contains a huge number of exception classes but the exam expects you to know about only a few of them, which I will cover now. These exception classes are important because you will encounter them quite often while working with Java.

I have omitted the package name from the classes below for brevity but note that all of the following exception classes belong to the **java.lang** package.

Common Exceptions that are usually thrown by the JVM

1. **ArithmetricException** extends **RuntimeException**

The JVM throws this exception when you try to divide a number by zero. Example :

```
public class X {  
    static int k = 0;  
    public static void main(String[] args){  
        k = 10/0; //ArithmetricException  
    }  
}
```

2. **IndexOutOfBoundsException** extends **RuntimeException**

This exception is a common superclass of exceptions that are thrown where an invalid index is used to access a value that supports indexed access.

For example, the JVM throws its subclass **ArrayIndexOutOfBoundsException** when

you attempt to access an array with an invalid index value such as a negative value or a value that is greater than the length (minus one, of course) of the array. Methods of String class throw another of its subclass `StringIndexOutOfBoundsException` when you try to access a character at an invalid index.

Example :

```
int[] ia = new int[]{ 1, 2, 3}; // ia is of length 3
System.out.println(ia[-1]); //ArrayIndexOutOfBoundsException
System.out.println(ia[3]); //ArrayIndexOutOfBoundsException

System.out.println("0123".charAt(4)); //StringIndexOutOfBoundsException
```

3. `NullPointerException` extends `RuntimeException`

The JVM throws this exception when you attempt to call a method or access a field using a reference variable that is pointing to null. Example :

```
String s = null;
System.out.println(s.length()); //NullPointerException because s is null
```

14.3 Use try and catch blocks

14.3.1 Creating a method that throws an exception

Now that you know about the throws clause and the types of exceptions, let us look at the rules for creating a method that throws an exception. Actually, there is just one rule - If there is a possibility of a **checked exception** getting thrown out of a method, then that exception or its superclass exception must be declared in the **throws clause** of the method. The following are examples of a few valid methods that illustrate this rule:

1. --

```
void process(int x) throws Exception{
    if(x == 2) throw new Exception(); //throws Exception only if x==2
    else return;
}
```

It doesn't matter whether the code throws an exception every time it runs or only some times. If there is a path of execution in which an exception will be thrown, the method must list that exception in the throws clause.

2. --

```
void process() { //no throws clause necessary
    if(someCondition) throw new RuntimeException();
    else throw new Error();
}
```

`RuntimeException` and `Error` are **unchecked exceptions** and are therefore, exempt from being declared in the throws clause. Declaring them in the throws clause is valid though.

3. --

```
void process() throws Exception{ //declaring Exception in the throws clause even
    though it is not thrown by the method body
    System.out.println("hello");
}
```

A method can declare any exception in its throws clause irrespective of whether that method actually throws that exception or not. It is sometimes useful to "future-proof" a method by declaring `Exception` in the throws clause if you believe that the method's implementation may change later. By declaring `Exception` in the throws clause, the users of the method will not have to change their code if the method actually starts throwing `Exception` or any of `Exception` subclasses later.

4. --

```
void process1() {
    try{
        if(someCondition) throw new Exception(); //will be caught by the catch
        block
        else return;
    }catch(Exception e){
        }
}
```

The requirement to list an exception in the throws clause is applicable only when an exception is thrown out of the method to the caller. If the code inside a method throws an exception but that exception is caught within the method itself, there is no need to declare it in the throws clause of the method. Remember that an exception can only be caught by a catch block and not by a finally block. Therefore, the throws clause is necessary in the following method:

```
void process2() throws Exception{
    try{
        if(someCondition) throw new Exception();
        else return;
    }finally {
        System.out.println("in finally"); //will be executed but the exception is
        not caught here
    }
}
```

Although not important for the exam, deciding which exception to throw and which to declare is an important matter.

Declaring exceptions

As shown in point number 8 above, declaring a broad exception class in the throws clause is an easy way to get rid of any compilation errors with the method if the method throws different kinds of exceptions from different parts of its code. However, this is considered a bad practice because this burdens the user of the method with dealing with a broad range of exceptions. On the other hand, listing specific exception classes individually restricts the future modifiability of a method because throwing new exceptions later will break other people's code. It is recommended to be balanced in your approach towards listing exceptions in the throws clause. You should try to be only as specific as is possible without compromising the modifiability of the method. For example, I/O related methods may encounter different kinds of issues while reading a file. It may not be possible to identify all such issues while writing the method. New issues may be discovered later and you may have to modify your method to accommodate those. Therefore, it is better to declare a common superclass `IOException` in the throws clause instead of individual subclasses such as `FileNotFoundException` or `EOFException`.

Throwing exceptions

You should always throw the most specific exception possible. For example, if you don't find the file while trying to open it, you should throw a `FileNotFoundException` instead of an `IOException` or `Exception`. By throwing the most specific exception, you give more information to the caller about the problem. This helps the caller in determining the most suitable resolution of the problem.

14.3.2 Invoking a method that throws an exception

The rules for invoking a method that throws an exception are similar to the ones for creating a method that throws an exception. As far as the compiler is concerned, there is no difference between using a `throw` statement to throw an exception and invoking a method that throws an exception to throw an exception. In both the cases, the compiler forces you to either declare the exception in your `throws` clause or handle the exception using a `try/catch` block. Of course, as discussed before, the compiler is only concerned with **checked exceptions**. The following example illustrates this point:

```
public class TestClass{  
    public static void message() throws Exception{  
        if(true) throw new Exception();  
        else return;  
    }  
    public static void main(String[] args) throws Exception {  
        message();  
    }  
}
```

```
}
```

In the above code, the method `message()` throws an `Exception` using the `throw` statement. Since `Exception` is a checked exception, the compiler forces it to be declared in the `throws` clause. On the other hand, instead of throwing an exception explicitly using the `throw` statement, `main` invokes `message`. But the compiler knows that invoking `message` may result in an `Exception` being thrown (because it is mentioned in the `throws` clause of `message`) and so the compiler forces `main` to declare `Exception` in its `throws` clause as well. The other option for `main` is, of course, to handle the exception itself:

```
public static void main(String[] args) { //no throws clause necessary
    try{
        message();
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

> If a method doesn't want to catch the exception then it must declare that exception (or its superclass) in its `throws` clause. This is no different from the rule that you have seen before while creating a method that throws an exception. For example, assuming that the method `message` throws `Exception`, here are valid throws clauses for a method that invokes `message` :

```
//declare the same exception class
public static void callMessage() throws Exception {
    message();
}

//declare a super class of the exception class thrown by message
public static void callMessage() throws Throwable {
    message();
}
```

Of course, `callMessage` is not limited to throwing just the exceptions declared in the `throws` clause of `message`. It can add its own exceptions to the `throws` clause irrespective of whether the code inside the method throws them or not.

To catch or to throw

The decision to catch an exception or to let it propagate to the caller depends on whether you can resolve the problem that resulted in the exception being thrown or not. Consider the following code for a method that computes simple interest:

```
public static double computeInterest(double p, double r, int t) throws Exception{
    if(t<0) throw new Exception("t must be > 0");
    else return p*r*t;
}
```

and the following code that uses the above method:

```
public static void main(String[] args){
    double interest = 0.0;
    try{
        computeInterest(100, 0.1, -1);
    }catch(Exception e){
    }
    System.out.println(interest);
}
```

Upon execution, the main method prints interest as `0.0` even though the `computeInterest` method did not really compute interest at all. It threw an exception because `t` was less than 0. However, as a user of the program, you won't know that there was actually a failure during the computation of interest.

While the `computeInterest` method did its job of telling the `main` method of a problem in computation by throwing an exception, the `main` method swepted this problem silently under the rug by using an empty `catch` block. This is called "**swallowing the exception**" and is a bad practice.

The purpose of a `catch` block is to resolve the problem and not to cover up the problem. By covering up the problem, the program keeps running but starts producing illogical results. Ideally, `main` should not have caught the exception but declared the exception in its `throws` clause because it is in no position to resolve the problem. It would have been appropriate for a program with GUI to catch the exception and ask the user to input valid arguments.

Sometimes, it becomes necessary to catch an exception even though no resolution is possible at that point. The right approach in such a case is to log the exception to the console so that the program can be easily debugged later by inspecting the logs.

```
public static void main(String[] args){
    double interest = 0.0;
    try{
        computeInterest(100, 0.1, -1);
    }catch(Exception e){
        e.printStackTrace();
        //or System.out.println(e);
    }
    System.out.println(interest);
}
```

14.4 Exercise

1. Create a method named `countVowels` that takes an array of characters as input and returns the number of vowels in the array. Furthermore, the method should throw a checked exception if the array contains the letter '`'x'`'.

2. Invoke the `countVowels` method from main in a loop and print its return value for each command line argument. Observe what happens in the following situations: there is no command line argument, there are multiple arguments, there are multiple arguments but the first argument contains an '`x`'. (Use String's `toCharArray` method to get an array of characters from the string.)
3. Ensure that your main method prints the number of vowels in other command line arguments even if one argument contains an '`x`'.
4. Pass null to the `countVowels` method and observe the output.
5. Modify `countVowels` method to throw an `IllegalArgumentException` if it is passed a `null`.
6. Modify `countVowels` method to return `-1`, if the input array is `null` and `0`, if the input array length is less than `10`. Do not use an `if` statement.
7. Create a method `int sum(int[] values, int start, int end)` that throws an `IllegalArgumentException` when passed an array of length 0, a `NullPointerException` when passed a `null`, and `ArrayIndexOutOfBoundsException` when `start` and `end` do not fall within the range of the given array. It should return the sum of the values in the array from `start` to `end` but must throw `Exception` when sum is 0.
8. Invoke the sum method created above from main. Wrap the call in an appropriate try/catch statement. Print two different messages on console depending upon whether the call results in a checked exception or an unchecked exception. Propagate all exceptions up the call chain.

Symbols

! operator, 112
() operator, 116
*!=, /=, %=, +=, -=, <<=, >>=, >>>=,
&=, ^=, |= operators, 113
+ operator, 116
 string, 120, 254
++,- operators, 108
+, -, *, / operators, 106
+=
 String concatenation, 113
+= operator
 string, 121
 string concatenation, 256
- operator, 107
->operator, 116
. operator, 116
.java extension, 66
/* */ usage, 65
/** usage, 66
== operator
 strings, 264
==, != operators, 109
?: operator, 112
[] operator, 116
% operator, 107
&, , operators, 111
&, |, ^ operators, 114
~operator, 114
\uxxxx format, 93

^ operator, 112
autoboxing, 76
>>, <<operators, 115
>>>operator, 115
<, >, <=, >= operators, 108
0x notation, 93

// usage, 65

A

abstraction, 42
access modifier
 private, 203
 protected, 203
 public, 204
access modifiers
 order, 204
accessibility modifiers, 203
add methods
 ArrayList, 187
address, 46
API, 5, 41
app, 4
application, 4
application programmer, 41
Application Programming Interface, 5, *see*
 API
ArithmeticException, 283
array, 174
 classes, 175
 cloning, 178

- covariance, 181
 - declaration, 174
 - indexing, 177
 - length, 178
 - members, 178
 - reification, 180
 - size, 178
 - uses, 181
 - array access operator, 116
 - array creation expression, 175
 - array initializer, 177
 - ArrayListException, 178, 283
 - ArrayList**
 - advantages & disadvantages, 194
 - constructors, 185
 - methods, 186
 - vs array, 194
 - ArrayList class, 182
 - Arrays class, 182
 - ASCII, 258
 - assignment operators, 112
 - associativity, 127
 - attribute, 43
 - autoboxing
 - method overloading, 233
 - return value, 224
 - Automatic memory management, 14
- B**
- bag, 183
 - binary number format, 93
 - binary numeric promotion, 123
 - binary operators
 - (), 116
 - *=, /=, %=, +=, -=, <<=, >>=, >>>=,
 - &=, ^=, |=, 113
 - +, 116
 - += string concatenation, 113
 - >, 116
 - ., 116
 - =, 113
 - ==, !=, 109
 - [], 116
 - &, |, 111
 - &, |, ^, 114
 - &&, ||, 110
 - ~, 112
 - >>, 115
 - >>, <<, 115
 - <, >, <=, >=, 108
 - instanceof, 116
 - bitwise operators, 114
 - block scope, 215
 - BODMAS, 125
 - boilerplate code
 - meaning, 161
 - break
 - in switch, 142
 - switch statement, 144
 - break statement
 - loops, 163
 - bytecode, 9, 13
- C**
- camel case, 54
 - case label, 141
 - Case labels, 143
 - cast, 95
 - assignment operators, 113
 - cast operator, 116
 - casting
 - primitives, 95
 - catch block, 277, 278
 - omission, 278
 - catch clause, 278
 - character encoding, 258
 - charAt method
 - String, 262
 - CharSequence interface, 254
 - charsets, 258
 - checked exceptions, 280
 - checked vs unchecked exceptions
 - rationale, 281
 - class, 43
 - execution, 36
 - meaning, 46, 63
 - members, 63
 - structure, 63
 - class library, 41
 - class packaging

- purpose, 70
 - class variables, 50
 - classpath, 71
 - clone
 - array, 178
 - Collection
 - interface, 183
 - collection
 - meaning, 183
 - Collections API, 183
 - command line arguments, 37
 - comments, 65
 - JavaDoc, 66
 - compilation
 - multiple source files, 71
 - specifying dependencies, 72
 - compile time binding, 242
 - compile time constant, 91
 - compile time constants
 - case labels, 143
 - compiler, 2
 - complement operator, 114
 - compound assignment operators, 113
 - concat method
 - String, 263
 - conditional operator
 - ternary operator, 138
 - conflicting imports, 61
 - constant expressions, 90
 - constructor
 - benefits, 210
 - chaining, 211
 - default, 208
 - definition, 207
 - invocation, 212
 - overloading, 211
 - contains method
 - ArrayList, 189
 - String, 264
 - continue statement
 - loops, 164
 - control variable, 151
 - conventions, 54
 - in Java, 54
 - loop variables, 54
 - package naming, 55
 - covariance
 - arrays, 181
 - covariant
 - return value, 225
- ## D
- dangling else, 135
 - data type
 - kinds, 80
 - meaning, 80
 - declaration
 - meaning, 86
 - default, 142
 - accessibility, 203
 - default block
 - switch statement, 144
 - default package, 58
 - definite assignment, 90
 - definition
 - meaning, 86
 - diamond operator, 186
 - do while loop
 - syntax, 153
 - dot operator, 116
 - dynamically typed, 14, 80
- ## E
- encapsulation
 - access modifiers, 204
 - meaning, 204
 - endsWith method
 - String, 264
 - enhanced for loop, 161
 - generics, 163
 - List, 162
 - syntax, 162
 - entry point, 46
 - equals method
 - String, 264
 - equals() method
 - strings, 265
 - equalsIgnoreCase method, 264
 - evaluation order, 128
 - exception

bubbling up, 275
 declaration in throws clause, 284
 identification, 283
 logging, 288
 propagation, 274
 swallowing, 288
 terminology, 275
 types, 280
 Exception handling, 14
 exception handling
 meaning, 274
 exceptional situations, 272
 exceptions
 checked, 280
 errors, 282
 runtime exceptions, 282
 thrown by JVM, 283
 unchecked, 275, 280
 exclusive or, 112
 executable, 36
 execution
 jar file, 74
 explicit narrowing, 95
 Expression statements, 158
 expression vs statement, 116
 expressions
 evaluation order, 128

F

Facelets , 13
 fall through behavior
 switch, 145
 final variable, 97
 finally block, 277, 278
 floating point data type, 81
 for loop, 154
 condition, 159
 expression statements, 158
 infinite, 159
 initialization, 158
 syntax, 156
 updation, 159
 format, 267
 FQCN, 59

G

Garbage Collection, 14
 Generics, 13
 generics, 185
 get method
 ArrayList, 189

H

heap space
 strings, 259
 hex notation, 93
 hexadecimal format
 char, 93
 hexadecimal number format, 93
 hiding, 214
 high level language, 3

I

IDE, 31
 identifier, 55
 if condition
 assignment, 136
 pre post increment, 137
 if else statement, 132
 if else vs switch, 142
 if else vs ternary operator, 138
 if statement, 132
 IllegalArgumentException, 276, 278
 implements, 45
 implicit narrowing, 95
 implicit variable, 243
 this, 200
 implicit widening conversion, 94
 import statement, 60
 conflicts, 61
 indexOf
 String, 262
 indexOf method
 ArrayList, 189
 IndexOutOfBoundsException, 283
 String methods, 261
 information hiding, 204
 instance
 meaning, 51

instance initializer, 206
instanceof operator, 116
instantiating a class, 206
integral data type, 81
Integrated Development Environment, 31
intern() method, 260
interpreter, 2
IOException, 283
isEmpty method
 ArrayList, 190
Iterable
 enhanced for loop, 162
iteration
 meaning, 150
Iterator
 enhanced for loop, 162

J

Jakarta, 11
jar
 command, 73, 74
 Main-Class, 74
 manifest, 73
jar format, 73
Java
 features, 12
java
 -classpath option, 71
 -cp option, 71
Java API, 15, 41
Java APIs, 9
Java Archive, 73
Java Community Process, 12
Java Development Kit, 8
Java EE, 11
Java Platform, 11
Java Runtime Environment, 9
Java SE, 11
Java Virtual Machine, 8
java.lang package, 76
java.util package, 76
javac, 70
 -classpath option, 72
 -d option, 71
JavaDoc, 66

JavaScript, 80
JCP, 12
JDK, 8, 32
JRE, 9, 32, 41
JVM, 8

K

keyword
 break, 57
 case, 57
 class, 56
 continue, 57
 default, 57
 do, 57
 else, 56
 extends, 57
 for, 57
 if, 56
 implements, 57
 import, 57
 new, 56
 package, 57
 private, 57
 protected, 57
 public, 57
 super, 57
 switch, 57
 this, 57
 while, 57
keywords, 55, 92
killing a program, 275

L

L postfix, 92
label, 168
labeled break, 168, 169
labeled continue, 168
Lambda Expressions, 13
lambda operator, 116
lastIndexOf
 String, 262
left associative, 127
length
 array, 178
length method

String, 261
 library, 4
 lifespan scope, 213
 lifespan scope vs visibility scope
 compilation, 216
 List interface, 183
 literal, 55
 meaning, 92
 literals, 55
 local variable type inference, 101
 local variables
 default values, 89
 logical negation operator, 112
 logical operators, 110
 long literal, 92
 loop, *see* enhanced for loop
 loop statements in java, 150
 loops
 comparison, 171
 termination using break, 164
 loosely typed, 80
 LVTI, 101

M

machine language, 2
 main method
 arguments, 37
 overloading, 37
 signature, 36
 termination, 38
 MANIFEST.MF, 73
 Math class, 77
 META-INF, 73
 method, 43
 body, 223
 meaning, 222
 method name, 222
 numeric promotion, 224
 parameters, 222
 return type, 222
 returning void, 223
 method chaining
 ArrayList, 190
 method overloading
 selection, 231

method overloading
 meaning, 229
 most specific, 231
 varargs, 233
 widening and autoboxing, 233
 method signature, 228
 missing else, 135
 missing return statement, 223

N

nested loop, 165
 nested loops
 breaking and continuing, 167
 new, 99
 no-args constructor, 210
 non short circuiting operators, 111
 null
 literal, 93
 meaning, 82
 vs void, 82
 NullPointerException, 275, 284
 string comparison, 265
 numeric data type, 81
 numeric literals
 rules, 92
 numeric promotion, 122
 compound assignment, 124
 return value, 224
 unary operator, 124

O

object
 meaning, 46
 Object class, 76
 Object Orientation, 40
 obscuring, 215
 octal number format, 93
 OOP, 39, 40
 interface, 40
 software component, 40
 oop
 physical component, 40
 OpenJDK, 32
 Operating System, 4
 operator precedence, 126

operators
 associativity, 127

P

package, 9, 58
 default, 58
package statement, 58
packaging
 jar, 73
parentheses
 operators, 128
pass by reference, 85
pass by value, 85, 235
PEDMAS, 125
post-increment, 118
postfix, 118
pre-increment, 118
prefix, 118
primitive data type, 80
primitive data types, 76
primitive types
 subtype relation, 232

primitive variable, 83
printf, 267
Procedural Programming, 42
program, 2, 117
program memory, 47
programming, 2
programming language, 2
property, 43

R

Random
 class, 249
Random class, 77
reference data type, 81
 size, 83
reference types
 declaration, 98
reference variable, 46, 83
 null, 49
reference variable vs primitive variable, 48
reification, 180
relational operators, 108
remove methods

ArrayList, 188
replace method
 String, 263
reserved words, 55
return null vs void, 82
right assiciative, 127
right associative operators, 113
runtime, 282
runtime exceptions, 282
runtime exceptions vs errors, 282
RuntimeException, 275
 important subclasses, 283

S

scope, 213
 lifespan, 215
 meaning, 213
 overlap, 218
scopes
 examples, 216
scripting languages, 13
Security Manager, 12
SecurityException, 283
seed, 250
Servlet, 13
set method
 ArrayList, 189
shadowing, 201
 method variable, 214
shallow copy
 array, 178
shift operators, 115
short circuiting operators, 110
signature, *see* method signature
single file source code program, 35
size method
 ArrayList, 190
sizeof, 83
source code, 32
split method
 String, 264
stack trace, 275
standard Java library, 41
startsWith method
 String, 264

statements, 116
static
 accessing, 241
 accessing directly, 242
 meaning, 50
 member declaration, 241
 non-oop, 51
static binding, 242
static method, 51
 instance fields, 244
 this, 243
static vs final, 50
statically typed, 14, 80
String
 class, 254
 constructors, 254
 methods, 261
string
 comparison, 264
 immutability, 261
 in Java, 254
 interning, 259
 manipulation, 261
 nomenclature, 254
 storage, 259
string concatenation, 116, 120, 254
string pool, 259
StringIndexOutOfBoundsException, 284
strongly typed, 14, 54, 80
Structured Programming, 14
substring method
 String, 262
Swing, 13
switch expression, 142
switch statement, 141
 case labels, 143
 fall through, 145
System class, 77
System.exit()
 in try/catch, 279

T

terminating a program, 275
ternary operator, 112, 138
 short circuiting, 140

 type, 139
then, 57
this
 in static method, 243
instance initializer, 207
keyword, 201
 reference, 200
throw statement, 275, 276
Throwable
 subclasses, 275
Throwable class, 275
throws clause, 273, 276
 purpose, 280
tight coupling, 204
toBinaryString method, 115
toLowerCase/UpperCase method
 String, 263
toString method
 ArrayList, 186
 string concatenation, 255
translator, 2
trim method
 String, 263
try block, 277
try statement, 277
try/catch approach
 benefits, 273
two's complement, 96
type, 32
type inference, 101
type unsafe, 184

U

unary decrement, 108, 118
unary increment, 108, 118
unary numeric promotion, 122
unary operators
 !, 112
 ++, --, 108
 -, 107
 ~, 114
unboxing, 77
unchecked exceptions, 275, 280
underscore
 in identifier, 56

in names, 54
in numeric literals, 92
uninitialized variables, 89
unsigned shift operator, 115
UTF-16, 258

V

var declaration, 101
varargs, 227
 method overloading, 233
variable
 declaration, 86
 final, 97
 initialization, 87
 naming rules, 88
 sizes, 83
 types, 82
variable scope
 loop blocks, 215
variables
 default values, 89
visibility
 instance variable, 213
 loop variable, 214

scope, 213
static variable, 214
visibility scope, 213
visibility vs accessibility, 214
void
 as return type, 82
 meaning, 82
 vs null, 82

W

web application, 4
Web Start, 13
while loop
 infinite, 152
 syntax, 150
while loop to for loop, 155
widening
 method overloading, 233
widening conversion, 94
WORA, 9
wrapper classes, 76

X

Xor, 112



Hanumant Deshmukh is a certified professional Java architect, author, and director of a software consultancy firm. 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 more than twenty years ago through which he offers training courses and learning material for various Java certification exams. He has authored several best selling books on Java.

Enthuware has been providing mock exams for various Java certification exams for the past twenty years. Enthuware mock exams are well known for the quality of their questions, detailed explanations, and customer support.

Oracle Java Certification Paths

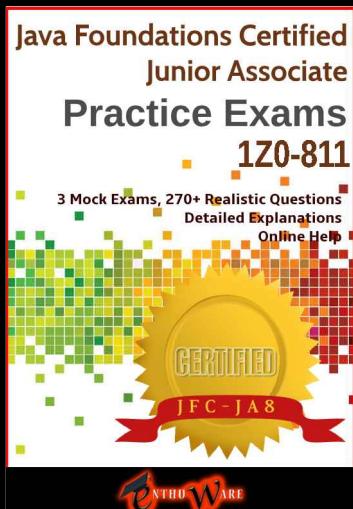
Oracle's Java 11 certification paths have changed significantly since October 2020. Oracle has eliminated the Associate level Java programming certification altogether. They have also scrapped the two part professional level certification exams (1Z0-815 and 1Z0-816), where the first part covered the basics and the second part covered the advanced topics, and replaced them with just one OCPJP exam that covers basic as well as advanced topics. Cost-wise, this is a good thing because now one has to pay the price of just one exam (1Z0-819) to get the OCP certification. However, this has also made the exam quite tough to pass because it covers several advanced topics such as Concurrency, NIO, JDBC, Modules, Localization, and Annotations. It will be very tough for entry level Java programmers to master these advanced topics.

Oracle Certified Foundations Associate Certification

To address this issue, Oracle has a Foundation level certification program called Oracle Certified Foundations Associate (OCFA). The name of the exam associated with this certification is Java Foundations Exam (exam code 1Z0-811). If you are a high schooler or a Java beginner, the 1Z0-811 exam is the best way to prove that you have learnt the basics of Java programming. This exam costs a lot less than the OCPJP exam and it focuses only on the concepts that Java beginners are expected to know. By preparing for the JFC Junior Associate exam, you will get to learn the fundamentals and you will also get a verifiable certification to show on your resume, which will help in your internship. You can then proceed to prepare for the more advanced OCPJP certification. Although this certification is based on Java 8, that doesn't matter much because most of the topics covered in this exam have not changed in new Java versions.

OCFA Fundamentals

I have written this study guide with Java beginners in mind, who have little or no Java knowledge. I take you right from Java installation to being fully prepared for the OCFA exam. Since the concepts covered in this certification will prepare the ground for learning more advanced concepts later, I have explained the basics in details without cutting any corners. If you follow this guide thoroughly, you will not only have covered all the topics required to pass the exam, but you will also have a solid grounding in Java programming.



Enthuware also offers realistic mock exams that simulate the real exam environment to help with the final leg of your preparation. You should use the mock exams after going through this book to check how well you are prepared for the real exam.

Features:

1. 270+ Questions with detailed explanations
2. Questions covering all exam objectives
3. Highly Customizable - Standard tests, Sectionwise/ Objectivewise tests, and Custom tests
4. Comparative Scores
5. Easy access from all your devices

Help and Support

Email : support@enthuware.com
Whatsapp: +1 980 272 1787
Forum : enthuware.com/forum