THE EXPERT'S VOICE® IN JAVA

# Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809

## A Comprehensive OCPJP 8 Certification Guide

—

S G Ganesh
Hari Kiran
Tushar Sharma

Apress®

**Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809: A Comprehensive OCPJP 8 Certification Guide**



ISBN-13 (pbk): 978-1-4842-1835-8

ISBN-13 (electronic): 978-1-4842-1836-5



Managing Director: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Developmental Editor: Douglas Pundick
Technical Reviewer: Vishal Biyani
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, James DeWolf, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Rita Fernando
Copy Editor: Lori Cavanaugh, Tiffany Taylor
Compositor: SPi Global
Indexer: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

# Contents at a Glance

# Contents

# About the Authors

**S G Ganesh** has Oracle Certified Professional, Java SE 8 Programmer (OCPJP 8) certification. He has 12+ years of working experience in the IT industry. He is currently a corporate trainer and independent consultant based in Bangalore, India. He quit his well-paying job to pursue his passion for sharing knowledge through corporate training and writing books. He worked for Siemens (Corporate Research and Technologies, Bangalore) in "Software Architecture and Development" team for 6+ years. Before Siemens, he worked in Hewlett-Packard's C++ compiler team, Bangalore for 4.5 years. He also has IEEE Software Engineering Certified Instructor (SECI) and IEEE Professional Software Engineering Master (PSEM) certifications. He has authored/co-authored many articles, research papers, and books. For more information, visit his LinkedIn page: http://bit.ly/sgganesh.

**Hari Kiran** is an independent consultant based in Bangalore, India. He has 12+ years of experience in the IT industry. He has worked for large IT companies including HCL Technologies, CSC, and Cisco Systems. He is an experienced Java developer. Throughout his career, he has worked on various technologies related to Java.

**Tushar Sharma** is currently a researcher at Athens University of Economics and Business. Earlier, he worked with Siemens Research and Technology Center, Bangalore, India for more than 7 years. His career interests include software design, refactoring, design smells, code and design quality, technical debt, design patterns, and change impact analysis. He has an MS (by research) degree in Computer Science from the Indian Institute of Technology-Madras (IIT-M), Chennai, India, where he specialized in design patterns and refactoring. He co-authored the book *Refactoring for Software Design Smells: Managing Technical Debt* published by Morgan Kaufmann in November 2014. He has Oracle Certified Professional, Java SE 7 Programmer (OCPJP 7) certification. He is an IEEE Senior Member. He can be reached at tusharsharma@ieee.org.

# About the Technical Reviewer

**Vishal Biyani** started his career working on Java 1.3 and he is excited that Java8 finally has streams and closures. Over the years he has performed various roles in technology projects for large as well as smaller companies. Biyani's current focus is helping organizations implement DevOps and continuous delivery principles. Biyani writes some of his thoughts at www.vishalbiyani.com and can be reached at vrbiyani@gmail.com.

# Acknowledgements

We would like to convey our sincere thanks to the entire Apress team for making this book possible.

Our first and foremost thanks go to our acquisitions editor Celestin Suresh John who played a key role from the conceptualization to the production stage of the book.

Our special thanks to Jeffrey Pepper for his support for our initial proposal to revise the earlier OCPJP 7 book, to coordinating editor Rita Fernando for her excellent coordination of this book project, and to developmental editor Douglas Pundick for improving the quality of the chapters. Thank you Jeff, Rita, and Douglas for your help!

A special thanks to our technical reviewer Vishal Biyani for his help in improving the quality of the book.

We convey our thanks to Anindya Bandopadhyay for sharing his feedback on improving the date and time API chapter in this book.

We would like to convey our sincere thanks to the following readers who reported errors in the earlier OCPJP 7 book: Sheila Weiss, Sebastiaan Heunis, John Doe, Steve Tarlton, Beto Montejo, Michael Klenk, Luca Aliberti, Mikael Strand, Jonathan S. Weissman, Bob, Gaël Jaffré, EpicWestern, John Stark, FlyTrap, Bruno Soares Bravo, Jaymoid, Denis Talochkin, Souvik Goswami, and Pawel K. We have corrected the reported mistakes in this book for a better reading/learning experience.

We take this opportunity to express our gratitude to friends and family for their continued support and encouragement.

—S G Ganesh, Hari Kiran, and Tushar Sharma

# Introduction

This book is a comprehensive guide to prepare for the OCPJP 8 exam. This book covers all the exam objectives for *Oracle Certified Professional, Java SE 8 Programmer* certification (1Z0-809 exam).

The book covers all of the exam topics for *Java SE 8 Programmer II* (1Z0-809) exam. The chapters and sections in this book map one-to-one to the exam objectives and subtopics. This one-to-one mapping between chapters and the exam objectives ensures that we cover only the topics to the required breadth and depth—no more, no less.

This book follows an example-driven approach to improve your reading and study experience. Additionally, in each chapter we use visual cues (such as notes, caution signs, and exam tips) to help you prepare for the OCPJP 8 exam.

## Prerequisites

Since the OCAJP 8 (a.k.a. *Java SE 8 Programmer* I/1Z0-808) exam is a prerequisite for the more comprehensive OCPJP 8 exam (1Z0-809), we assume that you are already familiar with the fundamentals of the language. We focus only on the OCPJP 8 exam objectives, assuming that you have working knowledge in Java.

## Target Audience

This book is for you:

- If you are a student or a programmer aspiring to crack the OCPJP 8 exam.
- If you want to learn new features added in Java 8 (especially on functional programming).
- If you're a trainer for OCPJP 8 exam. You can use this book as training material for OCPJP 8 exam preparation.
- If you just want to refresh your knowledge of Java programming or gain a better understanding of various Java APIs.

Please note, however, that this book is neither a tutorial for learning Java nor a comprehensive reference book for Java.

# Roadmap for Reading This Book

To get the most out of reading this book, we recommend you follow these steps:

**Step 0:** Make sure you have JDK 8 installed on your machine and you're able to compile and run Java programs.

**Step 1:** First read the FAQs in Chapter 1 and become familiar with the exam (you may want to skip irrelevant questions or questions for which you already know the answers).

**Step 2:** Check the exam topics given in the beginning of each chapter and mark the topics you're not familiar or comfortable with. Read the chapters or sections corresponding to the topics you've marked for preparation.

**Step 3:** Try out as many sample programs as possible when you read the chapters.

**Step 4:** Once you feel you are ready to take the exam, take the mock exam (Chapter 14). If you don't pass it, go back to the chapters in which you are weak, read them, and try out more programs relating to those topics. If you've prepared well, you should be able to pass the actual OCPJP 8 exam.

**Step 5:** Register for the exam and take the exam based on your performance in the mock tests. The day before taking the exam, read the summary sections given at the end of each chapter.

# On Coding Examples in This Book

All the programs in this book are self-contained programs (with necessary import statements). We've tested the coding examples in this book using Oracle's Java compiler in JDK 8. It is important that you use a Java compiler and a JVM that supports Java 8 for trying out the programs in this book.

Java is a platform-independent language, but there are certain features that are better explained with a specific platform. Since Windows is the most widely used OS today, some of the programming examples (especially the ones in the NIO.2 chapter) are written with the Windows OS in mind. You may require minor modifications to those programs to get them working under other OSs (Linux, MAC OS, etc.).

# Contact Us

In case of any queries, suggestions, or corrections, please feel free to contact us at sgganesh@gmail.com, gharikir@gmail.com, and tusharsharma@ieee.org.

CHAPTER 1



# The OCPJP 8 Exam: FAQ

The acronym OCPJP 8 exam stands for Java SE 8 Programmer II exam (exam number 1Z0-809). In this first chapter, we address the frequently asked questions (FAQs) that may come up when you are preparing for the OCPJP 8 exam.

## Overview

### FAQ 1. Can you provide details of the Java associate and professional exams for Java 8?

The OCAJP 8 exam (Oracle Certified Associate Java Programmer certification, exam number 1Z0-808) is mainly meant for entry-level Java developers. When you pass this exam, it demonstrates that you have a strong foundation in Java.

The OCPJP 8 exam (Oracle Certified Professional Java Programmer certification, exam number IZ0-809) is meant for professional Java developers. When you pass this exam, it demonstrates that you can use a wide range of core Java features (especially the ones added in Java 8) in your regular work.

# FAQ 2. Can you compare the specifications of the exams targeting OCAJP 8 and OCPJP 8 certifications?

Yes, see Table 1-1.

***Table 1-1.*** *Comparison of the Oracle Exams Leading to OCAJP8 and OCPJP8Certifications*

| Exam Number | 1Z0-808 | 1Z0-809 | 1Z0-810 | 1Z0-813 |
|---|---|---|---|---|
| Expertise Level | Beginner | Intermediate | Intermediate | Intermediate |
| Exam Name | Java SE 8 Programmer I | Java SE 8 Programmer II | Upgrade Java SE 7 to Java SE 8 OCP Programmer | Upgrade to Java SE 8 OCP (Java SE 6 and all prior versions) |
| Associated Certification (abbreviation) | Oracle Certified Associate, Java SE 8 Programmer (OCAJP 8) | Oracle Certified Professional, Java SE 8 Programmer (OCPJP 8) | Oracle Certified Professional, Java SE 8 Programmer (upgrade)(OCPJP 8) | Oracle Certified Professional, Java SE 8 Programmer (OCPJP 8) |
| Prerequisite Certification | None | Java SE 8 Programmer I (OCAJP8) | Java SE 7 Programmer II (OCPJP 7) | Oracle Certified Professional Java SE 6 Programmer and all prior versions (OCPJP 6 and earlier versions) |
| Exam Duration | 2 hrs 30 minutes (150 mins) | 2 hrs 30 minutes (150 mins) | 2 hrs 30 minutes (150 mins) | 2 hrs 10 minutes (130 mins) |
| Number of Questions | 77Questions | 85 Questions | 81 Questions | 60 Questions |
| Pass Percentage | 65% | 65% | 65% | 63% |
| Cost | ~ USD 245 | ~USD 245 | ~USD 245 | ~USD 245 |
| Exam Topics | Java Basics<br>Working With Java Data Types<br>Using Operators and Decision Constructs<br>Creating and Using Arrays<br>Using Loop Constructs<br>Working with Methods and Encapsulation<br>Working with Inheritance<br>Handling Exceptions<br>Working with Selected classes from the Java API | Java Class Design<br>Advanced Java Class Design<br>Generics and Collections<br>Lambda Built-in Functional Interfaces<br>Java Stream API<br>Exceptions and Assertions<br>Use Java SE 8 Date/ Time API<br>Java I/O Fundamentals<br>Java File I/O (NIO.2)<br>Java Concurrency<br>Building Database Applications with JDBC<br>Localization | Lambda Expressions<br>Using Built-in Lambda Types<br>Filtering Collections with Lambdas<br>Collection Operations with Lambda<br>Parallel Streams<br>Lambda Cookbook<br>Method Enhancements<br>Use Java SE 8 Date/ Time API | Language Enhancements<br>Concurrency<br>Localization<br>Java File I/O (NIO.2)<br>Lambda<br>Java Collections<br>Java Streams |

***Notes***

- In the Cost row, the given USD cost of the exams is approximate as actual cost varies with currency of the country in which you take the exam: $245 in US, £155 in UK, Rs. 9604 in India, etc.
- The Exam Topics row lists only the top-level topics. For sub-topics, please check the Oracle's web pages for these exams.
- The details provided here are as on November 1, 2015. Please check the Oracle website for any updates on the exam details.

# Details About the Exam

## FAQ 3. OCAJP 8 certification is a prerequisite for OCPJP 8 certification. Does that mean that I have to take the OCAJP8 exam before I can take the OCPJP8 exam?

No, requirements for certification may be met in any order. You may take the OCPJP 8 exam before you take the OCAJP 8 exam, but you will not be granted OCPJP 8 certification until you have passed both 1Z0-808and 1Z0-809 exams.

## FAQ 4. How does the OCPJP 8 exam differ from the older OCPJP 7 exam?

When compared to the exam topics in OCPJP 7 exam, the OCPJP 8 exam is updated with topics added in the Java SE 8 release: lambda functions, Java built-in functional interfaces, stream API (including parallel streams), and date and time API, and other changes to the Java library.

## FAQ 5.Should I take the OCPJP8 exam or earlier versions such as the OCPJP 7 exam?

Although you can still take exams for older certifications such as OCPJP 7, OCPJP 8 is the best professional credential to have because it is validated against the latest Java SE 8 release.

## FAQ 6. What kinds of questions are asked in the OCPJP 8exam?

Some questions on the OCPJP 8 exam test your conceptual knowledge without reference to a specific program or code segment. But most of the questions are programming questions of the following types:

- Given a program or code segment, what is the output or expected behavior?
- Which option(s) would compile without errors or give the desired output?
- Which option(s) constitute the correct usage of a given API (in particular, newly introduced ones such as stream and date/time APIs)?

All questions are multiple choice. Most of them present four or five options, but some have six or seven options. Many questions are designed to have a set of multiple correct answers. Such questions clearly mention the number of options you need to select.

Exam questions are not constrained to be exclusively from the topics on the exam syllabus. You might, for example, get questions on Java fundamentals (from OCAJP syllabus) concerning the basics of exception handling and using wrapper types. You might also get questions on topics related to those on the exam syllabus but not specified in it. For example, in the exam, you may get a question on `java.util.function.BinaryOperator` interface though the "Java Built-in Functional Interfaces" exam topic does not explicitly mention this interface.

A given question is not constrained to test only one topic. Some questions are designed to test multiple topics with a single question. For instance, you may find a question on parallel streams that makes use of built-in functional interfaces and lambda expressions.

## FAQ 7. What does the OCPJP 8 exam test for?

The OCPJP 8 exam tests your understanding of the Java language features and APIs that are essential for developing real-world programs. The exam focuses on the following areas:

- *Language concepts that are useful for problem solving*: The exam tests not only your knowledge of how language features work, but also covers your grasp of the nitty-gritty and corner cases of language features. For example, you need to understand not only the generics feature in Java but also problems associated with type-erasure, mixing legacy containers with generic containers, and so on.
- *Java APIs*: The exam tests your familiarity with using the Java class library, as well as unusual aspects or corner cases, such as the following:
  - What is the binary equivalent for `java.util.function.Supplier`? (Answer: Since a `Supplier` does not take any arguments, there is no binary equivalent for the `Supplier` interface).
  - What will happen if you try to use a stream more than once? (Answer: Once a terminal operation is called on a stream, it is considered used or closed; any attempt at reusing the stream will result in throwing an `IllegalStateException`.)
- *Underlying concepts*: For example, the exam might test your understanding of how serialization works, the differences between overloading and overriding, how autoboxing and unboxing work in relation to generics, different kinds of liveness problems with threads, how parallel streams internally use the fork/join framework, etc.

Although the exam does not test memory skills, some questions presume rote knowledge of key elements, such as the following:

- The name of the abstract methods provided in the key functional interfaces in `java.util.function` package (`"test"` method for `Predicate`, `"accept"` method for `Consumer`, `"apply"` method for `Function`, and `"get"` method for `Supplier` interface).
- In the `java.util.stream.Stream` interface and its primitive type versions, you need to remember the name of the commonly used intermediate operations and terminal operations.

## FAQ 8. I've been a Java programmer for the last five years. Do I have to prepare for the OCPJP 8 exam?

*Short answer:* It's good that you have work experience, but you still need to prepare for the OCPJP 8 exam.

*Long answer:* No matter how much real-world programming experience you might have, there are two reasons you should prepare for this exam to improve your chances of passing it:

- *You may not have been exposed to certain topics on the exam.* Java is vast, and you might not have had occasion to work on every topic covered in the exam. For example, you may not be familiar with localization if you have never dealt with the locale aspects of the applications you were engaged with. Or your work might not have required you to use JDBC. Or you've always worked on single-threaded programs, so multithreaded programming might be new to you. Moreover, OCPJP8 emphasizes Java 8, and you might not have been exposed yet to such Java 8 topics as lambda expressions, sequential and parallel streams, date and time API, and built-in functional interfaces.
- *You may not remember the unusual aspects or corner cases.* No matter how experienced you are, there is always an element of surprise involved when you program. The OCPJP8 exam tests not just your knowledge and skills in respect of regular features, but also your understanding of unusual aspects or corner cases, such as the behavior of multithreaded code and the use of generics when both overloading and overriding are involved. So you have to bone up on pathological cases that you rarely encounter in your work.

## FAQ 9. How do I prepare for the OCPJP 8 exam?

Study this book. In addition,

- *Code, code, code!* Write lots and lots of small programs, experiment with them, and learn from your mistakes.
- *Read, read, read!* Read this book and the tutorial and reference resources on Oracle's site, especially
  - *Oracle's free online Java tutorials*: Access the Java tutorial at `http://docs.oracle.com/javase/tutorial/`.
  - Oracle's Java 8 central: You can download the latest Java SDK, gets links to access the Java SE community, read free technical articles on Java 8 from this page:

    `http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`
  - *Java documentation*: The Java API documentation is a mine of information. This documentation is available online (see `http://docs.oracle.com/javase/8/docs/api/`) and is shipped as part of the Java SDK. If you don't have immediate Internet access, you may find javac's `-Xprint` option handy. For example, to print the textual representation of `String` class, type the fully qualified name, as in

    ```
    javac -Xprint java.lang.String
    ```

This will print the list of members in `String` class on the console.

- *Read, code, read, code!* Cycle back and forth between your reading and coding so that your book knowledge and its practical application are mutually reinforcing. This way, you'll not just know a concept, but you'll also *understand* it.
- *Focus most on the topics you're least comfortable with.* Grade yourself on each of the topics in OCPJP 8 exam on an ascending scale from 1 to 10. Do remedial preparation in all the topics for which you rate yourself 8 or less.

## FAQ 10. How do I know when I'm ready to take the OCPJP 8 exam?

Take the mock exam given in Chapter 14 under actual exam conditions: stick to the 2.5-hour time limit; don't take any breaks and don't refer any books or websites. If you score 65% or above (which is the passing score for 1Z0-809 exam), you are likely to pass the actual exam.

# Taking the Exam

## FAQ 11. What are my options to register for the exam?

You have three registration options for the OCPJP 8 exam:

- Register and pay at the Pearson VUE website.
- Buy an exam voucher from Oracle and then register yourself in Pearson VUE website.
- Register and pay at the Oracle Testing Center (OTC), if you have one in your region.

## FAQ 12. How do I register for the exam, schedule a day and time for taking the exam, and appear for the exam?

***Option 1: Register and pay on the Pearson VUE website using the following steps:***

**Step 1.** Go to `www.pearsonvue.com/oracle/` (you will be directed here if you click the first option from Oracle Certification page). Click on "Schedule online" in "Schedule an exam" section.

**Step 2.** Select "Sign In." Click on "proctored" in the "what type of exam you are planning to take" section. Select this exam as `"Information Technology (IT)"` ➤ `"Oracle"` ➤ `"Proctored"`. Then you'll be asked to sign in.

**Step 3.** Log in to your web account on the Pearson site. If you don't have one, create one; you will get the user name and password by the e-mail you provide. When you log in first time, you need to change your password and set security questions and their answers. When you are done with this, you're ready to schedule your exam.

**Step 4.** Once logged in, you'll get the list of Oracle exams to select from. Select the following exam:

- 1Z0-809, Java SE 8Programmer II(aka OCPJP 8 exam)

These exams are in English (You can choose another language if you wish and if it is available in the list). This page will also show you the cost of the exam. Click `Next`.

**Step 5.** Now you need to select your test location. Choose `Country` ➤ `City` ➤ `State/Province`, and you'll be shown test locations close to you. Each center will have an icon for information: click it for the address and directions. Select up to four centers near your location and click `Next`.

**Step 6.** Select a test center and select a date and time for appointments. The page will indicate the available dates and time slots; choose the one most convenient for you. If you have an exam voucher or Oracle University coupon or Oracle promotion code, enter it here.

**Step 7.** Select from the available payment options (the usual way is to pay using your credit card) and pay your exam fees. Make sure that you have selected the right exam, appropriate test center, and date/time before paying the fees.

**Step 8.** Done! You will get an appointment confirmation payment receipt by e-mail.

***Option 2: Buy an exam voucher from Oracle and register on the Pearson VUE website.***

You can buy a generic exam voucher from Oracle and use it at the Pearson site. It costs US$245 if you are living in the United States and is denominated in an appropriate currency if you live elsewhere. To buy the voucher from Oracle, select "OU Java, Solaris, and other Sun Technology Exam eVoucher." You will be asked to create an Oracle account if you do not have one. Once the account is created, confirm customer type, customer contact information, and pay. Once you pay the fees, you can use the eVoucher at the Pearson VUE site.

***Option 3: Register and pay online to take the exam in person at an Oracle Testing Center (OTC).***

You can choose this option if a physical exam session is scheduled in your vicinity. It costs US$245 or the local equivalent.

## FAQ 13. What are the key things I need to remember before taking the exam and on the day of exam?

**Before the exam day:**

- You'll get e-mail from Pearson confirming your appointment and payment. Check the details on what you should bring when you go to the exam center. Note that you'll need at least two photo IDs.
- Before the exam, you'll get a call from the Pearson exam center where you've booked your appointment.

**On the exam day:**

- Go to the exam center at least 30 minutes before the exam starts. Your exam center will have lockers for storing your belongings.
- Show your exam schedule information and IDs and then complete the exam formalities, such as signing the documents.
- You'll be taken to a computer in the exam room and will log into the exam-taking software.

**Taking the exam:**

- You will see the following on the exam-taking software screen:
  - A timer ticking in one corner showing the time left
  - The current question number you are attempting
  - A check box to select if you want to review the question later
  - The button (labeled "Review") for going to a review screen where you can revisit the questions before completing the exam.
- Once you start, you'll get questions displayed one by one. You can choose the answers by selecting them in the check box. If you are unsure of an answer, select the Mark button so that you can revisit it at any point during the exam. You can also right-click on an option to strike-through that option (useful for eliminating incorrect options).
- You may not consult any person or print or electronic materials or programs during the exam.

**After the exam:**

- Once you're done with the exam, you will not be immediately shown the results. You have to log in to Oracle's CertView website (`https://education.oracle.com/certview.html`) to see the exam results.
- Irrespective of passing or failing the exam, *topics* from questions you've answered incorrectly will be supplied with your score.
- If you've passed the OCPJP 8 exam *and* you've also satisfied the applicable prerequisites for certification (e.g., OCAJP 8 certification as the prerequisite of OCPJP 8 certification via the 1Z0-809 exam), a printable certificate will be e-mailed to you.
- If you failed the exam, you may register and pay again to retake it after a 14-day waiting period.

CHAPTER 2

# Java Class Design

## Certification Objectives

**Implement encapsulation**

**Implement inheritance including visibility modifiers and composition**

**Implement polymorphism**

**Override hashCode, equals, and toString methods from Object class**

**Create and use singleton classes and immutable classes**

**Develop code that uses static keyword on initialize blocks, variables, methods, and classes**

Object-Orientation (OO) is the core of the most mainstream programming languages today. To create high-quality designs and software, it is important to get a good grasp of object oriented concepts. This chapter on class design and the next chapter on advanced class design provides you a firm foundation for creating quality designs in Java.

Since OCAJP 8 is a pre-requisite for the OCPJP 8 exam, we assume that you are familiar with basic concepts such as methods, fields, and how to define a constructor. Hence, in this chapter, we start directly discussing OCPJP 8 exam topics. In the first section, we discuss how to enforce encapsulation using access specifiers, and how to implement inheritance and polymorphism. In the next section, we delve into details on overriding methods in the `Object` class, define singleton and immutable classes, and analyze different ways of using the `static` keyword.

# Encapsulation

## Certification Objective

**Implement encapsulation**

Structured programming *decomposes* the program's functionality into various procedures *(functions)*, without much concern about the data each procedure can work with. Functions are free to operate and modify the (usually global and unprotected) data.

In Object Oriented Programming (OOP), data and associated behavior forms a single unit, which is referred to as a *class*. The term *encapsulation* refers to combining data and associated functions as a single unit. For example, in a `Circle` class, `radius` and `center` are defined as *private fields*. Now you can add methods such as `draw()` and `fillColor()` along with fields `radius` and `center`, since the fields and methods are closely related to each other. All the data (fields) required for the methods in the class are available inside the class itself. In other words, the class *encapsulates* its fields and methods together.

# Access Modifiers

## Certification Objective

**Implement inheritance including visibility modifiers and composition**

Access modifiers determine the level of visibility for a Java entity (a class, method, or field). *Access modifiers enable you to enforce effective encapsulation.* If all member variables of a class can be accessed from anywhere, then there is no point putting these variables in a class and no purpose in encapsulating data and behavior together in a class.

The OCPJP 8 exam includes both direct questions on access modifiers and indirect questions that require an underlying knowledge of access modifiers. Hence it is important to understand the various access modifiers supported in Java.

Java supports four types of access modifiers:

- Public
- Private
- Protected
- Default (no access modifier specified)

To illustrate the four types of access modifiers, let's assume that you have the following classes in a drawing application: `Shape, Circle, Circles`, and `Canvas` classes. The `Canvas` class is in `appcanvas` package and the other three classes are in `graphicshape` package (see Listing 2-1).

***Listing 2-1.*** Shape.java, Circle.java, Circles.java, and Canvas.java

```
// Shape.java
package graphicshape;

class Shape {
        protected int color;
}

// Circle.java
package graphicshape;

import graphicshape.Shape;

public class Circle extends Shape {
        private int radius;     // private field
        public void area() {    // public method
                // access to private field radius inside the class:
                System.out.println("area: " + 3.14 * radius * radius);
        }
        // The fillColor method has default access
        void fillColor() {
                //access to protected field, in subclass:
                System.out.println("color: " + color);
        }
}
```

```
// Circles.java
package graphicshape;

class Circles {
        void getArea() {
                Circle circle = new Circle();
                // call to public method area() within package:
                circle.area();
                // calling fillColor() with default access within package:
                circle.fillColor();
        }
}

// Canvas.java
package appcanvas;
import graphicshape.Circle;

class Canvas {
        void getArea() {
                Circle  circle = new Circle();
                circle.area();   // call to public method area(), outside package
        }
}
```

## Public Access Modifier

The public access modifier is the most liberal one. If a class or its members are declared as *public*, they can be accessed from any other class regardless of the package boundary. It is comparable to a public place in the real world, such as a company cafeteria that all employees can use irrespective of their department. As shown in Listing 2-1, the public method `area()` in `Circle` class is accessible within the same package, as well as outside of the package (in the `Canvas` class).

---


A *public method* in a class is accessible to the outside world only if the class is declared as public. If the class does not specify any access modifier (i.e., it has default access), then the public method is accessible only within the containing package.

---

## Private Access Modifier

The private access modifier is the most stringent access modifier. A private class member cannot be accessed from outside the class; only members of the same class can access these private members. It's comparable to a safe deposit box room in a bank, which can only be accessed by a set of authorized personnel and safe deposit box owners. In Listing 2-1, the private field radius of the `Circle` class is accessible only inside the `Circle` class and not in any other class regardless of the enclosing package.

## Protected and Default Access Modifiers

Protected and default access modifiers are quite similar to each other. If a member method or field is declared as protected or default, then the method or field can be accessed within the package. Note that there is no explicit keyword to provide default access; in fact, when no access modifier is specified, the member has default access. Also, note that default access is also known as package-protected access. Protected and default accesses are comparable to the situation in an office where a conference room is accessible only to one department.

What is the difference between protected and default access? One significant difference between these two access modifiers arises when we talk about a subclass belonging to another package than its superclass. In this case, protected members are accessible in the subclass, whereas default members are not.

---

 A class (or interface) cannot be declared as private or protected. Furthermore, member methods or fields of an interface cannot be declared as private or protected.

---

In Listing 2-1, the protected field `color` is accessed in the class `Circle` and the default method `fillColor()` is called from the class `Circles`.

The visibility offered by various access modifiers is summarized in Table 2-1.

***Table 2-1.*** *Access Modifiers and Their Visibility*

| Access modifiers/ accessibility | Within the same class | Subclass inside the package | Subclass outside the package | Other class inside the package | Other class outside the package |
|---|---|---|---|---|---|
| Public | Yes | Yes | Yes | Yes | Yes |
| Private | Yes | No | No | No | No |
| Protected | Yes | Yes | Yes | Yes | No |
| Default | Yes | Yes | No | Yes | No |

# Inheritance

*Inheritance* is a reusability mechanism in object-oriented programming. With inheritance, the common properties of various objects are exploited to form relationships with each other. The abstract and common properties are provided in the *superclass*, which is available to the more specialized *subclasses*. For example, a color printer and a black-and-white printer are kinds of a printer (*single inheritance*); an all-in-one printer is a printer, scanner, and photocopier (*multiple inheritance).*

Why is inheritance a powerful feature? Because it supports modeling classes in a hierarchy, and such hierarchical models are easy to understand. For example, you can logically categorize vehicles as two-wheelers, three-wheelers, four-wheelers, and so on. In the four-wheelers category, there are cars, vans, buses, and trucks. In the cars category, there are hatchbacks, sedans, and SUVs. When you categorize hierarchically, it becomes easy to understand, model, and write programs.

Consider a simple example used in earlier sections: class `Shape` is a base class and `Circle` is a derived class. In other words, a `Circle` is a `Shape`; similarly, a `Square` is a `Shape`. Therefore, an inheritance relationship can be referred to as an *IS-A relationship*.

In the Java library, you can see extensive use of inheritance. Figure 2-1 shows a partial inheritance hierarchy from `java.lang` library. The `Number` class abstracts various numerical (reference) types such as `Byte`, `Integer`, `Float`, `Double`, `Short`, and `BigDecimal`.

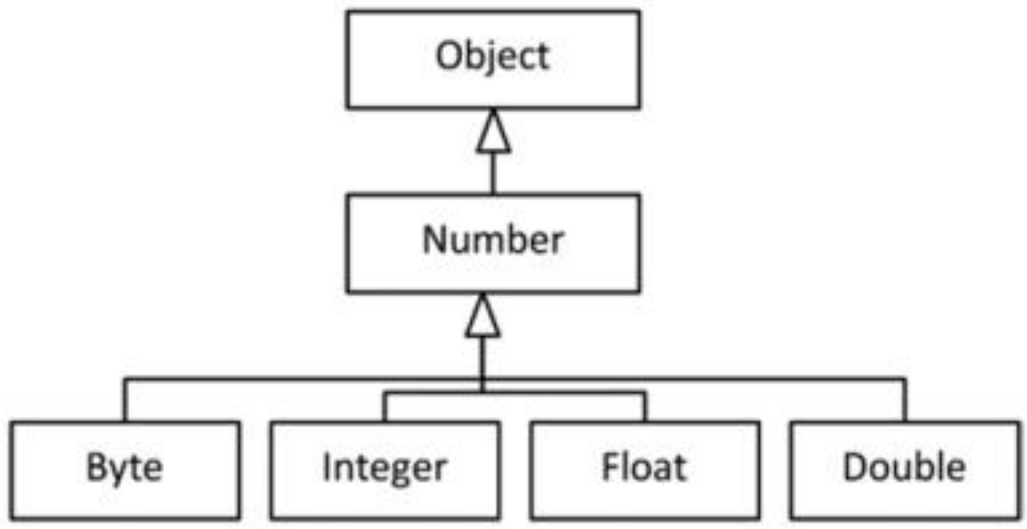


***Figure 2-1.*** *A partial inheritance hierarchy in java.lang package*

The class `Number` has many common methods that are inherited by the derived classes. The derived classes do not have to implement the common methods implemented by the `Number` class. Also, you can supply a derived type where the base type is expected. For instance, a `Byte` is a `Number`, which means you can provide an object of `Byte` where an object of `Number` is expected. You can write general-purpose methods (or algorithms) when you write methods for the base type. Listing 2-2 shows a simple example.

***Listing 2-2.*** TestNumber.java

```
// Illustrates how abstracting different kinds of numbers in a Number hierarchy
// becomes useful in practice
public class TestNumber {
        // take an array of numbers and sum them up
        public static double sum(Number []nums) {
                double sum = 0.0;
                for(Number num : nums) {
                        sum += num.doubleValue();
                }
                return sum;
        }

        public static void main(String []s) {
                // create a Number array
                Number []nums = new Number[4];
                // assign derived class objects
                nums[0] = new Byte((byte)10);
                nums[1] = new Integer(10);
                nums[2] = new Float(10.0f);
                nums[3] = new Double(10.0f);
                // pass the Number array to sum and print the result
                System.out.println("The sum of numbers is: " + sum(nums));
        }
}
```

This program prints

```
The sum of numbers is: 40.0
```

In the `main()` method, you declare `nums` as a `Number[]`. A `Number` reference can hold any of its derived type objects. You are creating objects of type `Byte`, `Integer`, `Float`, and `Double` with initial value 10; the `nums` array holds these elements. (Note that you needed an explicit cast in `new Byte((byte) 10)` instead of plain `Byte(10)` because `Byte` takes a `byte` argument and 10 is an `int`.)

The `sum` method takes a `Number[]` and returns the sum of the `Number` elements. The `double` type can hold the largest range of values, so you use `double` as the return type of the `sum` method. `Number` has a `doubleValue` method and this method returns the value held by the `Number` as a `double` value. The `for` loop traverses the array, adds the `double` values, and then returns the `sum` once you're done.

As you can see, the `sum()` method is a general method that can handle any `Number[]`. A similar example can be given from the Java standard library where `java.util.Arrays` class has a static method `binarySearch()`:

```
static int binarySearch(Object[] a, Object key, Comparator c)
```

This method searches a given key (an `Object` type) in the given array of `Objects`. `Comparator` is an interface declaring the `equals` and `compare` methods. You can use `binarySearch` for objects of any class type that implements this `Comparator` interface. As you can see, inheritance is a powerful and useful feature for writing general-purpose methods.

# Polymorphism

## Certification Objective

### Implement polymorphism

The Greek roots of the term *polymorphism* refer to the "several forms" of an entity. In the real world, every message you communicate has a context. Depending on the context, the meaning of the message may change and so may the response to the message. Similarly in OOP, a message can be interpreted in multiple ways (polymorphism), depending on the object.

Polymorphism can be of two forms: *dynamic* and *static*.

- When different forms of a single entity are resolved during runtime (*late binding*), such polymorphism is called *dynamic polymorphism*. In the previous section on inheritance, we discussed overriding. Overriding is an example of *runtime polymorphism*.
- When different forms of a single entity are resolved at compile time (*early binding*), such polymorphism is called *static polymorphism*. *Function overloading* is an example of static polymorphism, and let us explore it now.

Please note that abstract methods use runtime polymorphism. We discuss abstract methods in interfaces and abstract classes in the next chapter (Chapter 3 – Advanced Class Design).

# Runtime Polymorphism

You just learned that a base class reference can refer to a derived class object. You can invoke methods from the base class reference; however, the actual method invocation depends on the dynamic type of the object pointed to by the base class reference. The type of the base class reference is known as the *static type* of the object and the actual object pointed by the reference at runtime is known as the *dynamic type* of the object.

When the compiler sees the invocation of a method from a base class reference and if the method is an overridable method (a nonstatic and nonfinal method), the compiler defers determining the exact method to be called to runtime (late binding). At runtime, based on the actual dynamic type of the object, an appropriate method is invoked. This mechanism is known as *dynamic method resolution* or *dynamic method invocation*.

## Runtime Polymorphism: An Example

Consider that you have the `area()` method in `Shape` class. Depending on the derived class—`Circle` or `Square`, for example—the `area()` method will be implemented differently, as shown in Listing 2-3.

***Listing 2-3.*** TestShape.java

```
class Shape {
        public double area() { return 0; } // default implementation
        // other members
}
class Circle extends Shape {
        private int radius;
        public Circle(int r) { radius = r; }
        // other constructors
        public double area() {return Math.PI * radius * radius; }
        // other declarations
}

class Square extends Shape {
        private int side;
        public Square(int a) { side = a; }
        public double area() { return side * side; }
        // other declarations
}

public class TestShape {
        public static void main(String []args) {
                Shape shape1 = new Circle(10);
                System.out.println(shape1.area());
                Shape shape2 = new Square(10);
                System.out.println(shape2.area());
        }
}
```

This program prints

```
314.1592653589793
100.0
```

This program illustrates how the `area()` method is called based on the dynamic type of the Shape. In this code, the statement `shape1.area();` calls the `Circle's area()` method while the statement `shape2.area();` calls `Square's area()` method and hence the result.

Now, let's ask a more fundamental question: Why do you need to override methods? In OOP, the fundamental idea in inheritance is to provide a default or common functionality in the base class; the derived classes are expected to provide more specific functionality. In this Shape base class and the `Circle` and `Square` derived classes, the Shape provided the default implementation of the `area()` method. The derived classes of `Circle` and `Square` defined their version of the `area()` method that overrides the base class `area()` method. So, depending on the type of the derived object you create, from base class reference, calls to `area()` method will be resolved to the correct method. Overriding (i.e., runtime polymorphism) is a simple yet powerful idea for extending functionality.

Let us now discuss compile-time polymorphism (overloading). After that, we will immediately return back to this topic of runtime polymorphism to discuss more topics such as how to deal with visibility modifiers when overriding and choosing between composition and inheritance.

# Method Overloading

In a class, how many methods can you define with the same name? Many! In Java, you can define multiple methods with the same name, provided the argument lists differ from each other. In other words, if you provide different types of arguments, different numbers of arguments, or both, then you can define multiple methods with the same name. This feature is called *method overloading*. The compiler will resolve the call to a correct method depending on the actual number and/or types of the passed parameters.

Let's implement a method in the `Circle` class called `fillColor()` that fills a circle object with different colors. When you specify a color, you need use a coloring scheme, and let us consider two schemes - RGB scheme and HSB scheme.

1. When you represent a color by combining Red, Green, and Blue color components, it is known as RGB scheme. By convention, each of the color values is typically given in the range 0 to 255.

2. When you represent a color by combining Hue, Saturation, and Brightness values, it is known as HSB scheme. By convention, each of the values is typically given in the range 0.0 to 1.0.

Since RGB values are integer values and HSB values are floating point values, how about supporting both these schemes for calling `fillColor()` method?

```
class Circle {
        // other members
        public void fillColor (int red, int green, int blue) {
                /* color the circle using RGB color values - actual code elided */
        }

        public void fillColor (float hue, float saturation, float brightness) {
                /* color the circle using HSB values - actual code elided */
        }
}
```

As you can see, both `fillColor()` methods have exactly the same name and both take three arguments; however, the argument types are different. Based on the type of arguments used while calling `fillColor()` method on `Circle`, the compiler will decide exactly which method to call. For instance, consider following method calls:

```
Circle c1 = new Circle(10, 20, 10);
c1.fillColor(0, 255, 255);

Circle c2 = new Circle(50, 100, 5);
c2.fillColor(0.5f, 0.5f, 1.0f);
```

In this code, for the `c1` object, the call to `fillColor()` has integer arguments 0, 255, and 255. Hence, the compiler resolves this call to the method `fillColor(int red, int green, int blue)`. For the `c2` object, the call to `fillColor()` has arguments 0.5f, 0.5f, and 1.0f; hence it resolves the call to `fillColor(float hue, float saturation, float brightness)`.

In the above example, method `fillColor()` is an overloaded method. The method has same name and the same number of arguments, but the types of the arguments differ. It is also possible to overload methods with different numbers of arguments.

Such overloaded methods are useful for avoiding repeating the same code in different functions. Let's look at a simple example in Listing 2-4.

***Listing 2-4.*** HappyBirthday.java

```
class HappyBirthday {
        // overloaded wish method with String as an argument
        public static void wish(String name) {
                System.out.println("Happy birthday " + name + "!");
        }

        // overloaded wish method with no arguments;
        // this method in turn invokes wish(String) method
        public static void wish() {
                wish("to you");
        }

        public static void main(String []args) {
                wish();
                wish("dear James Gosling");
        }
}
```

It prints:

```
Happy birthday to you!
Happy birthday dear James Gosling!
```

Here, the method `wish(String name)` is meant for wishing "Happy Birthday" when the name of the person is known. The default method `wish()` is for wishing "Happy Birthday" to anyone. As you can see, you don't have to write `System.out.println` again in the `wish()` method; you can just reuse the `wish(String)` method definition by passing the default value "to you" as argument to `wish()`. Such reuse is effective for large and related method definitions since it saves time writing and testing the same code.

# Constructor Overloading

A default constructor is useful for creating objects with a default initialization value. When you want to initialize the objects with different values in different instantiations, you can pass them as the arguments to constructors. And yes, you can have multiple constructors in a class, which is *constructor overloading*. In a class, the default constructor can initialize the object with default initial values, while another constructor can accept arguments that need to be used for object instantiation.

Here is an example of `Circle` class that has overloaded constructors (see Listing 2-5).

***Listing 2-5.*** Circle.java

```
public class Circle {
        private int xPos;
        private int yPos;
        private int radius;

        // three overloaded constructors for Circle
        public Circle(int x, int y, int r) {
                xPos = x;
                yPos = y;
                radius = r;
        }

        public Circle(int x, int y) {
                xPos = x;
                yPos = y;
                radius = 10; // default radius
        }

        public Circle() {
                xPos = 20; // assume some default values for xPos and yPos
                yPos = 20;
                radius = 10; // default radius
        }

        public String toString() {
                return "center = (" + xPos + "," + yPos + ") and radius = " + radius;
        }

         public static void main(String[]s) {
                System.out.println(new Circle());
                System.out.println(new Circle(50, 100));
                System.out.println(new Circle(25, 50, 5));
         }
}
```

This program prints

```
center = (20,20) and radius = 10
center = (50,100) and radius = 10
center = (25,50) and radius = 5
```

As you can see, the compiler has resolved the constructor calls depending on the number of arguments. The default constructor takes no argument and in this case we have assumed some default values for `xPos`, `yPos`, and `radius` (with values 20, 20, and 10 respectively). The `Circle` constructor with two arguments (`int x` and `int y`) sets the position of `xPos` and `yPos` based on the values of the passed arguments and assumes default value of 10 for the radius member. The `Circle` constructor that takes all the three arguments sets the corresponding fields in the `Circle` class.

Did you notice that you are duplicating the code inside these three constructors? To avoid that code duplication—and reduce your typing effort—you can invoke one constructor from another constructor. Of the three constructors, the constructor taking x-position, y-position, and radius is the most general constructor. The other two constructors can be rewritten in terms of calling the three argument constructors, like so:

```
public Circle(int x, int y, int r) {
        xPos = x;
        yPos = y;
        radius = r;
}

public Circle(int x, int y) {
        this(x, y, 10); // passing default radius 10
}

public Circle() {
        this(20, 20, 10);
        // assume some default values for xPos, yPos and radius
}
```

The output is exactly the same as for the previous program, but this program is shorter. In this case, you used the `this` keyword (which refers to the current object) to call one constructor from another constructor of the same class.

## Overload Resolution

When you define overloaded methods, how does the compiler know which method to call? Can you guess the output of the code in Listing 2-6?

***Listing 2-6.*** Overloaded.java

```
class Overloaded {
        public static void aMethod (int val)    { System.out.println ("int");    }
        public static void aMethod (short val)  { System.out.println ("short");  }
        public static void aMethod (Object val) { System.out.println ("object"); }
        public static void aMethod (String val) { System.out.println ("String"); }

        public static void main(String[] args) {
                byte b = 9;
                aMethod(b);     // first call
                aMethod(9);     // second call
                Integer i = 9;
                aMethod(i);     // third call
                aMethod("9");   // fourth call
        }
}
```

It prints

```
short
int
object
String
```

Here is how the compiler resolved these calls:

1. In the first method call, the statement is `aMethod(b)` where the variable `b` is of type `byte`. There is no `aMethod` definition that takes `byte` as an argument. The closest type (in size) is `short` type and not `int`, so the compiler resolves the call `aMethod(b)` to `aMethod(short val)` definition.

2. In the second method call, the statement is `aMethod(9)`. The constant value 9 is of type `int`. The closest match is `aMethod(int)`, so the compiler resolves the call `aMethod(9)` to `aMethod(int val)` definition.

3. The third method call is `aMethod(i)`, where the variable `i` is of type `Integer`. There is no `aMethod` definition that takes `Integer` as an argument. The closest match is `aMethod(Object val)`, so it is called. Why not `aMethod(int val)`? For finding the closest match, the compiler allows implicit upcasts, not downcasts, so `aMethod(int val)` is not considered.

4. The last method call is `aMethod("9")`. The argument is a `String` type. Since there is an exact match, `aMethod(String val)` is called.

This process of the compiler trying to *resolve* the method call from given overloaded method definitions is called *overload resolution*. For resolving a method call, it first looks for the *exact* match—the method definition with exactly same number of parameters and types of parameters. If it can't find an exact match, it looks for the *closest match* by using upcasts. If the compiler can't find any match, then you'll get a compiler error, as in Listing 2-7.

***Listing 2-7.*** OverloadingError.java

```
class OverloadingError {
        public static void aMethod (byte val )  { System.out.println ("byte");  }
        public static void aMethod (short val ) { System.out.println ("short"); }

        public static void main(String[] args) {
                aMethod(9);
        }
}
```

Here is the compiler error:

```
OverloadingError.java:6: error: no suitable method found for aMethod(int)
                aMethod(9);
                ^
method OverloadingError.aMethod(byte) is not applicable
      (argument mismatch; possible lossy conversion from int to byte)
method OverloadingError.aMethod(short) is not applicable
      (argument mismatch; possible lossy conversion from int to short)
1 error
```

The type of constant 9 is `int`, so there is no matching definition for `aMethod` for the call `aMethod(9)`. As you saw earlier with respect to the overload resolution, the compiler can do upcasts (e.g., `byte` to `int`) for the closest match, but it does not consider downcasts (e.g., `int` to `byte` or `int` to `short`, as in this case). Hence, the compiler does not find any matches and throws you an error.

What if the compiler finds two matches? It will also become an error! Listing 2-8 shows an example.

***Listing 2-8.*** AmbiguousOverload.java

```
class AmbiguousOverload {
        public static void aMethod (long val1, int val2) {
                System.out.println ("long, int");
        }

        public static void aMethod (int val1, long val2) {
                System.out.println ("int, long");
        }

        public static void main(String[] args) {
                aMethod(9, 10);
        }
}
```

Here is the compiler error:

```
AmbiguousOverload.java:11: error: reference to aMethod is ambiguous
                aMethod(9, 10);
                ^
  both method aMethod(long,int) in AmbiguousOverload and method aMethod(int,long) in
AmbiguousOverload match
1 error
```

Why did this call become an "ambiguous" call? The constants 9 and 10 are `int`s. There are two `aMethod` definitions: one is `aMethod(long, int)` and another is `aMethod(int, long)`. So there is no exact match for the call `aMethod(int, int)`. An integer can be implicitly upcasted to both `long` as well as `Integer`. Which one will the compiler choose? Since there are two matches, the compiler complains with an error that the call is ambiguous.

---

 Overload resolution fails (with a compiler error) if there are no matches or ambiguous matches.

---

## Points to Remember

Here are some interesting rules regarding method overloading that will help you in the OCPJP 8 exam:

- Overload resolution takes place entirely at compile time (not at runtime).
- You cannot overload methods with the methods differing in return types alone.
- You cannot overload methods with the methods differing in exception specifications alone.

- For overload resolution to succeed, you need to define methods such that the compiler finds one exact match. If the compiler finds no matches for your call or if the matching is ambiguous, the overload resolution fails and the compiler issues an error.

---

 The *signature* of a method is made up of the method name, number of arguments, and types of arguments. You can overload methods with same name but with different signatures. Since return type and exception specification are not part of the signature, you cannot overload methods based on return type or exception specification alone.

---

# Overriding Methods in Object Class

## Certification Objective

### Override hashCode, equals, and toString methods from Object class

Let us now discuss overriding some of the methods in `Object` class. You can override `clone()`, `equals()`, `hashCode()`, `toString()`, and `finalize()` methods in your classes. Since `getClass()`, `notify()`, `notifyAll()`, and the overloaded versions of `wait()` method are declared `final`, you cannot override these methods.

Why should we override methods in the `Object` class? To answer this question, let's discuss what happens when we don't override the `toString()` method (Listing 2-9).

***Listing 2-9.*** Point.java

```
class Point {
        private int xPos, yPos;

        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }

        public static void main(String []args) {
                // Passing a Point object to println
                // automatically invokes the toString method
                System.out.println(new Point(10, 20));
        }
}
```

It prints

```
Point@19821f (Actual address might differ on your machine, but a similar string will show up)
```

The `toString()` method is defined in the `Object` class, which is inherited by all the classes in Java. Here is the overview of the `toString()` method as defined in the `Object` class:

```
public String toString()
```

The `toString()` method takes no arguments and returns the `String` representation of the object. The default implementation of this method returns `ClassName@hex` version of the object's hashcode. That is why you get this unreadable output. Note that this hexadecimal value will be different for each instance, so if you try this program, you'll get a different hexadecimal value as output. For example, when we ran this program again, we got this output: `Point@affc70`. Hence, we need to override the `toString` method in this `Point` class.

## Overriding toString() Method

When you create new classes, you are expected to override this method to return the desired textual representation of your class. Listing 2-10 shows an improved version of the `Point` class with the overridden version of the `toString()` method.

***Listing 2-10.*** Point.java

```
// improved version of the Point class with overridden toString method
class Point {
        private int xPos, yPos;

        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }

        // this toString method overrides the default toString method implementation
        // provided in the Object base class
        public String toString() {
                return "x = " + xPos + ", y = " + yPos;
        }

        public static void main(String []args) {
                System.out.println(new Point(10, 20));
        }
}
```

This program now prints

```
x = 10, y = 20
```

This is much cleaner, as you would expect. To make it clear, here is a slightly different version of the `main()` method in this `Point` class implementation:

```
public static void main(String []args) {
        Object obj = new Point(10, 20);
        System.out.println(obj);
}
```

It prints

```
x = 10, y = 20
```

Here, the static type of the `obj` variable is `Object` class, and the dynamic type of the object is `Point`. The `println` statement invokes the `toString()` method of the `obj` variable. Here, the method `toString()` of the derived class—the `Point`'s `toString()` method—is invoked due to runtime polymorphism.

## Overriding Issues

While overriding, you need to be careful about the access levels, the name of the method, and its signature. Here is the `toString()` method in the `Point` class just discussed:

```
public String toString() {
        return "x = " + xPos + ", y = " + yPos;
}
```

How about using the `protected` access specifier instead of `public` in this method definition? Will it work?

```
protected String toString() {
        return "x = " + xPos + ", y = " + yPos;
}
```

No, it doesn't. For this change, the compiler complains

```
Point.java:12: error: toString() in Point cannot override toString() in Object
        protected String toString() {
                         ^
attempting to assign weaker access privileges; was public
1 error
```

While overriding, you can provide stronger access privilege, not weaker access; otherwise it will become a compiler error.

Here is another slightly modified version of `toString()` method. Will it work?

```
public Object toString() {
        return "x = " + xPos + ", y = " + yPos;
}
```

You get the following compiler error:

```
Point.java:12: error: toString() in Point cannot override toString() in Object
public Object toString() {
                ^
return type Object is not compatible with String
1 error
```

In this case, you got a compiler error for mismatch because the return type in the overriding method should be exactly the same as the base class method.

Here is another example:

```
public String ToString() {
        return "x = " + xPos + ", y = " + yPos;
}
```

Now the compiler doesn't complain. But this is a new method named `ToString` and it has nothing to do with the `toString` method in `Object`. Hence, this `ToString` method *does not* override the `toString` method.

Keep the following points in mind for correct overriding. The overriding method

- Should have the same argument list types (or compatible types) as the base version.
- Should have the same return type.
  - But from Java 5 onwards, the return type can be a subclass–covariant return types (which you'll learn shortly).
- Should *not* have a more restrictive access modifier than the base version.
  - But it may have a less restrictive access modifier.
- Should *not* throw new or broader checked exceptions.
  - But it may throw fewer or narrower checked exceptions, or any unchecked exception.
- And, oh yes, the names should exactly match!

Remember that you cannot override a method if you do not inherit it. Private methods cannot be overridden because they are not inherited.

---

 The signatures of the base method and overriding method should be compatible for overriding to take place. Incorrect overriding is a common source of bugs in Java programs. In questions related to overriding, look out for mistakes or problems in overriding when answering the questions.

---

## COVARIANT RETURN TYPES

You know that the return types of the methods should exactly match when overriding methods. However, with the covariant return types feature introduced in Java 5, you can provide the derived class of the return type in the overriding method. Well, that's great, but why do you need this feature? Check out these overridden methods with the same return type:

```
abstract class Shape {
        // other methods elided
        public abstract Shape copy();
}

class Circle extends Shape {
        // other methods elided
        public Circle(int x, int y, int radius) { /* initialize fields here */ }
        public Shape copy() { /* return a copy of this object */ }
}
```

```
class Test {
        public static void main(String []args) {
                Circle c1 = new Circle(10, 20, 30);
                Circle c2 = c1.copy();
        }
}
```

This code will give a compiler error of "`incompatible types: Shape cannot be converted to Circle`". This is because of the lack of an explicit downcast from `Shape` to `Circle` in the assignment "`Circle c2 = c1.copy();`".

Since you know clearly that you are going to assign a `Circle` object returned from `Circle`'s copy method, you can give an explicit cast to fix the compiler error:

```
Circle c2 = (Circle) c1.copy();
```

Since it is tedious to provide such downcasts (which are more or less meaningless), Java provides covariant return types where you can give the derived class of the return type in the overriding method. In other words, you can change the definition of `copy` method in `Circle` class as follows:

```
public Circle copy() { /* return a copy of this object */ }
```

Now the assignment in the main method `Circle c2 = c1.copy();` is valid and no explicit downcast is needed (which is good).

# Overriding equals() Method

Let's now override `equals` method in the `Point` class. Before that, here is the signature of the `equals()` method in the `Object` class:

```
public boolean equals(Object obj)
```

The `equals()` method in the `Object` class is an overridable method that takes the `Object` type as an argument. It checks if the contents of the current object and the passed `obj` argument are equal. If so, the `equals()` returns true; otherwise it returns false.

Now, let us enhance to code in Listing 2-10 and override override the `equals()` method in a class named `Point` (see Listing 2-11). Is this a correct implementation?

***Listing 2-11.*** Point.java

```
public class Point {
        private int xPos, yPos;

        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }
```

```
        // override the equals method to perform
        // "deep" comparison of two Point objects
        public boolean equals(Point other){
                if(other == null)
                        return false;
               // two points are equal only if their x and y positions are equal
               if( (xPos == other.xPos) && (yPos == other.yPos) )
                        return true;
                else
                        return false;
          }

        public static void main(String []args) {
                Point p1 = new Point(10, 20);
                Point p2 = new Point(50, 100);
                Point p3 = new Point(10, 20);
                System.out.println("p1 equals p2 is " + p1.equals(p2));
                System.out.println("p1 equals p3 is " + p1.equals(p3));
          }
}
```

This prints

```
p1 equals p2 is false
p1 equals p3 is true
```

The output is as expected, so is this `equals()` implementation correct? No! Let's make the following slight modification in the `main()` method (modifications in this code is highlighted using underline like this):

```
public static void main(String []args) {
        Object p1 = new Point(10, 20);
        Object p2 = new Point(50, 100);
        Object p3 = new Point(10, 20);
        System.out.println("p1 equals p2 is " + p1.equals(p2));
        System.out.println("p1 equals p3 is " + p1.equals(p3));
}
```

Now it prints

```
p1 equals p2 is false
p1 equals p3 is false
```

Why? Both `main()` methods are equivalent. However, this newer `main()` method uses the `Object` type for declaring `p1`, `p2`, and `p3`. The dynamic type of these three variables is `Point`, so it should call the overridden `equals()` method. However, the overriding is wrong: The `equals()` method should have `Object` as the argument instead of the `Point` argument! The current implementation of the `equals()` method in the `Point` class *hides* (*not* overrides) the `equals()` method of the `Object` class. Hence, the `main()` method calls the base version, which is the default implementation of `Point` in `Object` class!


If the name or signature of the base class method and the overriding method don't match, you will cause subtle bugs. So ensure that they are exactly the same.

In order to overcome the subtle problems of overloading, you can use `@Override` annotation, which was introduced in Java 5. This annotation explicitly expresses to the Java compiler the intention of the programmer to use method overriding. In case the compiler is not satisfied with your overridden method, it will issue a complaint, which is a useful alarm for you. Also, the annotation makes the program more understandable, since the `@Override` annotation just before a method definition helps you understand that you are overriding a method.

Here is the code with `@Override` annotation for the `equals` method:

```
@Override
public boolean equals(Point other) {
        if(other == null)
                return false;
        // two points are equal only if their x and y positions are equal
        if((xPos == other.xPos) && (yPos == other.yPos))
                return true;
        else
                return false;
}
```

You'll get a compiler error now for this code:

```
Point.java:11: error: method does not override or implement a method from a supertype
@Override
^
1 error
```

How can you fix it? You need to pass the `Object` type to the argument of the `equals` method. Listing 2-12 shows the program with the fixed `equals` method.

***Listing 2-12.*** Point.java

```
public class Point {
        private int xPos, yPos;

        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }

        // override the equals method to perform "deep" comparison of two Point objects
        @Override
        public boolean equals(Object other) {
                if(other == null)
                        return false;
```

```
                // check if the dynamic type of 'other' is Point
                // if 'other' is of any other type than 'Point', the two objects cannot be
                // equal if 'other' is of type Point (or one of its derived classes), then
                // downcast the object to Point type and then compare members for equality
                if(other instanceof Point) {
                        Point anotherPoint = (Point) other;
                        // two points are equal only if their x and y positions are equal
                        if((xPos == anotherPoint.xPos) && (yPos == anotherPoint.yPos))
                                return true;
                }
                return false;
        }

        public static void main(String []args) {
                Object p1 = new Point(10, 20);
                Object p2 = new Point(50, 100);
                Object p3 = new Point(10, 20);
                System.out.println("p1 equals p2 is " + p1.equals(p2));
                System.out.println("p1 equals p3 is " + p1.equals(p3));
        }
}
```

Now this program prints

```
p1 equals p2 is false
p1 equals p3 is true
```

This is the expected output and with the correct implementation of the `equals` method implementation.

## Invoking Superclass Methods

It is often useful to call the base class method inside the overridden method. To do that, you can use the `super` keyword. In derived class constructors, you can call the base class constructor using the `super` keyword. Such a call should be the *first statement* in a constructor if it is used. You can use the `super` keyword for referring to the base class members also. In those cases, it need not be the first statement in the method body. Let's look at an example.

You implemented a `Point` class that is a 2D-point: it had x and y positions. You can also implement a 3D-point class with x, y, and z positions. For that you do not need to start implementing it from scratch: you can extend the 2D-point and add the z position in the 3D-point class. First, you'll rename the simple implementation of `Point` class to `Point2D`. Then you'll create the `Point3D` class by extending this `Point2D` (see Listings 2-13 and 2-14).

***Listing 2-13.*** Point2D.java

```
class Point2D {
        private int xPos, yPos;
        public Point2D(int x, int y) {
                xPos = x;
                yPos = y;
        }
```

```
        public String toString() {
                return "x = " + xPos + ", y = " + yPos;
        }

        public static void main(String []args) {
                System.out.println(new Point2D(10, 20));
        }
}
```

***Listing 2-14.*** Point3D.java

```
// Here is how we can create Point3D class by extending Point2D class
public class Point3D extends Point2D {
        private int zPos;

         // provide a public constructors that takes three arguments (x, y, and z values)
         public Point3D(int x, int y, int z) {
                // call the superclass constructor with two arguments
                // i.e., call Point2D(int, int) from Point2D(int, int, int) constructor)
                super(x, y); // note that super is the first statement in the method
                        zPos = z;
        }

         // override toString method as well
        public String toString() {
                return super.toString() + ", z = " + zPos;
        }

         // to test if we extended correctly, call the toString method of a Point3D object
        public static void main(String []args) {
                System.out.println(new Point3D(10, 20, 30));
        }
}
```

This program prints

```
x = 10, y = 20, z = 30
```

In the class `Point2D`, the class members `xPos` and `yPos` are private, so you cannot access them directly to initialize them in the `Point3D` constructor. However, you can call the superclass constructor using `super` keyword and pass the arguments. Here, `super(x, y);` calls the base class constructor `Point2D(int, int)`. This call to the superclass constructor should be the first statement; if you call it after `zPos = z;`, you'll get a compiler error:

```
public Point3D(int x, int y, int z) {
        zPos = z;
        super(x, y);
}

Point3D.java:19: call to super must be first statement in constructor
                super(x, y);
```

Similarly, you can invoke the `toString()` method of the base class `Point2D` in the `toString()` implementation of the derived class `Point3D` using the `super` keyword.

## Overriding the hashCode() Method

Overriding the `equals` and `hashCode` methods correctly is important for using with classes such as `HashMap` and `HashSet`, which we will discuss further in Chapter 4. Listing 2-15 is a simple `Circle` class example so you can understand what can go wrong when using collections such as `HashSets`.

***Listing 2-15.*** TestCircle.java

```
// This program shows the importance of overriding equals() and hashCode() methods
import java.util.*;

class Circle {
        private int xPos, yPos, radius;
        public Circle(int x, int y, int r) {
                xPos = x;
                yPos = y;
                radius = r;
        }

        public boolean equals(Object arg) {
                if(arg == null) return false;
                if(this == arg) return true;
                if(arg instanceof Circle) {
                        Circle that = (Circle) arg;
                        if( (this.xPos == that.xPos) && (this.yPos == that.yPos)
                                && (this.radius == that.radius )) {
                                return true;
                        }
                }
                return false;
        }
}

class TestCircle {
        public static void main(String []args) {
                Set<Circle> circleList = new HashSet<Circle>();
                circleList.add(new Circle(10, 20, 5));
                System.out.println(circleList.contains(new Circle(10, 20, 5)));
        }
}
```

It prints `false` (not `true`)! Why? The `Circle` class overrides the `equals()` method, but it doesn't override the `hashCode()` method. When you use objects of `Circle` in standard containers, it becomes a problem. For fast lookup, the containers compare hashcode of the objects. If the `hashCode()` method is not overridden, then—even if an object with same contents is passed—the container will not find that object! So you need to override the `hashCode()` method.


If you're using an object in containers like `HashSet` or `HashMap`, make sure you override the `hashCode()` and `equals()` methods correctly. If you don't, you'll get nasty surprises (bugs) while using these containers!

Okay, how do you override the `hashCode()` method? In the ideal case, the `hashCode()` method should return unique hash codes for different objects.

The `hashCode()` method *should* return the same hash value if the `equals()` method returns true. What if the objects are different (so that the `equals()` method returns false)? It is better (although not required) for the `hashCode()` to return different values if the objects are different. The reason is that it is difficult to write a `hashCode()` method that gives unique value for every different object.


The methods `hashCode()` and `equals()` need to be consistent for a class. For practical purposes, ensure that you follow this one rule: the `hashCode()` method should return the same hash value for two objects if the `equals()` method returns true for them.

When implementing the `hashCode()` method, you can use the values of the instance members of the class to create a hash value. Here is a simple implementation of the `hashCode()` method of the `Circle` class:

```
public int hashCode() {
        // use bit-manipulation operators such as ^ to generate close to unique
        // hash codes here we are using the magic numbers 7, 11 and 53,
        // but you can use any numbers, preferably primes
        return (7 * xPos) ^ (11 * yPos) ^ (53 * yPos);
}
```

Now if you run the `main()` method, it prints "true". In this implementation of the `hashCode()` method, you multiply the values by a prime number as well as bit-wise operation. You can write complex code for `hashCode()` if you want a better hashing function, but this implementation is sufficient for practical purposes.

You can use bitwise operators for `int` values. What about other types, like floating-point values or reference types? To give you an example, here is `hashCode()` implementation of `java.awt.Point2D`, which has floating point values `x` and `y`. The methods `getX()` and `getY()` return the `x` and `y` values respectively:

```
public int hashCode() {
        long bits = java.lang.Double.doubleToLongBits(getX());
        bits ^= java.lang.Double.doubleToLongBits(getY()) * 31;
        return (((int) bits) ^ ((int) (bits >> 32)));
}
```

This method uses the `doubleToLongBits()` method, which takes a `double` value and returns a `long` value. For floating-point values `x` and `y` (returned by the `getX` and `getY` methods), you get `long` values in bits and you use bit-manipulation to get `hashCode()`.

Now, how do you implement the hashCode method if the class has reference type members? For example, consider using an instance of Point class as a member instead of xPos and yPos, which are primitive type fields:

```
class Circle {
        private int radius;
        private Point center;
        // other members elided
}
```

In this case, you can use the hashCode() method of Point to implement Circle's hashCode method:

```
public int hashCode() {
        return center.hashCode() ^ radius;
}
```

# Object Composition

## Certification Objective

### Implement inheritance including visibility modifiers and composition

Individual abstractions offer certain functionalities that need to be combined with other objects to represent a bigger abstraction: a composite object that is made up of other smaller objects. You need to make such composite objects to solve real-life programming problems. In such cases, the composite object shares HAS-A relationships with the containing objects, and the underlying concept is referred to as *object composition*.

By way of analogy, a computer is a composite object containing other objects such as CPU, memory, and a hard disk. In other words, the computer object shares a HAS-A relationship with other objects. Listing 2-16 defines a Circle class that uses a Point object to define Circle's center.

***Listing 2-16.*** Circle.java

```
// Point is an independent class and here we are using it with Circle class
class Point {
        private int xPos;
        private int yPos;
        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }
        public String toString() {
                return "(" + xPos + "," + yPos + ")";
        }
}

// Circle.java
public class Circle {
        private Point center;    // Circle "contains" a Point object
        private int radius;
```

```
        public Circle(int x, int y, int r) {
                center = new Point(x, y);
                radius = r;
        }
        public String toString() {
                return "center = " + center + " and radius = " + radius;
        }

        public static void main(String []s) {
                System.out.println(new Circle(10, 10, 20));
        }
        // other members (constructors, area method, etc) are elided ...
}
```

In this example, `Circle` has a `Point` object. In other words, `Circle` and `Point` share a has-a relationship; in other words, `Circle` is a composite object containing a `Point` object. This is a better solution than having independent integer members `xPos` and `yPos`. Why? You can reuse the functionality provided by the `Point` class. Notice the `toString()` method in the `Circle` class:

```
public String toString() {
        return "center = " + center + " and radius = " + radius;
}
```

Here, the use of the variable `center` expands to `center.toString()` and hence the `toString` method of `Point` can be reused in the `Circle`'s `toString` method.

## Composition vs. Inheritance

You are now equipped with a knowledge of composition as well as inheritance (which we covered earlier in this chapter). In some situations, it's difficult to choose between the two. It's important to remember that nothing is a silver bullet—you cannot solve all problems with one construct. You need to analyze each situation carefully and decide which construct is best suited for it.

A rule of thumb is to use HAS-A and IS-A phrases for composition and inheritance, respectively. For instance,

- A computer HAS-A CPU.
- A circle IS-A shape.
- A circle HAS-A point.
- A laptop IS-A computer.
- A vector IS-A list.

This rule can be useful for identifying wrong relationships. For instance, the relationship of car IS-A tire is completely wrong, which means you cannot have an inheritance relationship between the classes `Car` and `Tire`. However, the car HAS-A tire (meaning car has one or more tires) relationship is correct—you can compose a `Car` object containing `Tire` objects.

In real scenarios, the relationship distinctions can be nontrivial. You learned that you can make a base class and put the common functionality of many classes in it. However, many people ignore a big caution sign suspended over this practice—always check whether the IS-A relationship exists between the derived classes and the base class. If the IS-A relationship does not hold, it's better to use composition instead of inheritance.

For example, take a set of classes DynamicDataSet and SnapShotDataSet that require a common functionality—say, sorting. Now, one could derive these data set classes from a sorting implementation, as given in Listing 2-17.

***Listing 2-17.*** Sorting.java

```
import java.awt.List;

public class Sorting {
        public List sort(List list) {
                // sort implementation
                return list;
        }
}

class DynamicDataSet extends Sorting {
        // DynamicDataSet implementation
}

class SnapshotDataSet extends Sorting {
        // SnapshotDataSet implementation
}
```

Do you think this is a good solution? No, it's not a good solution for the following reasons:

- The rule of thumb does not hold here. DynamicDataSet is not a Sorting type. If you make such mistakes in class design, it can be very costly—and you might not be able to fix them later if a lot of code has accumulated that makes the wrong use of inheritance relationships. For example, Stack extends Vector in the Java library. Yet a stack clearly is not a vector, so it could not only create comprehension problems but also lead to bugs. When you create an object of Stack class provided by the Java library, you can add or delete items from anywhere in the container because the base class is Vector, which allows you to delete from anywhere in the vector.

- What if these two types of data set classes have a genuine base class, DataSet? In that case, either Sorting will be the base class of DataSet or one could put the class Sorting in between DataSet and two types of data sets. Both solutions would be wrong.

- There is another challenging issue: what if one DataSet class wants to use one sorting algorithm (say, MergeSort) and another data set class wants to use a different sorting algorithm (say, QuickSort)? Will you inherit from two classes implementing two different sorting algorithms? First, you cannot directly inherit from multiple classes, since Java does not support multiple class inheritance. Second, even if you were able to somehow inherit from two different sorting classes (MergeSort extends QuickSort, QuickSort extends DataSet), that would be an even worse design.

In this case it is best to use composition—in other words, use a HAS-A relationship instead of an IS-A relationship. The resultant code is given in Listing 2-18.

***Listing 2-18.*** Sorting.java

```
import java.awt.List;

interface Sorting {
        List sort(List list);
}

class MergeSort implements Sorting {
        public List sort(List list) {
                // sort implementation
                return list;
        }
}

class QuickSort implements Sorting {
        public List sort(List list) {
                // sort implementation
                return list;
        }
}

class DynamicDataSet {
        Sorting sorting;
        public DynamicDataSet() {
                sorting = new MergeSort();
        }
        // DynamicDataSet implementation
}

class SnapshotDataSet {
        Sorting sorting;
        public SnapshotDataSet() {
                sorting = new QuickSort();
        }
        // SnapshotDataSet implementation
}
```

---


Use inheritance when a subclass specifies a base class, so that you can exploit dynamic polymorphism. In other cases, use composition to get code that is easy to change and loosely coupled. In summary, **favor composition over inheritance**.

---

# Singleton and Immutable Classes

## Certification Objective

**Create and use singleton classes and immutable classes**

There are many situations where you need to create special kinds of classes. In this section let us discuss two such special kinds of classes: singletons and immutable classes.

## Creating Singleton Class

There are scenarios in which you want to make sure that only one instance is present for a particular class. For example, assume that you defined a class that modifies a registry, or you implemented a class that manages printer spooling, or you implemented a thread-pool manager class. In all these situations, you might want to avoid hard-to-find bugs by instantiating no more than one object of such classes. In these situations, you could create a *singleton* class.

A singleton class ensures that only one instance of that class is created. To ensure point of access, the class controls instantiation of its object. Singleton classes are found in many places in Java Development Kit (JDK), such as `java.lang.Runtime`.

Figure 2-2 shows the class diagram of a singleton class. It comprises a single class, the class that you want to make as a singleton. It has a private constructor and a static method to get the singleton object.


***Figure 2-2.*** *UML class diagram of a singleton class*


The singleton class offers two things: one and only one instance of the class, and a global single point of access to that object.

Assume that you want to implement a class for logging application details for tracing the application execution for debugging. For this objective, you may want to ensure that only one instance of `Logger` class exists in your application, and hence you can make `Logger` class a singleton class (see Listing 2-19).

***Listing 2-19.*** Logger.java

```
// Logger class must be instantiated only once in the application; it is to ensure that the
// whole of the application makes use of that same logger instance
public class Logger {
        // declare the constructor private to prevent clients
        // from instantiating an object of this class directly
        private Logger() {    }

        // by default, this field is initialized to null
        // the static method to be used by clients to get the instance of the Logger class
        private static Logger myInstance;

        public static Logger getInstance() {
                if(myInstance == null) {
                        // this is the first time this method is called,
                        // and that's why myInstance is null
                        myInstance = new Logger();
                }
                // return the same object reference any time and
                // every time getInstance is called
                return myInstance;
        }
        public void log(String s) {
                // a trivial implementation of log where
                // we pass the string to be logged to console
                System.err.println(s);
        }
}
```

Look at the singleton implementation of the `Logger` class. The constructor of the class is declared as private, so you cannot simply create a new instance of the `Logger` class using the `new` operator. The only way to get an instance of this class is to call the static member method of the class via the `getInstance()` method. This method checks whether a `Logger` object already exists or not. If not, it creates a `Logger` instance and assigns it to the static member variable. In this way, whenever you call the `getInstance()` method, it will always return the same object of the `Logger` class.

## Ensuring That Your Singleton Is Indeed a Singleton

It is really important (as well as difficult) to ensure that your singleton implementation allows only instance of the class. For instance, the implementation provided in Listing 2-19 works only if your application is single threaded. In the case of multiple threads, trying to get a singleton object may result in creation of multiple objects, which of course defeats the purpose of implementing a singleton. Listing 2-20 shows a version of the `Logger` class that implements the singleton design pattern in a multi-threaded environment.

***Listing 2-20.*** Logger.java

```
public class Logger {
        private Logger() {
                // private constructor to prevent direct instantiation
        }
```

```
        private static Logger myInstance;
        public static synchronized Logger getInstance() {
                if(myInstance == null)
                        myInstance = new Logger();
                return myInstance;
        }
        public void log(String s){
                // log implementation
                System.err.println(s);
        }
}
```

Note the use of the keyword synchronized in this implementation. This keyword is a Java concurrency mechanism to allow only one thread at a time into the synchronized scope. You will learn more about this keyword in Chapter 11 on concurrency.

So, you made the whole method synchronized in order to make it accessible by only a thread at a time. This makes it a correct solution, but there is a problem: poor performance. You wanted to make this method synchronized only at the first time the method is called, but since you declared the whole method as synchronized, all subsequent calls to this method make it a performance bottleneck.

Listing 2-21 shows another implementation of the Logger class that is based on the "initialization on demand holder" idiom. This idiom uses inner classes and does not use any synchronization construct (we discuss inner classes in Chapter 3). It exploits the fact that inner classes are not loaded until they are referenced.

***Listing 2-21.*** Logger.java

```
public class Logger {
        private Logger() {
                // private constructor
        }
        public static class LoggerHolder {
                public static Logger logger = new Logger();
        }
        public static Logger getInstance() {
                return LoggerHolder.logger;
        }
        public void log(String s) {
                // log implementation
                System.err.println(s);
        }
}
```

This is an efficient working solution for singletons that works well for multi-threaded applications as well. However, before we close this discussion on singletons, two parting words of caution. First, use singletons wherever it is appropriate, but do not overuse it. Second, make sure that your singleton implementation ensures the creation of only one instance even if your code is multi-threaded.

# Immutable Classes

What is an immutable object? Once an object is created and initialized, it cannot be modified. We can call accessor methods (i.e., getter methods), copy the objects, or pass the objects around—but no method should allow modifying the state of the object. Wrapper classes (such as `Integer` and `Float`) and `String` class are well-known examples of classes that are immutable.

Let us now discuss `String` class. `String` is immutable: once you create a `String` object, you cannot modify it. How about methods such as `trim` that removes leading and trailing whitespace characters–do such methods modify the state of the `String` object? No. If there are any leading or trailing whitespace characters, the `trim` method removes them and returns a new `String` object instead of modifying that `String` object.

There are many advantages with creating immutable objects. Let us discuss some of these advantages in the context of `String` class:

- Immutable objects are safer to use than mutable objects. Once you check its value, you can be sure that it remains the same and is not modified behind your back (by some other code). So, it is less error-prone when we use immutable objects. For instance, if you have a reference to a string and found that it has the characters "contents", if you retain that reference and use it later, you can be sure that it still has the characters "contents" in it (because no code can modify it).

- Immutable objects are thread-safe. For instance, a thread can access a `String` object without worrying if any other thread would change it when it is accessing the object–it cannot happen because a `String` object is immutable.

- Immutable objects that have same state can save space by sharing the state internally. For example, when the contents are same, `String` objects share the same contents (known as "string interning"). You can use the `intern()` method to ascertain that:

```
String str1 = new String("contents");
String str2 = new String("contents");
System.out.println("str1 == str2 is " + (str1 == str2));
System.out.println("str1.intern() == str2.intern() is "
    + (str1.intern() == str2.intern()));

// this code prints:
str1 == str2 is false
str1.intern() == str2.intern() is true
```

Because of the benefits of using immutable objects, Joshua Bloch in his book Effective Java strongly encourages the use of immutable classes: *"Classes should be immutable unless there's a very good reason to make them mutable... If a class cannot be made immutable, you should still limit its mutability as much as possible."*

## Defining Immutable Classes

Keep the following aspects in mind for creating your own immutable objects:

- Make the fields final and initialize them in the constructor. For primitive types, the field values are final, there is no possibility of changing the state after it got initialized. For reference types, you cannot change the reference.

- For reference types that are mutable, you need to take of some more aspects to ensure immutability. Why? Even if you make the mutable reference type final it is possible that the members may refer to objects created outside the class or may be referred by others. In this case,
  - Make sure that the methods don't change the contents inside those mutable objects.
  - Don't share the references outside the classes–for example, as a return value from methods in that class. If the references to fields that are mutable are accessible from code outside the class, they can end up modifying the contents of the object.
  - If you must return a reference, return the deep copy of the object (so that the original contents remain intact even if the contents inside the returned object is changed).
- Provide only accessor methods (i.e., getter methods) but don't provide mutator methods (i.e., setter methods)
  - In case changes must be made to the contents of the object, create a new immutable object with the necessary changes and return that reference.
- Declare the class final. Why? If the class is inheritable, methods in its derived class can override them and modify the fields.

Because the `final` keyword is mentioned as an exam topic under the title "Advanced Class Design", we cover it in the next chapter (Chapter 3); please review that section if you are not familiar with using `final` keyword.

Let us now review the `String` class to understand how these aspects of taken care in its implementation:

- All its fields are made private. The `String` constructors initialize the fields.
- There are methods such as `trim`, `concat`, and `substring` that need to change the contents of the `String` object. To ensure immutability, such methods return new `String` objects with modified contents.
- The `String` class is final, so you cannot extend it and override its methods.

Here is a circle class that is immutable. For brevity, this example shows only the relevant methods for illustrating how to define an immutable class (Listing 2-22).

***Listing 2-22.*** ImmutableCircle.java

```
// Point is a mutable class
class Point {
        private int xPos, yPos;

        public Point(int x, int y) {
                xPos = x;
                yPos = y;
        }

        public String toString() {
                return "x = " + xPos + ", y = " + yPos;
        }
```

```
        int getX() { return xPos; }
        int getY() { return yPos; }
}

// ImmutableCircle is an immutable class - the state of its objects
// cannot be modified once the object is created

public final class ImmutableCircle {
        private final Point center;
        private final int radius;
        public ImmutableCircle(int x, int y, int r) {
                center = new Point(x, y);
                radius = r;
        }
        public String toString() {
                return "center: " + center + " and radius = " + radius;
        }
        public int getRadius() {
                return radius;
        }
        public Point getCenter() {
                // return a copy of the object to avoid
                // the value of center changed from code outside the class
                return new Point(center.getX(), center.getY());
        }
        public static void main(String []s) {
                System.out.println(new ImmutableCircle(10, 10, 20));
        }
        // other members are elided ...
}
```

This program prints

```
center: x = 10, y = 10 and radius = 20
```

Note the following aspects in the definition of the `ImmutableCircle` class:

- The class is declared `final` to prevent inheritance and overriding of its methods
- The class has only final data members and they are `private`
- Because `center` is a mutable field, the getter method `getCenter()` returns a copy of the `Point` object

Immutable objects also have certain drawbacks. To ensure immutability, methods in immutable classes may end-up creating numerous copies of the objects. For instance, every time `getCenter()` is called on the `ImmutableCircle` class, this method creates a copy of the `Point` object and returns it. For this reason, we may need to define a mutable version of the class as well, for example, a mutable `Circle` class.

The `String` class is useful in most scenarios, if we call methods such as `trim`, `concat`, or `substring` in a loop, these methods are likely to create numerous (temporary) `String` objects. Fortunately, Java provides `StringBuffer` and `StringBuilder` classes that are not mutable. They provide functionality similar to `String`, but you can mutate the contents within the objects. Hence, depending on the context, we can choose to use `String` class or one of `StringBuffer` or `StringBuilder` classes.

# Using the "static" Keyword

## Certification Objective

**Develop code that uses static keyword on initialize blocks, variables, methods, and classes**

Now let us discuss how you can use `static` keyword in different ways in Java. Suppose you wanted a write a simple class that counts the number of objects of its class type created so far. Will the program in Listing 2-23 work?

***Listing 2-23.*** Counter.java

```
// Counter class should count the number of instances created from that class
public class Counter {
        private int count; // variable to store the number of objects created
        // for every Counter object created, the default constructor will be called;
        // so, update the counter value inside the default constructor
        public Counter() {
                count++;
        }
        public void printCount() { // method to print the counter value so far
                System.out.println("Number of instances created so far is: " + count);
        }
        public static void main(String []args) {
                Counter anInstance = new Counter();
                anInstance.printCount();
                Counter anotherInstance = new Counter();
                anotherInstance.printCount();
        }
}
```

The output of the program is

```
Number of instances created so far is: 1
Number of instances created so far is: 1
```

Oops! From the output, it is clear that the class does not keep track of the number of objects created. What happened?

You've used an *instance variable* `count` to keep track of the number of objects created from that class. Since every instance of the class has the value count, it always prints `1`! What you need is a variable that can be shared across all its instances. This can be achieved by declaring a variable `static`. A static variable is associated with its class rather than its object or instance; hence they are known as *class variables*. A static variable is initialized only once when execution of the program starts. A static variable shares its state with all instances of the class. You access a static variable using its class name (instead of an instance). Listing 2-24 shows the correct implementation of the `Counter` class with both the `count` variable and the `printCount` method declared static.

***Listing 2-24.*** Counter.java

```
// Counter class should count the number of instances created from that class
public class Counter {
        private static int count; // variable to store the number of objects created
        // for every Counter object created, the default constructor will be called;
        // so, update the counter value inside the default constructor
        public Counter() {
                count++;
        }
        public static void printCount() { // method to print the counter value so far
                System.out.println("Number of instances created so far is: " + count);
        }
        public static void main(String []args) {
                Counter anInstance = new Counter();
                // note we call printCount using the class name
                // instead of instance variable name
                Counter.printCount();
                Counter anotherInstance = new Counter();
                Counter.printCount();
        }
}
```

This program prints

```
Number of instances created so far is: 1
Number of instances created so far is: 2
```

Here, the static variable `count` is initialized when the execution started. At the time of first object creation, the count is incremented to one. Similarly, when the second object got created, the value of the `count` became 2. As the output of the program shows, both objects updated the same copy of the `count` variable.

Note how we changed the call to `printCount()` to use class name `Counter`, as in `Counter.printCount()`. The compiler will accept the previous two calls of `anInstance.printCount()` and `anotherInstance.printCount()` as there is no semantic difference between calling a static method using a class name or instance variable name. However, to use instance variables to call static methods is not recommended. It is conventional practice to call instance methods using instance variables and to call static methods using class names.

A static method can only access static variables and can call only static methods. In contrast, an instance method (nonstatic) may call a static method or access a static variable.

## Static Block

Apart from static variables and methods, you can also define a *static block* in your class definition. This static block will be executed by JVM when it loads the class into memory. For instance, in the previous example, you can define a static block to initialize the count variable to default 1 instead of the default value 0, as shown in Listing 2-25.

***Listing 2-25.*** Counter.java

```
public class Counter {
        private static int count;
        static {
                // code in this static block will be executed when
                // the JVM loads the class into memory
                count = 1;
        }
        public Counter() {
                count++;
        }
        public static void printCount() {
                System.out.println("Number of instances created so far is: " + count);
        }
        public static void main(String []args) {
                Counter anInstance = new Counter();
                Counter.printCount();
                Counter anotherInstance = new Counter();
                Counter.printCount();
        }
}
```

This program prints

```
Number of instances created so far is: 2
Number of instances created so far is: 3
```

Do not confuse a static block with a constructor. A constructor will be invoked when an instance of the class is created, while the static block will be invoked when the JVM loads the corresponding class.

## Points to Remember

- The `main()` method, where the main execution of the program starts, is always declared static. Why? If it were an instance method, it would be impossible to invoke it. You'd have to start the program to be able to create an instance and then call the method, right?
- You cannot override a static method provided in a base class. Why? Based on the instance type, the method call is resolved with runtime polymorphism. Since static methods are associated with a class (and not with an instance), you cannot override static methods, and runtime polymorphism is not possible with static methods.
- A static method cannot use the `this` keyword in its body. Why? Remember that static methods are associated with a class and not an instance. Only instance methods have an implicit reference associated with them; hence class methods do not have a `this` reference associated with them.
- A static method cannot use the `super` keyword in its body. Why? You use the `super` keyword for invoking the base class method from the overriding method in the derived class. Since you cannot override static methods, you cannot use the `super` keyword in its body.

- Since static methods cannot access instance variables (nonstatic variables), they are most suited for utility functions. That's why there are many utility methods in Java. For example, all methods in the `java.lang.Math` library are static.
- Calling a static method is considered to be slightly more efficient compared to calling an instance method. This is because the complier need not pass the implicit `this` object reference while calling a static method, unlike an instance method.

# Summary

Let us briefly review the key points from each certification objective in this chapter. Please read it before appearing for the exam.

**Implement encapsulation**

- *Encapsulation*: Combining data and the functions operating on it as a single unit.
- You cannot access the *private* methods of the base class in the derived class.
- You can access the *protected* method either from a class in the same package (just like package private or default) as well as from a derived class.
- You can also access a method with a *default access modifier* if it is in the same package.
- You can access *public* methods of a class from any other class.

**Implement inheritance including visibility modifiers and composition**

- *Inheritance*: Creating hierarchical relationships between related classes. Inheritance is also called an *"IS-A" relationship*.
- You use the `super` keyword to call base class methods.
- Inheritance implies IS-A and composition implies HAS-A relationship.
- Favor composition over inheritance.

**Implement polymorphism**

- *Polymorphism*: Interpreting the same message (i.e., method call) with different meanings depending on the context.
- Resolving a method call based on the dynamic type of the object is referred to as *runtime polymorphism*.
- Overloading is an example of *static polymorphism* (*early binding*) while overriding is an example of *dynamic polymorphism* (*late binding*).
- *Method overloading*: Creating methods with same name but different types and/or numbers of parameters.
- You can have *overloaded constructors*. You can call a constructor of the same class in another constructor using the `this` keyword.

- *Overload resolution* is the process by which the compiler looks to resolve a call when overloaded definitions of a method are available.
- In *overriding*, the name of the method, number of arguments, types of arguments, and return type should match exactly.
- In *covariant return types*, you can provide the derived class of the return type in the overriding method.

**Override hashCode, equals, and toString methods from Object class**

- You can override `clone()`, `equals()`, `hashCode()`, `toString()` and `finalize()` methods in your classes. Since `getClass()`, `notify()`, `notifyAll()`, and the overloaded versions of `wait()` method are declared `final`, you cannot override these methods.
- If you're using an object in containers like `HashSet` or `HashMap`, make sure you override the `hashCode()` and `equals()` methods correctly. For instance, ensure that the `hashCode()` method returns the same hash value for two objects if the `equals()` method returns true for them.

**Create and use singleton classes and immutable classes**

- A singleton ensures that only one object of its class is created.
- Making sure that an intended singleton implementation is indeed singleton is a nontrivial task, especially in a multi-threaded environment.
- Once an immutable object is created and initialized, it cannot be modified.
- Immutable objects are safer to use than mutable objects; further, immutable objects are thread safe; further, immutable objects that have same state can save space by sharing the state internally.
- To define an immutable class, make it final. Make all its fields private and final. Provide only accessor methods (i.e., getter methods) but don't provide mutator methods. For fields that are mutable reference types, or methods that need to mutate the state, create a deep copy of the object if needed.

**Develop code that uses static keyword on initialize blocks, variables, methods, and classes**

- There are two types of member variables: class variables and instance variables. All variables that require an instance (object) of the class to access them are known as *instance variables*. All variables that are shared among all instances and are associated with a class rather than an object are referred to as *class variables* (declared using the `static` keyword).
- All static members do not require an instance to call/access them. You can directly call/access them using the class name.
- A static member can call/access only a static member of the same class.

## QUESTION TIME

1. What will be the output of this program?

```
class Color {
     int red, green, blue;

     void Color() {
             red = 10;
             green = 10;
             blue = 10;
     }

     void printColor() {
           System.out.println("red: " + red + " green: " + green + " blue: " +
           blue);
     }

     public static void main(String [] args) {
             Color color = new Color();
             color.printColor();
     }
}
```

   A. Compiler error: no constructor provided for the class

   B. Compiles fine, and when run, it prints the following: red: 0 green: 0 blue: 0

   C. Compiles fine, and when run, it prints the following: red: 10 green: 10 blue: 10

   D. Compiles fine, and when run, crashes by throwing `NullPointerException`

2. Consider the following program and predict the behavior of this program:

```
class Base {
     public void print() {
             System.out.println("Base:print");
     }
}

abstract class Test extends Base { //#1
     public static void main(String[] args) {
             Base obj = new Base();
             obj.print(); //#2
     }
}
```

   A. Compiler error "an abstract class cannot extend from a concrete class" at statement marked with comment #1

   B. Compiler error "cannot resolve call to print method" at statement marked with comment #2

C. The program prints the following: `Base:print`

D. The program will throw a runtime exception of `AbstractClassInstantiationException`

3. Consider the following program:

```
class Base {}
class DeriOne extends Base {}
class DeriTwo extends Base {}

class ArrayStore {
     public static void main(String []args) {
             Base [] baseArr = new DeriOne[3];
             baseArr[0] = new DeriOne();
             baseArr[2] = new DeriTwo();
             System.out.println(baseArr.length);
     }
}
```

Which one of the following options correctly describes the behavior of this program?

A. This program prints the following: 3

B. This program prints the following: 2

C. This program throws an `ArrayStoreException`

D. This program throws an `ArrayIndexOutOfBoundsException`

4. Determine the output of this program:

```
class Color {
     int red, green, blue;

     Color() {
             Color(10, 10, 10);
     }

     Color(int r, int g, int b) {
             red = r;
             green = g;
             blue = b;
     }

     void printColor() {
             System.out.println("red: " + red + " green: " + green + " blue: " +
             blue);
     }

     public static void main(String [] args) {
             Color color = new Color();
             color.printColor();
     }
}
```

A. Compiler error: cannot find symbol

B. Compiles without errors, and when run, it prints: red: 0 green: 0 blue: 0

C. Compiles without errors, and when run, it prints: red: 10 green: 10 blue: 10

D. Compiles without errors, and when run, crashes by throwing `NullPointerException`

5. Choose the correct option based on this code segment:

```
class Rectangle { }
class ColoredRectangle extends Rectangle { }
class RoundedRectangle extends Rectangle { }
class ColoredRoundedRectangle extends ColoredRectangle, RoundedRectangle { }
```

Choose an appropriate option:

A. Compiler error: '{' expected cannot extend two classes

B. Compiles fine, and when run, crashes with the exception `MultipleClassInheritanceException`

C. Compiler error: class definition cannot be empty

D. Compiles fine, and when run, crashes with the exception `EmptyClassDefinitionError`

6. Consider the following program and determine the output:

```
class Test {
     public void print(Integer i) {
             System.out.println("Integer");
     }
     public void print(int i) {
             System.out.println("int");
     }
     public void print(long i) {
             System.out.println("long");
     }
     public static void main(String args[]) {
             Test test = new Test();
                 test.print(10);
     }
}
```

A. The program results in a compiler error ("ambiguous overload")

B. `long`

C. `Integer`

D. `int`

```
        public static void main(String [] args) {
                System.out.println(new Color());
        }
}
```

A. Compiler error: incompatible types

B. Compiles fine, and when run, it prints the following: The color is: 30

C. Compiles fine, and when run, it prints the following: The color is: 101010

D. Compiles fine, and when run, it prints the following: The color is: red green blue

10. Choose the best option based on the following program:

```
class Color {
     int red, green, blue;

     Color() {
             this(10, 10, 10);
     }

     Color(int r, int g, int b) {
             red = r;
             green = g;
             blue = b;
      }

     String toString() {
          return "The color is: " + " red = " + red + " green = " + green + "
          blue = " + blue;
      }

     public static void main(String [] args) {
          // implicitly invoke toString method
          System.out.println(new Color());
     }
}
```

A. Compiler error: attempting to assign weaker access privileges; `toString` was public in `Object`

B. Compiles fine, and when run, it prints the following: The color is: red = 10 green = 10 blue = 10

C. Compiles fine, and when run, it prints the following: The color is: red = 0 green = 0 blue = 0

D. Compiles fine, and when run, it throws `ClassCastException`

**Answers**:

1. B. Compiles fine, and when run, it prints the following: red: 0 green: 0 blue: 0

   Remember that a constructor does not have a return type; if a return type is provided, it is treated as a method in that class. In this case, since `Color` had void return type, it became a method named `Color()` in the `Color` class, with the default `Color` constructor provided by the compiler. By default, data values are initialized to zero, hence the output.

2. C. The program prints the following: `Base:print`

   It is possible for an abstract class to extend a concrete class, though such inheritance often doesn't make much sense. Also, an abstract class can have static methods. Since you don't need to create an object of a class to invoke a static method in that class, you can invoke the `main()` method defined in an abstract class.

3. C. This program throws an `ArrayStoreException`

   The variable `baseArr` is of type `Base[]`, and it points to an array of type `DeriOne`. However, in the statement `baseArr[2] = new DeriTwo()`, an object of type `DeriTwo` is assigned to the type `DeriOne`, which does not share a parent-child inheritance relationship-they only have a common parent, which is `Base`. Hence, this assignment results in an `ArrayStoreException`.

4. A. Compiler error: cannot find symbol

   The compiler looks for the method `Color()` when it reaches this statement: `Color(10, 10, 10);`. The right way to call another constructor is to use the `this` keyword as follows: `this(10, 10, 10);`.

5. A. Compiler error: '{' expected – cannot extend two classes

   Java does not support multiple class inheritance. Since `ColoredRectangle` and `RoundedRectangle` are classes, it results in a compiler error when `ColoredRoundedRectangle` class attempts to extend these two classes. Note that it is acceptable for a class to be empty.

6. D. int

   If `Integer` and `long` types are specified, a literal will match to `int`. So, the program prints `int`.

7. B. Public and protected both can be used

   You can provide only a less restrictive or same-access modifier when overriding a method.

8. A. `finalize()` method and B. `clone()` method

   The methods `finalize()` and `clone()` can be overridden. The methods `getClass()`, `notify()`, and `wait()` are `final` methods and so cannot be overridden.

9. C. Compiles fine, and when run, it prints the following: The color is: 101010

   The `toString()` implementation has the expression "The color is:" + red + blue + green. Since the first entry is String, the + operation becomes the string concatenation operator with resulting string "The color is: 10". Following that, again there is a concatenation operator + and so on until finally it prints "The color is: 101010".

10. A. Compiler error: attempting to assign weaker access privileges; `toString` was public in `Object`

    No access modifier is specified for the `toString()` method. `Object's toString()` method has a public access modifier; you cannot reduce the visibility of the method. Hence, it will result in a compiler error.

CHAPTER 5

# Lambda Built-in Functional Interfaces

**Certification Objectives**

**Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier**

**Develop code that uses primitive versions of functional interfaces**

**Develop code that uses binary versions of functional interfaces**

**Develop code that uses the UnaryOperator interface**

The `java.util.function` has numerous built-in interfaces. Other packages in the Java library (notably `java.util.stream` package) make use of the interfaces defined in this package. For OCPJP 8 exam, you should be familiar with using key interfaces provided in this package.

As we discussed earlier (in Chapter 3), a functional interface declares a single abstract method (but in addition it can have any number of default or static methods). Functional interfaces are useful for creating lambda expressions. The entire `java.util.function` package consists of functional interfaces.

Before defining your own functional interfaces, consider using readily available functional interfaces defined in the `java.util.function` package based on your need. If the signature of the lambda function you are looking for is not available in any of the functional interfaces provided in this library, you can define your own functional interfaces.

## Using Built-in Functional Interfaces

**Certification Objective**

**Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier**

In this section, let us discuss four important built-in interfaces included in the `java.util.function` package: `Predicate`, `Consumer`, `Function`, and `Supplier`. See Table 5-1 and Figure 5-1 to get an overview of these functional interfaces.

***Table 5-1.*** *Key Functional Interfaces in java.util.function Package*

| Functional Interface | Brief Description | Common Use |
|---|---|---|
| `Predicate<T>` | Checks a condition and returns a boolean value as result | In `filter()` method in `java.util.stream.Stream` which is used to remove elements in the stream that don't match the given condition (i.e., predicate) as argument. |
| `Consumer<T>` | Operation that takes an argument but returns nothing | In `forEach()` method in collections and in `java.util.stream.Stream`; this method is used for traversing all the elements in the collection or stream. |
| `Function<T, R>` | Functions that take an argument and return a result | In `map()` method in `java.util.stream.Stream` to transform or operate on the passed value and return a result. |
| `Supplier<T>` | Operation that returns a value to the caller (the returned value could be same or different values) | In `generate()` method in `java.util.stream.Stream` to create an infinite stream of elements. |

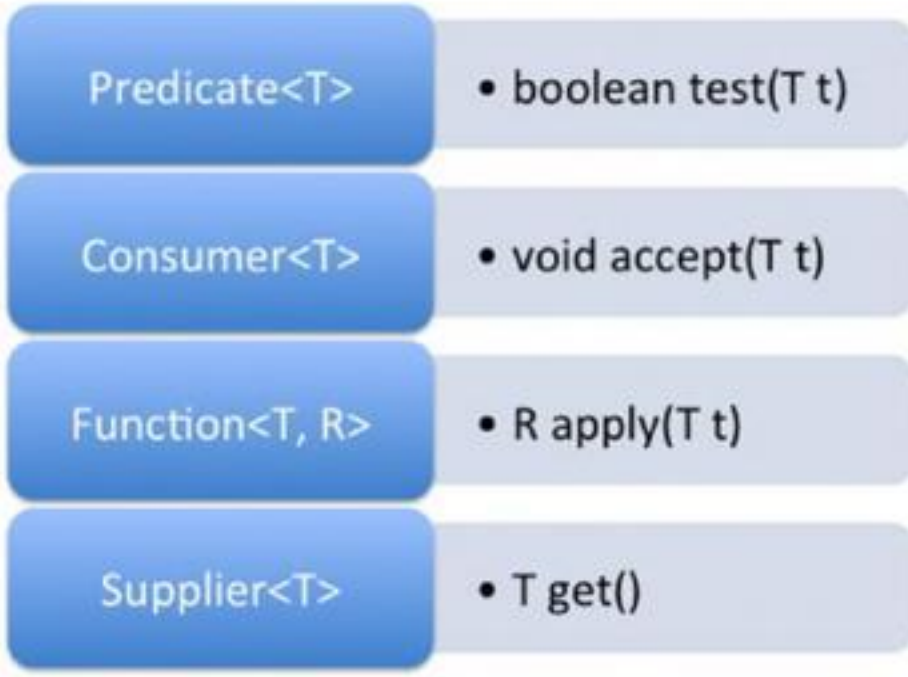


***Figure 5-1.*** *Abstract method declarations in key functional interfaces in java.util.function package*

## The Predicate Interface

In code, we often need to use functions that check a condition and return a boolean value. Consider the following code segment:

```
Stream.of("hello", "world")
    .filter(str -> str.startsWith("h"))
    .forEach(System.out::println);
```

This code segment just prints "hello" on the console. The `filter()` method returns true only if the passed string starts with "h", and hence it "filters out" the string "world" from the stream because the string does not start with "h". In this code, the `filter()` method takes a `Predicate` as an argument. Here is the `Predicate` functional interface:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // other methods elided
}
```

The abstract method named `test()` that takes an argument and returns `true` or `false` (Figure 5-2).

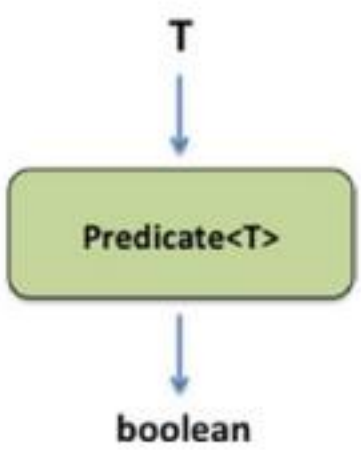


***Figure 5-2.*** *A Predicate<T> takes an argument of type T and returns a boolean value as the result*

---

A `Predicate<T>` "affirms" something as `true` or `false`: it takes an argument of type `T`, and returns a `boolean` value. You can call `test()` method on a `Predicate` object.

---

This functional interface also defines default methods named `and()` and `or()` that take a `Predicate` and return a `Predicate`. These methods have behavior similar to && and || operators. The method `negate()` returns a `Predicate`, and its behavior is similar to the ! operator. How are they useful? Here is a program that illustrates the use of `and()` method in `Predicate` interface (Listing 5-1).

***Listing 5-1.*** PredicateTest.java

```
import java.util.function.Predicate;

public class PredicateTest {
    public static void main(String []args) {
        Predicate<String> nullCheck = arg -> arg != null;
        Predicate<String> emptyCheck = arg -> arg.length() > 0;
        Predicate<String> nullAndEmptyCheck = nullCheck.and(emptyCheck);
        String helloStr = "hello";
        System.out.println(nullAndEmptyCheck.test(helloStr));

        String nullStr = null;
        System.out.println(nullAndEmptyCheck.test(nullStr));
    }
}
```

This program prints:

```
true
false
```

In this program, the object `nullCheck` is a `Predicate` that returns `true` if the given `String` argument is not `null`. The `emptyCheck` predicate returns true if the given string is not empty. The `nullAndEmptyCheck` predicate combines `nullCheck` and `emptyCheck` predicates by making use of the default method named `and()` provided in `Predicate`. Since `helloStr` points to the string "hello" in the first call `nullAndEmptyCheck.test(helloStr)`, and the string is not empty, it returns `true`. However, in the next call, `nullStr` is null, and hence the call `nullAndEmptyCheck.test(nullStr)` returns `false`.

To give another example for using `Predicates`, here is a code segment that makes use of the `removeIf()` method added in the `Collection` interface in Java 8 (Listing 5-2).

***Listing 5-2.*** RemoveIfMethod.java

```
import java.util.List;
import java.util.ArrayList;

public class RemoveIfMethod {
    public static void main(String []args) {
        List<String> greeting = new ArrayList<>();
        greeting.add("hello");
        greeting.add("world");

        greeting.removeIf(str -> !str.startsWith("h"));
        greeting.forEach(System.out::println);
    }
}
```

It prints "hello" in the console. The default method `removeIf()` defined in the `Collection` interface (a super interface of `ArrayList`) takes a `Predicate` as an argument:

```
default boolean removeIf(Predicate<? super E> filter)
```

In the call to `removeIf()` method, we are passing a lambda expression that matches the abstract method `boolean test(T t)` declared in the `Predicate` interface:

```
greeting.removeIf(str -> !str.startsWith("h"));
```

As a result, the string "world" from the `ArrayList` object greeting is removed and hence only "hello" is printed in the console. In this code we have used the ! operator. Instead of that, how about using the equivalent `negate()` method defined in `Predicate`? Yes, it is possible, and here is the changed code:

```
greeting.removeIf(((Predicate<String>) str -> str.startsWith("h")).negate());
```

When you execute the program in Listing 5.2 with this change, the program prints "hello". Note how we have performed explicit typecast (to `Predicate<String>`) in this expression. Without this explicit type cast–as in `((str -> str.startsWith("h")).negate())`–the compiler cannot perform type inference to determine the matching functional interface and hence will report an error.

This program prints:

```
4
9
16
```

This program creates a stream of `Strings` by splitting the string "4, -9, 16". The method reference `Integer::parseInt` is passed to `map()` method–this call returns an `Integer` object for each element in the stream. In the second call to `map()` method in the stream, we have used the lambda function (`i -> (i < 0) ? -i : i`) to produce a list of non-negative integers (alternatively, we could have used `Math::abs` method). The `map()` method we have used here takes a `Function` as an argument (this example is to illustrate where a `Function` interface is useful). Finally, the resulting integers are printed using the `forEach()` method.

The `Function` interface defines a single abstract method named `apply()` that takes an argument of generic type `T` and returns an object of generic type `R` (Figure 5-4):

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    // other methods elided
}
```

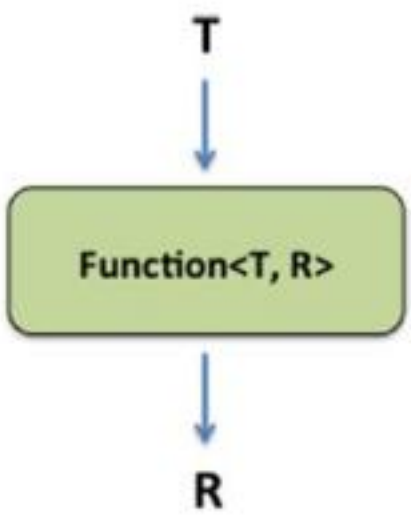


***Figure 5-4.*** *A Function<T, R> takes an argument of type T and returns a value of type R*

The `Function` interface also has default methods such as `compose()`, `andThen()`, and `identity()`.

---

A `Function<T, R>` "operates" on something and returns something: it takes one argument (of generic type `T`) and returns an object (of generic type `R`). You can call `apply()` method on a `Function` object.

---

Here is a simple example that uses a `Function`:

```
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("supercalifragilisticexpialidocious"));
// prints: 34
```

When executed, this program prints which one of the following?

A. java.io.IOException

B. java.io.FileNotFoundException

C. java.lang.Exception

D. java.lang.Throwable

**Answers**:

1. Options B and C

   In option A and D, the throws clause declares to throw exceptions `IOException` and `Exception` respectively, which are more general than the `FileSystemException`, so they are not compatible with the base method definition. In option B, the `foo()` method declares to throw `AccessDeniedException`, which is more specific than `FileSystemException`, so it is compatible with the base definition of the `foo()` method. In option C, the throws clause declares to throw `FileSystemException`, which is the same as in the base definition of the `foo()` method. Additionally it declares to throw `RuntimeException`, which is not a checked exception, so the definition of the `foo()` method is compatible with the base definition of the `foo()` method.

2. B. class `java.lang.IllegalStateException`

   In the expression `new RuntimeException(oob);`, the exception object `oob` is already chained to the `RuntimeException` object. The method `initCause()` cannot be called on an exception object that already has an exception object chained during the constructor call. Hence, the call `re.initCause(oob);` results `in initCause()` throwing an `IllegalStateException`.

3. D. This program fails with compiler error(s)

   The `foo()` method catches `ArrayIndexOutOfBoundsException` and chains it to an `Exception` object. However, since `Exception` is a checked exception, it must be declared in the throws clause of `foo()`. Hence this program results in this compiler error:

```
ExceptionTest.java:6: error: unreported exception Exception; must be caught or
declared to be thrown
        throw new Exception(oob);
    ^
1 error
```

4. D. This program fails with compiler error(s)

   For multi-catch blocks, the single pipe (|) symbol needs to be used and not double pipe (||), as provided in this program. Hence this program will fail with compiler error(s).

5.1. C. catch (`AccountExpiredException` | `AccountNotFoundException` exception)

   For A and B, the base type handler is provided with the derived type handler, hence the multi-catch is incorrect. For D, the exception name exception1 is redundant and will result in a syntax error. C is the correct option and this will compile fine without errors.

CHAPTER 8

# Using the Java SE 8 Date/Time API

## Certification Objectives

**Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration**

**Work with dates and times across timezones and manage changes resulting from daylight savings including format date and times values**

**Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit**

The new Java date and time API is provided in the `java.time` package. This new API in Java 8 replaces the older classes supporting date- and time-related functionality such as the `Date`, `Calendar`, and `TimeZone` classes provided as part of the `java.util` package.

Why did Java 8 introduce a new date and time API when it already had classes such as `Date` and `Calendar` from the early days of Java? The main reason was *inconvenient API design*. For example, the `Date` class has both date and time components; if you only want time information and not date-related information, you have to set the date-related values to zero. Some aspects of the classes are unintuitive as well. For example, in the `Date` constructor, the range of date values is 1 to 31 but the range of month values is 0 to 11 (not 1 to 12)! Further, there are many concurrency-related issues with `java.util.Date` and `SimpleDateFormatter` because they are not thread-safe.

Java 8 provides very good support for date- and time-related functionality in the newly introduced `java.time` package. Most of the classes in this package are immutable and thread-safe. This chapter explains how to use important classes and interfaces in this package, including `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration`, and `TemporalUnit`. You also learn how to work with time zones and daylight savings and how to format date and times values.

The `java.time` API incorporates the concept of *fluent interfaces*: it is designed in such a way that the code is more readable and easier to use. For this reason, classes in this package have numerous static methods (many of them factory methods). In addition, the methods in the classes follow a common naming convention (for example, they use the prefixes `plus` and `minus` to add or subtract date or time values).

# Understanding Important Classes in java.time

## Certification Objectives

**Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration**

**Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit**

The `java.time` package consists of four subpackages:

- `java.time.temporal`—Accesses date/time fields and units
- `java.time.format`—Formats the input and output of date/time objects
- `java.time.zone`—Handles time zones
- `java.time.chrono`—Supports calendar systems such as Japanese and Thai calendars

This chapter focuses only on date/time topics covered by the exam objectives. Let's get started by learning to use the `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration` classes.

## Using the LocalDate class

`java.time.LocalDate` represents a date without time or time zone. `LocalDate` is represented in the ISO-8601 calendar system in a year-month-day format (YYYY-MM-DD): for example, 2015-10-26.


The Java 8 date and time API uses ISO 8601 as the default calendar format. In this internationally accepted format, the date and time values are sorted from the largest to the smallest unit of time: year, month/week, day, hour, minute, second, and millisecond/nanosecond.

Here's an example that uses `LocalDate`:

```
LocalDate today = LocalDate.now();
System.out.println("Today's date is: " + today);
```

This code printed the following when we ran it:

```
Today's date is: 2015-10-26
```

The `LocalDate.now()` method gets the current date using the system clock, based on the default time zone. You can get a `LocalDate` object by explicitly specifying the day, month, and year components:

```
LocalDate newYear2016 = LocalDate.of(2016, 1, 1);
System.out.println("New year 2016: " + newYear2016);
```

This code prints the following:

```
New year 2016: 2016-01-01
```

How about this code?

```
LocalDate valentinesDay = LocalDate.of(2016, 14, 2);
System.out.println("Valentine's day is on: " + valentinesDay);
```

It throws an exception:

```
Exception in thread "main" java.time.DateTimeException: Invalid value for MonthOfYear
(valid values 1 - 12): 14
```

In this case, the `month` and `dayOfMonth` argument values are interchanged. The `of()` method of `LocalDate` is declared as follows:

```
LocalDate of(int year, int month, int dayOfMonth)
```

To avoid making this mistake, you can use the overloaded version `LocalDate.of(int year, Month month, int day)`. The second argument, `java.time.Month`, is an enumeration that represents the 12 months of the year. If you interchange the day and month arguments, you get a compiler error. Here is the improved version that uses this enumeration:

```
LocalDate valentinesDay = LocalDate.of(2016, Month.FEBRUARY, 14);
System.out.println("Valentine's day is on: " + valentinesDay);
```

This code prints

```
Valentine's day is on: 2016-02-14
```

The `LocalDate` class has methods with which you can add or subtract days, weeks, months, or years to or from the current `LocalDate` object. For example, suppose your visa expires 180 days from now. Here is a code segment that shows the expiry date (assuming today's date is 2015-10-26):

```
long visaValidityDays = 180L;
LocalDate currDate = LocalDate.now();
System.out.println("My Visa expires on: " + currDate.plusDays(visaValidityDays));
```

This code segment prints the following:

```
My Visa expires on: 2016-04-23
```

In addition to the `plusDays()` method, `LocalDate` also provides `plusWeeks()`, `plusMonths()`, and `plusYears()` methods, as well as methods for subtracting: `minusDays()`, `minusWeeks()`, `minusMonths()`, and `minusYears()`. Table 8-1 lists a few more methods in the `LocalDate` class that you need to know (this table mentions classes such as `ZoneId`—they're discussed later in this chapter).

***Table 8-1.*** *Important Methods in the* `LocalDate` *Class*

| Method | Short Description | Example Code |
|---|---|---|
| `LocalDate now(Clock clock)`<br>`LocalDate now(ZoneId zone)` | Returns a `LocalDate` object with the current date using the passed `clock` or `zone` argument | `// assume today's date is 26 Oct 2015`<br><br>`LocalDate.now(Clock.systemDefaultZone());`<br>`// returns current date as 2015-10-26`<br><br>`LocalDate.now(ZoneId.of("Asia/Kolkata"));`<br>`// returns current date as 2015-10-26`<br><br>`LocalDate.now(ZoneId.of("Asia/Tokyo"));`<br>`// returns current date as 2015-10-27` |
| `LocalDate ofYearDay(int year, int dayOfYear)` | Returns the `LocalDate` from the year and `dayOfYear` passed as arguments | `LocalDate.ofYearDay(2016,100);`<br>`// returns date as 2016-04-09` |
| `LocalDate parse(CharSequence dateString)` | Returns the `LocalDate` from the `dateString` passed as the argument | `LocalDate.parse("2015-10-26");`<br>`// returns a LocalDate corresponding`<br>`// to the passed string argument;`<br>`hence it`<br>`// returns date as 2015-10-26` |
| `LocalDate ofEpochDay(Long epochDay)` | Returns the `LocalDate` by adding the number of days to the epoch starting day (the epoch starts in 1970) | `LocalDate.ofEpochDay(10);`<br>`// returns 1970-01-11;` |

## Using the LocalTime Class

The `java.time.LocalTime` class is similar to `LocalDate` except that `LocalTime` represents time without dates or time zones. The time is in the ISO-8601 calendar system format: `HH:MM:SS.nanosecond`. Both `LocalTime` and `LocalDate` use the system clock and the default time zone.

Here is an example that uses `LocalTime`:

```
LocalTime currTime = LocalTime.now();
System.out.println("Current time is: " + currTime);
```

When we executed it, it printed the following:

```
Current time is: 12:23:05.072
```

As mentioned, `LocalTime` uses the system clock and its default time zone. To create different time objects based on specific time values, you can use the overloaded `of()` method of the `LocalTime` class:

```
System.out.println(LocalTime.of(18,30));
// prints: 18:30
```

When we ran this code, it printed the following:

```
Today's date and current time is: 2015-10-29T21:04:36.376
```

In this output, note that the character *T* stands for *time*, and it separates the date and time components. Using `LocalDateTime.now()` gets the current date and time using the system clock and its default time zone.

Many classes in the `java.time` package, including `LocalDate`, `LocalTime`, and `LocalDateTime`, support `isAfter()` and `isBefore()` methods for comparison:

```
LocalDateTime christmas = LocalDateTime.of(2015, 12, 25, 0, 0);
LocalDateTime newYear = LocalDateTime.of(2016, 1, 1, 0, 0);
System.out.println("New Year 2016 comes after Christmas 2015? "+newYear.isAfter(christmas));
```

This code prints the following:

```
New Year 2016 comes after Christmas 2015? true
```

You can use the `toLocalDate()` and `toLocalTime()` methods, respectively, to get `LocalDate` and `LocalTime` objects from a given `LocalDateTime` object:

```
LocalDateTime dateTime = LocalDateTime.now();
System.out.println("Today's date and current time: " + dateTime);
System.out.println("The date component is:  " + dateTime.toLocalDate());
System.out.println("The time component is: " + dateTime.toLocalTime());
```

When we executed this code, it printed

```
Today's date and current time: 2015-11-04T13:19:10.497
The date component is:  2015-11-04
The time component is: 13:19:10.497
```

Similar to the methods listed in Tables 8-1 and 8-2, `LocalDateTime` has methods such as `now()`, `of()`, and `parse()`. Again, similar to `LocalDate` and `LocalTime`, this class also provides methods to add or subtract years, months, days, hours, minutes, seconds, and nanoseconds. To avoid repetition, these methods are not listed again here.

## Using the Instant Class

Suppose you want to trace the execution of a Java application or store the application events in a file. For these purposes, you need to get timestamp values, and you can do so using the `java.time.Instant` class. The instant values began on January 1, 1970, at 00:00:00 hours (known as the *Unix epoch*).

The `Instant` class internally uses a `long` variable that holds the number of seconds since the start of the Unix epoch: `1970-01-01T00:00:00Z` (values that occur before this epoch are treated as negative values). In addition, `Instant` uses an integer variable to store the number of nanoseconds elapsed for each second. The program in Listing 8-1 uses the `Instant` class.

## Using the Duration Class

We discussed the `Period` class earlier—it represents time in terms of years, months, and days. `Duration` is the time equivalent of `Period`. The `Duration` class represents time in terms of hours, minutes, seconds, and so on. It is suitable for measuring machine time or when working with `Instance` objects. Similar to the `Instance` class, the `Duration` class stores the seconds component as a `long` value and nanoseconds using an `int` value.

Say you want to wish your best friend Becky a happy birthday at midnight tonight. Here is how you can find out how many hours to go:

```
LocalDateTime comingMidnight =
        LocalDateTime.of(LocalDate.now().plusDays(1), LocalTime.MIDNIGHT);
LocalDateTime now = LocalDateTime.now();

Duration between = Duration.between(now, comingMidnight);
System.out.println(between);
```

This code prints the following:

```
PT7H13M42.003S
```

This example uses the overloaded version of the `of()` method in the `LocalDateTime` class: `LocalDateTime of(LocalDate, LocalTime)`. The `LocalDate.now()` call returns the current date, but you need to add a day to this value so that you can use `LocalTime.MIDNIGHT` to refer to the upcoming midnight. The `between()` method in `Duration` takes two time values—in this case, two `LocalDateTime` objects. When we executed this program, the time was 16:46:17; from then to midnight was 7 hours, 13 minutes, and 42 seconds. That is indicated by the `toString()` output, `Period`: `PT7H13M42.003S`. The prefix `PT` indicates eriod time, `H` indicates hours, `M` indicates minutes, and `S` indicates seconds.

Table 8-4 lists some of the important methods of the `Duration` class. `TemporalUnit` and `ChronoUnit` are discussed later in this chapter.

***Table 8-4.*** *Important Methods in the `Duration` Class*

| Method | Short Description | Example Code |
|---|---|---|
| `Duration of(long number, TemporalUnit unit)` | Returns a `Duration` object for the given number in the specified format | `Duration.of(3600, ChronoUnit.MINUTES) // returns "PT60H"` |
| `Duration ofDays(long unit)`<br>`Duration ofHours(long unit)`<br>`Duration ofMinutes(long unit)`<br>`Duration ofSeconds(long unit)`<br>`Duration ofMillis(long unit)`<br>`Duration ofNanos(long unit)` | Returns `Duration` based on the unit given in the argument | `Duration.ofDays(4)`<br>`// returns "PT96H"`<br>`Duration.ofHours(2)`<br>`// returns "PT2H"`<br>`Duration.ofMinutes(15)`<br>`// returns "PT15M"`<br>`Duration.ofSeconds(30)`<br>`//returns "PT30S"`<br>`Duration.ofMillis(120)`<br>`// returns "PT0.12S"`<br>`Duration.ofNanos(120)`<br>`// returns "PT0.00000012S"` |
| `Duration parse(CharSequence string)` | Returns a `Period` from the `string` passed as the argument | `Duration.parse("P2DT10H30M")`<br>`// returns a Duration object`<br>`// with value PT58H30M` |

# Using Time Zone–Related Classes

There are three important classes related to time zones that you need to know in order to work with dates and times across time zones: `ZoneId`, `ZoneOffset`, and `ZonedDateTime`. Let's discuss them now.

## Using the ZoneId Class

In the `java.time` package, the `java.time.ZoneId` class represents time zones. Time zones are typically identified using an offset from Greenwich Mean Time (GMT, also known as UTC/Greenwich).

For instance, we live in India, and the only time zone in India is Asia/Kolkata (zones are given using this region/city format). This code prints the time zone:

```
System.out.println("My zone id is: " + ZoneId.systemDefault());
```

For our time zone, it printed this:

```
My zone id is: Asia/Kolkata
```

You can get the list of time zones by calling the static method `getAvailableZoneIds()` in `ZoneId`, which returns a `Set<String>`:

```
Set<String> zones = ZoneId.getAvailableZoneIds();
System.out.println("Number of available time zones is: " + zones.size());
zones.forEach(System.out::println);
```

Here is the result:

```
Number of available time zones is: 589
Asia/Aden
America/Cuiaba
// rest of the output elided...
```

You can pass any of these time-zone identifiers to the `of()` method to create the corresponding `ZoneId` object, as in

```
ZoneId AsiaKolkataZoneId = ZoneId.of("Asia/Kolkata");
```

## Using the ZoneOffset Class

`ZoneId` identifies a time zone, such as `Asia/Kolkata`. Another class, `ZoneOffset`, represents the time-zone offset from UTC/Greenwich. For example, zone ID "Asia/Kolkata" has a zone offset of +05:30 (plus 5 hours and 30 minutes) from UTC/Greenwich. The `ZoneOffset` class extends the `ZoneId` class. We discuss an example that uses `ZoneOffset` in the next section.

## Using the ZonedDateTime Class

In Java 8, if you want to deal only with the date, time, or time zone, you can use `LocalDate`, `LocalTime`, or `ZoneId`, respectively. What if you want all three—date, time, and time zone—together? For that, you can use the `ZonedDateTime` class:

```
LocalDate currentDate = LocalDate.now();
LocalTime currentTime = LocalTime.now();
ZoneId myZone = ZoneId.systemDefault();
ZonedDateTime zonedDateTime = ZonedDateTime.of(currentDate, currentTime, myZone);
System.out.println(zonedDateTime);
```

Here is the result:

```
2015-11-05T11:38:40.647+05:30[Asia/Kolkata]
```

This code segment uses the overloaded static method `ZonedDateTime of(LocalDate, LocalTime, ZoneID)`. Given a `LocalDateTime`, you can use a `ZoneId` to get a `ZonedDateTime` object:

```
LocalDateTime dateTime = LocalDateTime.now();
ZoneId myZone = ZoneId.systemDefault();
ZonedDateTime zonedDateTime = dateTime.atZone(myZone);
```

To illustrate the conversion between these different time zone–related classes, here is a code segment that creates a `ZoneId` object, adds that zone information to a `LocalDateTime` object to get a `ZonedDateTime` object, and finally gets the zone offset from the `ZonedDateTime`:

```
ZoneId myZone = ZoneId.of("Asia/Kolkata");
LocalDateTime dateTime = LocalDateTime.now();
ZonedDateTime zonedDateTime = dateTime.atZone(myZone);
ZoneOffset zoneOffset = zonedDateTime.getOffset();
System.out.println(zoneOffset);
```

It prints the following:

```
+05:30
```

Assume that you are in Singapore, the date is January 1, 2016, and the time is 6:00 a.m. Before talking to your friend who lives in Auckland (New Zealand), you want to find out the time difference between Singapore and Auckland. Listing 8-3 shows a program that uses the `ZoneId`, `ZonedDateTime`, and `Duration` classes, to illustrate how to use these classes together.

***Listing 8-3.*** `TimeDifference.java`

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.Duration;
```

The call `zoneId.getRules().getDaylightSavings(Instant.now());` returns a `Duration` object based on whether DST is in effect at that time. If `Duration.isZero()` is false, DST is in effect in that zone; otherwise, it is not. In this example, the Kolkata time zone does not have DST in effect, but the Auckland time zone has +1 hour of DST.

# Formatting Dates and Times

When programming with dates and times, you often have to print them in different formats. Also, you may have to read date/time information given in different formats. To read or print date and time values in various formats, you can use the `DateTimeFormatter` class in the `java.time.format` package.

The `DateTimeFormatter` class provides many predefined constants for formatting date and time values. Here is a list of a few such predefined formatters (with sample output values):

- `ISO_DATE` (2015-11-05)
- `ISO_TIME` (11:25:47.624)
- `RFC_1123_DATE_TIME` (Thu, 5 Nov 2015 11:27:22 +0530)
- `ISO_ZONED_DATE_TIME` (2015-11-05T11:30:33.49+05:30[Asia/Kolkata])

Here is a simple example that uses the predefined `ISO_TIME` of type `DateTimeFormatter`:

```
LocalTime wakeupTime = LocalTime.of(6, 0, 0);
System.out.println("Wake up time: " + DateTimeFormatter.ISO_TIME.format(wakeupTime));
```

This printed the following:

```
Wake up time: 06:00:00
```

What if you want to use a custom format instead of any of the predefined formats? To do so, you can use the `ofPattern()` method in the `DateTimeFormatter` class:

```
DateTimeFormatter customFormat = DateTimeFormatter.ofPattern("dd MMM yyyy");
System.out.println(customFormat.format(LocalDate.of(2016, Month.JANUARY, 01)));
```

Here is the result:

```
01 Jan 2016
```

You encode the format of the date or time using letters to form a date or time pattern string. Usually these letters are repeated in the pattern.

---

 Uppercase and lowercase letters can have similar or different meanings when used in format strings for dates and times. Read the Javadoc for these patterns carefully before trying to use these letters. For example, in dd-MM-yy, *MM* refers to *month*; however, in dd-mm-yy, *mm* refers to *minutes*!

---

The previous code segment gave a simple example of creating a custom date format. Similar letters are available for creating custom date and time pattern strings. Here is the list of important letters and their meanings for creating patterns for dates (with examples):

- `G` (era: BC, AD)
- `y` (year of era: 2015, 15)
- `Y` (week-based year: 2015, 15)
- `M` (month: 11, Nov, November)
- `w` (week in year: 13)
- `W` (week in month: 2)
- `E` (day name in week: Sun, Sunday)
- `D` (day of year: 256)
- `d` (day of month: 13)

The program in Listing 8-4 uses simple and complex pattern strings to create custom date formats.

***Listing 8-4.*** `CustomDatePatterns.java`

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class CustomDatePatterns {
    public static void main(String []args) {
        // patterns from simple to complex ones
        String [] dateTimeFormats = {
                "dd-MM-yyyy", /* d is day (in month), M is month, y is year */
                "d '('E')' MMM, YYYY", /*E is name of the day (in week), Y is year*/
                "w'th week of' YYYY", /* w is the week of the year */
                "EEEE, dd'th' MMMM, YYYY" /*E is day name in the week */
        };
            LocalDateTime now = LocalDateTime.now();
            for(String dateTimeFormat : dateTimeFormats) {
                System.out.printf("Pattern \"%s\" is %s %n", dateTimeFormat,
                        DateTimeFormatter.ofPattern(dateTimeFormat).format(now));
            }
        }
    }
```

Here is the result:

```
Pattern "dd-MM-yyyy" is 05-11-2015
Pattern "d '('E')' MMM, YYYY" is 5 (Thu) Nov, 2015
Pattern "w'th week of' YYYY" is 45th week of 2015
Pattern "EEEE, dd'th' MMMM, YYYY" is Thursday, 05th November, 2015
```

As you can see, repeated letters result in a longer form for an entry. For example, when you use `E` (which is the name of the day in the week), the result is "Thu", whereas using `EEEE` prints the full form of the day name, which is "Thursday".

Another important thing to notice is how to print text within the given pattern string. For that, you use text separated by single quotes, which is printed as is by `DateTimeFormatter`. For example, `'('E')'` prints "(Wed)". If you give an incorrect pattern or forget to use single quotes to separate your text from pattern letters in the pattern string, you get a `DateTimeParseException` for passing an "Illegal pattern."

Now, let's look at a similar example for creating custom time-pattern strings. Here is the list of important letters for defining a custom time pattern:

`a` (marker for the text a.m./p.m. marker)

`H` (hour: value range 0–23)

`k` (hour: value range 1–24)

`K` (hour in a.m./p.m.: value range 0–11)

`h` (hour in a.m./p.m.: value range 1–12)

`m` (minute

`s` (second)

`S` (fraction of a second)

`z` (time zone: general time-zone format)

For more letters and their descriptions, see the Javadoc for the `DateTimeFormatter` class. Listing 8-5 shows a program that uses simple and complex pattern strings to create custom time formats.

***Listing 8-5.*** `CustomTimePatterns.java`

```
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

// Using examples, illustrates how to use "pattern strings" for creating custom time formats
class CustomTimePatterns {
    public static void main(String []args) {
        // patterns from simple to complex ones
        String [] timeFormats = {
                "h:mm",          /* h is hour in am/pm (1-12), m is minute */
                "hh 'o''clock'", /* '' is the escape sequence to print a single quote */
                "H:mm a",        /* H is hour in day (0-23), a is am/pm*/
                "hh:mm:ss:SS",   /* s is seconds, S is milliseconds */
                "K:mm:ss a"      /* K is hour in am/pm(0-11) */
        };
        LocalTime now = LocalTime.now();
        for(String timeFormat : timeFormats) {
            System.out.printf("Time in pattern \"%s\" is %s %n", timeFormat,
                        DateTimeFormatter.ofPattern(timeFormat).format(now));
        }
    }
}
```

# Summary

Let's briefly review the key points from each certification objective in this chapter. Please read it before appearing for the exam.

**Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration**

- The Java 8 date and time API uses ISO 8601 as the default calendar format.
- The `java.time.LocalDate` class represents a date without time or time zones; the `java.time.LocalTime` class represents time without dates and time zones; the `java.time.LocalDateTime` class represents both date and time without time zones.
- The `java.time.Instant` class represents a Unix timestamp.
- The `java.time.Period` is used to measure the amount of time in terms of years, months, and days.
- The `java.time.Duration` class represents time in terms of hours, minutes, seconds, and fraction of seconds.

**Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values**

- `ZoneId` identifies a time zone; `ZoneOffset` represents time zone offset from UTC/Greenwich.
- `ZonedDateTime` provides support for all three aspects: date, time, and time zone.
- You have to account for daylight savings time (DST) when working with different time zones.
- The `java.time.format.DateTimeFormatter` class provides support for reading or printing date and time values in different formats.
- The `DateTimeFormatter` class provides predefined constants (such as `ISO_DATE` and `ISO_TIME`) for formatting date and time values.
- You encode the format of the date or time using case-sensitive letters to form a date or time pattern string with the `DateTimeFormatter` class.

**Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit**

- The enumeration `java.time.temporal.ChronoUnit` implements the `java.time.temporal.TemporalUnit` interface.
- Both `TemporalUnit` and `ChronoUnit` deal with time unit values such as seconds, minutes, and hours and date values such as days, months, and years.

A. The program gives a compiler error in the line marked with the comment `DEF`

B. The program gives a compiler error in the line marked with the comment `USE`

C. The code segment prints: `2015-02-31`

D. The code segment prints: `2015-02-03`

E. This code segment throws `java.time.DateTimeException` with the message `"Invalid date 'FEBRUARY 31'"`

5. **Consider this code segment:**

```
DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("EEEE", Locale.US);
System.out.println(formatter.format(LocalDateTime.now()));
```

**Which of the following outputs matches the string pattern `"EEEE"` given in this code segment?**

A. F

B. Friday

C. Sept

D. September

**Answers:**

1. The code segment prints: `-1`

   Here are the arguments to the `between()` method in the `Period` class:

   ```
   Period between(LocalDate startDateInclusive, LocalDate
   endDateExclusive)
   ```

   The first argument is the start and the second argument is the end, and hence the call `Period.between(now, babyDOB)` results in -1 (not +1).

2. C. `Instant` class

   The `Instant` class stores the number of seconds elapsed since the start of the Unix epoch (1970-01-01T00:00:00Z). The `Instant` class is suitable for storing a log of application events in a file as timestamp values.

   The `ZoneId` and `ZoneOffset` classes are related to time zones and hence are unrelated to storing timestamp values. The `Duration` class is for time-based values in terms of quantity of time (such as seconds, minutes, and hours). The `Period` class is for date-based values such as years, months, and days.

3. D. The code segment prints: `+08:00`

   Given a `ZonedDateTime` object, the `getOffset()` method returns a `ZoneOffset` object that corresponds to the offset of the time zone from UTC/Greenwich. Given that the time-offset value for the Asia/Singapore zone from UTC/Greenwich is +08:00, the `toString()` method of `ZoneOffset` prints the string "+08:00" to the console.

4. E. This code segment throws `java.time.DateTimeException` with the message `"Invalid date 'FEBRUARY 31'"`.

   The date value 31 passed in the call `LocalDate.of(2015, 2, 31);` is invalid for the month February, and hence the `of()` method in the `LocalDate` class throws `DateTimeException`.

   One of the predefined values in `DateTimeFormatter` is `ISO_DATE`. Hence, it does not result in a compiler error for the statement marked with the comment `DEF`. The statement marked with the comment `USE` compiles without errors because it is the correct way to use the `format()` method in the `DateTimeFormatter` class.

5. B. Friday

   `E` is the day name in the week; the pattern `"EEEE"` prints the name of the day in its full format. "Fri" is a short form that would be printed by the pattern `"E"`, but `"EEEE"` prints the day of the week in full form: for example, "Friday". Because the locale is `Locale.US`, the result is printed in English. The output "Sept" or "September" is impossible because `E` refers to the name in the week, not in a month.

# Index

## D



## E



## F

## G



## H



## I



## J, K

## L

## M



## N



## O