



the training specialist

bringing people and technology together

Oracle
Database

Oracle
e-Business

Unit Testing
Frameworks

Java
Technology

OOAD

Red Hat
Linux

Oracle
Development

MySQL

Maven

Web Development
HTML & CSS

Spring

SUSE
Linux

Oracle Fusion
Middleware

MariaDB

Hibernate

JavaScript
& jQuery

UNIX

LPI
Linux

Oracle Business
Intelligence

MongoDB

Acceptance Testing
Frameworks

PHP, Python
Perl & Ruby

Solaris, AIX
& HP-UX

Business Analysis
ITIL® & PRINCE2®

London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh
scheduled | closed | virtual training

www.stayahead.com +44 (0) 20 7600 6116 sales@stayahead.com

Java Programming 2

Although StayAhead Training Limited makes every effort to ensure that the information in this manual is correct, it does not accept any liability for inaccuracy or omission.

StayAhead Training does not accept any liability for any damages, or consequential damages, resulting from any information provided within this manual.



Java Programming 2

5 days training

Java Programming 2 Course Overview

The Java Programming 2 course is closely aligned with the Oracle Java SE 8 Programmer II exam.

The course focuses on the core language features and Application Programming Interfaces (APIs) you will use to design effective object-oriented and functional programming applications with the Java Standard Edition 8 Platform.

This course will teach you how to design and develop robust Java code that is easy to test and maintain and can be integrated into multiple applications.

The most important new topics introduced in Java SE 8 are covered, such as Functional Programming and Stream API, which allow you to use the most up to date techniques in your code.

Exercises and examples are used throughout the course to give practical hands-on experience with the techniques covered.

Skills Gained

The delegate will practise:

- Implementing code using inheritance and polymorphism
- Using design patterns to ensure robust design of classes
- Overriding key methods of the Object class to provide interoperability with collections and other APIs
- Understanding and using Generics within existing classes and create new Generic classes
- Creating and using collections including sets, maps and queues

- Selecting and incorporating standard functional interfaces in code
- Using Stream API to generate, filter, process and reduce stream data
- Writing Lambda expressions for use within code including with functional interfaces
- Declaring try-with-resources blocks and using AutoCloseable classes
- Working with Java SE8 Date/Time API classes
- Reading and writing data from/to the Console
- Managing files and directories
- Writing concurrent code and using Concurrent API
- Using parallel streams
- Building database applications with JDBC

Who will the Course Benefit?

The Java Programming 2 course is aimed at staff and consultants working as part of a Java development team to develop high-quality robust software. Roles include Programmers, Designers, Architects, Testers and anyone who needs a good understanding of the use of the Java language and infrastructure.

Course Objectives

By the end of the course delegates should be able to:

- Implement encapsulation, inheritance and polymorphism
- Override hashCode, equals, and toString methods from Object class
- Create and use singleton and immutable classes
- Create inner classes including static inner class, local class, nested class, and anonymous inner class
- Use enumerated types including methods, and constructors in an enum type
- Create and use Lambda expressions
- Create and use a generic class
- Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects
- Use java.util.Comparator and java.lang.Comparable interfaces
- Filter a collection by using lambda expressions
- Use the built-in interfaces included in the java.util.function package such as Predicate, Consumer, Function, and Supplier
- Develop code to extract data from an object using peek() and map() methods
- Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch
- Develop code that uses the Optional class
- Use try-catch and throw statements
- Test invariants by using assertions
- 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
- Read and write data from the console
- Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package
- Use Path interface to operate on file and directory paths
- Use Files class to check, read, delete, copy, move, manage metadata of a file or directory
- Use Stream API with NIO.2
- Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
- Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList
- Use parallel Fork/Join Framework
- Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections
- Read and set the locale by using the Locale object
- Create and read a Properties file

Examinations

The Java Programming 2 course helps delegates prepare for the following exam:

Oracle Certified Professional (OCP) Java SE 8 Programmer II exam (1Z0-809)

Attending this course will help delegates gain many of the skills and product knowledge requirements as outlined in the exam objectives. Studying this course and its associated reference materials can form part of the preparation to achieve certification. Experience, practice, self-study and aptitude are also key requirements for exam success.

Before taking any exam, ensure you have the recommended experience. The Oracle website lists all exam requirements and these are updated regularly.

Exams are not included as part of the course.

Requirements

Delegates attending this course should have experience programming with an object-oriented language, ideally Java. This knowledge can be obtained by attendance on the pre-requisite Java Programming 1 course.

Pre-Requisite Courses

- Java Programming 1

Follow-On Courses

- Java Web Development - Servlets & JSP
- Java Web Services
- Developing Applications with Java EE
- Unit Testing using JUnit

Note: Course technical content is subject to change without notice.

Table of Contents

Chapter 1: Java Class Design

Introduction.....	1- 3
Coupling and Cohesion.....	1- 5
Coupling	1- 5
Cohesion.....	1- 6
Composition	1- 7
Access Modifiers.....	1-11
private	1-11
Default or package	1-11
protected.....	1-12
public.....	1-12
Overloading and Overriding	1-13
Overloading.....	1-13
Overriding	1-15
Abstract Classes	1-17
Static and Final.....	1-18
Static	1-18
Final.....	1-20
Imports	1-23
The instanceof Operator.....	1-26
Virtual Methods	1-28
Variable Inheritance	1-30
@Override Annotation.....	1-32
Overriding the equals, hashCode, and toString methods from Object.....	1-33
toString()	1-33
equals()	1-34
hashCode()	1-39
Enums	1-43
Enums in switch statements	1-45
Enum Constructors.....	1-45

Chapter 1: Java Class Design (continued)

Nested Classes.....	1-47
Member Inner Classes	1-47
Local Inner Classes	1-49
Anonymous Inner Classes.....	1-52
Static Nested Classes.....	1-52
Summary	1-55
Exercises.....	1-56

Chapter 2: Java Design Patterns

Introduction.....	2- 3
Designing Interfaces.....	2- 4
Functional Programming.....	2- 6
Functional Interfaces	2- 6
Using Lambdas with Functional Interfaces	2- 7
Lambda Expression Syntax.....	2- 9
Predicate Functional Interface.....	2- 9
Polymorphism	2-11
Casting Object References	2-12
Java Design Principles.....	2-14
Encapsulating Data	2-14
IS-A Relationships.....	2-15
The HAS-A Relationship	2-15
Design Patterns	2-17
Singleton Pattern.....	2-18
Immutable Objects	2-19
Builder Pattern.....	2-21
Factory Pattern.....	2-24
Summary	2-27
Exercises.....	2-28

Chapter 3: Generics and Collections

Introduction.....	3- 3
Java Collections	3- 4
Arrays and ArrayList.....	3- 4
Searching and Sorting	3- 5
Wrapper Classes and Autoboxing.....	3- 6
The Diamond Operator	3- 7
Using Generics	3- 9
Generic Classes.....	3- 9
Generic Interfaces	3-10
Generic Methods	3-11
Bounds	3-12
Unbounded Wildcards	3-12
Upper-Bounded Wildcards	3-13
Lower-Bounded Wildcards	3-13
Lists, Sets, Maps, and Queues	3-15
Common Collection Methods.....	3-17
add()	3-17
isEmpty() and size()	3-17
clear()	3-18
contains()	3-18
List Interface.....	3-19
List Implementations	3-19
Big O Notation.....	3-20
List Methods	3-20
Iterating through a List.....	3-21
Set Interface.....	3-23
Set Implementations	3-23
Set Methods.....	3-24
Queue Interface.....	3-25
Map Interface	3-28
HashMap	3-28
Hashtable	3-29

Chapter 3: Generics and Collections (continued)

LinkedHashMap	3-29
TreeMap.....	3-29
Map Methods	3-29
Comparing Collection Types	3-31
Comparator and Comparable Interfaces	3-33
Comparable	3-33
Comparator	3-35
Searching and Sorting.....	3-36
Method References	3-38
Static Methods	3-39
Specific Instance Method	3-39
Runtime Instance Method.....	3-39
Constructor Method	3-39
Java 8 Methods.....	3-41
Conditional Removal	3-41
Update All Elements.....	3-41
Looping with Lambdas.....	3-42
Java 8 Map API Methods	3-42
merge.....	3-43
Summary	3-44
Exercises.....	3-46

Chapter 4: Functional Programming and Streams

Introduction.....	4- 3
Collections vs Streams	4- 3
Lambda Expression Variables.....	4- 4
Built-In Java Functional Interfaces.....	4- 5
Supplier	4- 6
Consumer and BiConsumer.....	4- 6
Predicate and BiPredicate.....	4- 7
Function and BiFunction	4- 8
UnaryOperator and BinaryOperator	4- 8

Chapter 4: Functional Programming and Streams (continued)

Optional Objects.....	4-10
Streams	4-12
Stream Source	4-13
Terminal Operations	4-14
count()	4-14
min() and max()	4-15
findAny() and findFirst()	4-15
allMatch(), anyMatch() and noneMatch()	4-16
forEach().....	4-16
reduce()	4-17
collect().....	4-18
Intermediate Operations.....	4-21
filter().....	4-21
distinct()	4-21
limit() and skip().....	4-21
map()	4-22
flatMap()	4-22
sorted()	4-23
peek()	4-23
Constructing a Stream Pipeline.....	4-24
Primitive Streams	4-26
Creating Primitive Streams.....	4-26
Optional and Primitive Streams	4-28
Summary Statistics.....	4-29
Functional Interfaces for Primitives.....	4-31
boolean Functional Interfaces	4-31
double, int, and long Functional Interfaces	4-31
Advanced Stream Concepts	4-33
Linking Streams to the Underlying Data.....	4-33
Chaining Optionals	4-33
Collecting Results	4-34
Basic Collectors	4-36

Chapter 4: Functional Programming and Streams (continued)

Collecting into Maps	4-37
Grouping, Partitioning, and Mapping	4-38
Summary	4-40
Exercises.....	4-42

Chapter 5: Dates, Strings and Localisation

Introduction.....	5- 3
Date and Time API Classes.....	5- 4
Creating Dates and Times	5- 4
Manipulating Dates and Times.....	5- 6
Period Class.....	5- 7
Duration Class	5- 7
Instant Class	5- 8
Daylight Savings Time	5- 9
String Class Functionality	5-11
StringBuilder	5-11
Internationalisation and Localisation	5-13
Specifying a Locale	5-13
Resource Bundles	5-14
Java Class Resource Bundles.....	5-16
Selecting the Resource Bundle	5-16
Formatting Numbers.....	5-18
Formatting.....	5-18
Parsing.....	5-19
Formatting Dates and Times.....	5-21
Summary	5-23
Exercises.....	5-24

Chapter 6: Exceptions and Assertions

Introduction.....	6- 3
Handling Exceptions	6- 4
Exception Types.....	6- 4
Try Blocks	6- 6
Throw and Throws	6- 6
Custom Exceptions.....	6- 7
Multi-catch Try Blocks	6- 8
Try-With-Resources	6- 9
AutoCloseable.....	6-10
Closeable.....	6-11
Suppressed Exceptions.....	6-12
Java Assertions	6-13
Assert Statement.....	6-13
Enabling Assertions	6-14
Reasons to Use Assertions	6-15
Summary	6-16
Exercises.....	6-17

Chapter 7: Concurrency

Introduction.....	7- 3
Threads	7- 3
Thread Types	7- 5
Thread Concurrency	7- 5
Creating Threads	7- 7
Polling with Sleep	7- 8
Using ExecutorService	7-10
Single-Thread Executor.....	7-10
Executor Shutdown	7-11
Submitting Tasks	7-12
Submitting Task Collections	7-13
Retrieving Results.....	7-14
Callable Interface	7-15

Chapter 7: Concurrency (continued)

Waiting for Tasks to Finish.....	7-16
Task Scheduling.....	7-17
Thread Pools.....	7-19
Synchronising Access to Data.....	7-22
Atomic Classes.....	7-23
Synchronized Blocks.....	7-26
Synchronized Methods	7-27
The Cost of Synchronization.....	7-28
Concurrent Collections	7-29
Rationale for using Concurrent Collections	7-29
Memory Consistency Errors	7-29
Concurrent Classes.....	7-30
Blocking Queues.....	7-32
SkipList Collections.....	7-33
CopyOnWrite Collections.....	7-34
Converting to Synchronised Collections.....	7-34
Parallel Streams	7-36
Creating Parallel Streams	7-36
parallel()	7-36
parallelStream()	7-36
Parallel Processing.....	7-37
Performance Improvements	7-38
Independent Operations	7-39
Stateful Operations	7-40
Processing Parallel Reductions	7-40
Combining Results with reduce().....	7-42
reduce() Method Arguments.....	7-42
Combing Results	7-43
Managing Concurrent Processes.....	7-44
CyclicBarrier.....	7-44
Fork-Join Framework.....	7-46
Recursion	7-46

Chapter 7: Concurrency (continued)

Fork/Join Task	7-48
Work Stealing.....	7-49
Join	7-50
RecursiveAction.....	7-50
RecursiveTask	7-52
Threading Problems	7-54
Liveness	7-54
Thread Deadlock	7-54
Starvation.....	7-55
Livelock.....	7-55
Race Conditions.....	7-55
Summary	7-57
Exercises.....	7-58

Chapter 8: IO

Introduction.....	8- 3
Files and Directories.....	8- 4
File System Terminology.....	8- 4
The File Class.....	8- 4
Creating a File Object.....	8- 5
File Objects	8- 6
IO Streams.....	8- 8
Stream Fundamentals.....	8- 8
Stream Names.....	8- 9
Byte Streams vs. Character Streams.....	8- 9
Input and Output	8- 9
Low- and High-Level Streams	8-10
Stream Base Classes	8-10
Java I/O Class Names	8-11
Common Stream Operations.....	8-12
Closing the Stream	8-12
Flushing the Stream	8-12

Chapter 8: IO (continued)

Marking the Stream	8-13
Skiping over Data.....	8-14
Using Data Streams	8-15
FileInputStream and FileOutputStream	8-15
BufferedInputStream and BufferedOutputStream	8-16
FileReader and FileWriter	8-17
BufferedReader and BufferedWriter.....	8-17
PrintStream and PrintWriter	8-18
ObjectInputStream and ObjectOutputStream	8-19
The Serializable Interface	8-20
Serialising and Deserialising Objects	8-20
User IO.....	8-23
Pre-Console.....	8-23
Console	8-23
reader() and writer()	8-24
format() and printf().....	8-24
flush()	8-25
readLine().....	8-25
readPassword().....	8-25
Summary	8-26
Exercises.....	8-27

Chapter 9: NIO.2

Introduction.....	9- 3
Path Interface	9- 4
Path Factory and Helper Classes.....	9- 4
Creating Path Objects.....	9- 5
Using Paths.....	9- 5
FileSystem Object.....	9- 6
Converting File objects.....	9- 6
Paths and Files Classes	9- 8
Path Objects.....	9- 8

Chapter 9: NIO.2 (continued)

toString(), getNameCount(), and getName()	9- 8
getFileName(), getParent(), and getRoot()	9- 9
isAbsolute() and toAbsolutePath()	9-10
subpath()	9-11
relativize()	9-11
resolve()	9-12
normalize()	9-13
toRealPath()	9-13
Files Class.....	9-15
exists()	9-15
isSameFile()	9-15
createDirectory() and createDirectories()	9-15
copy()	9-16
move()	9-17
delete() and deleteIfExists()	9-17
readAllLines()	9-17
File Attributes.....	9-19
Basic File Attributes	9-19
isDirectory(), isRegularFile(), and isSymbolicLink()	9-19
isHidden()	9-20
isReadable() and isExecutable()	9-20
size()	9-21
getLastModifiedTime() and setLastModifiedTime()	9-21
getOwner() and setOwner()	9-22
File Attribute Views	9-23
Introducing Views	9-23
Reading Attributes.....	9-24
BasicFileAttributes	9-24
Modifying Attributes.....	9-25
BasicFileAttributeView.....	9-25
NIO.2 Stream Methods	9-27
Directory Walking	9-27

Chapter 9: NIO.2 (continued)

Search Strategies	9-28
Walking a Directory	9-28
Circular Filesystem Paths	9-29
Searching a Directory.....	9-29
Listing Directory Contents.....	9-30
Printing File Contents.....	9-31
IO File and NIO.2 Methods.....	9-31
Summary	9-33
Exercises.....	9-34

Chapter 10: JDBC

Introduction.....	10- 3
JDBC Interfaces	10- 5
Connecting to a Database.....	10- 6
Building the JDBC URL.....	10- 6
Getting the Database Connection.....	10- 6
Getting a Statement.....	10- 7
ResultSet Type	10- 8
ResultSet Concurrency Mode	10- 8
Executing SQL Statements.....	10- 9
PreparedStatement	10-10
Using a ResultSet	10-11
Reading a ResultSet.....	10-11
Getting Column Data	10-11
Scrollable ResultSet	10-13
Closing Database Resources.....	10-15
SQL Exceptions.....	10-16
Summary	10-17
Exercises.....	10-19

Chapter 11: Practical Exercises

Section 1 (Java Class Design)	11- 3
Exercise 1.1	11- 3
Exercise 1.2	11- 7
Exercise 1.3	11- 7
Exercise 1.3a (Challenge Exercise).....	11- 8
Exercise 1.4	11- 8
Exercise 1.5	11-10
Exercise 1.5a (Challenge Exercise).....	11-10
Exercise 1.5b.....	11-11
Exercise 1.6	11-11
Exercise 1.6a (Challenge Exercise).....	11-12
Section 2 (Java Design Patterns).....	11-13
Exercise 2.1	11-13
Exercise 2.2	11-14
Exercise 2.3	11-14
Exercise 2.4	11-15
Exercise 2.5	11-16
Exercise 2.6	11-17
Exercise 2.7	11-17
Section 3 (Generics and Collections)	11-19
Exercise 3.1	11-19
Exercise 3.2	11-20
Section 4 (Functional Programming and Stream API)	11-21
Exercise 4.1	11-21
Section 5 (Dates, Strings and Localisation)	11-22
Exercise 5.1	11-22
Exercise 5.2	11-22
Exercise 5.3 (Challenge exercise)	11-22
Section 6 (Exceptions and Assertions)	11-24
Exercise 6.1	11-24
Exercise 6.2 (Challenge exercise)	11-24
Section 7 (Concurrency).....	11-25

Chapter 11: Practical Exercises (continued)

Exercise 7.1	11-25
Section 8 (IO)	11-26
Exercise 8.1	11-26
Exercise 8.2	11-26
Section 9 (NIO.2)	11-27
Exercise 9.1	11-27
Section 10 (JDBC)	11-28
Exercise 10.1	11-28
Exercise 10.2	11-28
Exercise 10.3 (Challenge exercise)	11-28

CHAPTER 1

Java Class Design

Introduction

The basic unit of software in Java is the class and so the design of classes and how they work together in an application is a critical factor in designing quality software.

Some of the key aims of java class design are that classes should be:

- Maintainable and easily updated
- Reusable in multiple contexts, including other applications
- Fully encapsulated with a clearly defined interface
- Highly cohesive with a single responsibility
- Modular with low dependency on other modules (loose coupling)
- Easily extensible with minimum impact

Class design includes three Java constructions which are closely related to each other and which are declared with the **class**, **interface** and **enum** keywords.

NOTE

There is a well-known set of five Object-Oriented design principles known as SOLID which are:

- **Single Responsibility Principle**
- **Open Closed Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**

More details may be found at: butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

These principles apply to OO design irrespective of the language used. This section of the course focuses on the design of classes within the Java language.

In this section, we will start by reviewing some core design principles which you would have learned about in the Java Programming 1 course.

We will then look at the uses of the instanceof operator to compare objects and test for the IS-A relationship between objects, classes and interfaces.

We will also look at polymorphism and specifically how it applies to virtual method invocation using classes from the same inheritance hierarchy.

The `@Override` annotation is recommended as a good practice when coding sub classes and we will look at how and when it should be used.

The Object class, which is the ultimate super class of all other classes, has some key methods which are inherited by all classes and may need to be overridden. These are `equals()` `hashCode()` and `toString()`. We will work through when, why and how they should be overridden and what needs to be considered in order to maintain consistent behaviour.

Abstract classes are a key component of well-designed class hierarchies and we will look at how they can be best used in different circumstances.

Enums are used when a limited list of values is required and we will look at some of the advanced facilities provided by this flexible construction.

Finally, we will look at nested classes which have a number of uses in Java programming and explain when and why each variation can and should be used and the rules that must be applied when using them.

Coupling and Cohesion

Coupling and cohesion are closely related to the quality of an OO design. In general, good OO design calls for loose coupling and avoids tight coupling where possible. Additionally, well-designed Java classes should have high cohesion.

Some of the design goals for an OO application which are addressed by coupling and cohesion are the ability to:

- change code easily and with low impact
- find the most appropriate component to perform a specific task
- produce a design that is logical and easily understood

Coupling

When two components or classes are tightly coupled, they are highly dependent on each other.

The worst kind of coupling is when one class (A) is dependent on the implementation of another class (B). This means that assumptions are made in A's code about the code inside B. We call this 'coding to the implementation' rather than 'coding to the interface' which is preferred. If A only depends on the interface of B, i.e. its methods and their parameters and return types, then the internal implementation of B can be completely re-written, perhaps to improve performance or to correct a fault and this will have no effect on A whatsoever. However, where A has been coded to the implementation of B and that implementation changes, the effects are unpredictable and changing B may break the system or change its behaviour with a side-effect. This will need to be identified first and will probably result in the need to change the code in class A.

Any time when a change in one class is highly likely to require a change in another class they are said to be tightly coupled.

Other examples of tight coupling which may apply to Java classes are:

- Time: Both components need to be active at the same time (also known as synchronous communication)
- Location: Information about the physical location or address of one component is hard-coded in the other
- Name: The name of a component is hard-coded in the other
- Data types: Where highly specific data types are used in method invocation
- Operations: Where there are many specific operations with different purposes rather than a single, general-purpose operation that is less likely to change
- State: Where one component holds state (data) in variables between method calls
 - variables are part of the implementation details

Cohesion

Cohesion is a measure of how closely related all the aspects of a component or class are to each other.

High cohesion can be promoted by following a detailed process of OO analysis and design.

The aspects of a class that determine cohesion are:

- its methods or operations
- the data it deals with

These aspects are described using the term ‘responsibility’. A class is said to be responsible for the operations it performs and if these are closely related then it is highly cohesive. Some classes predominantly deal with data and so if all the data dealt with by a class are related then again it has high cohesion.

Highly cohesive classes make good sense for a number of reasons. A highly cohesive class should be easy to name because its operations or data are related to a real world object or concept.

For example, in a Human Resources system a class called ‘Employee’ deals with all the data required to define a real-world employee. This makes it easy to find and re-use because its name matches its functionality. Another class might be called MatchSkills and it is easy to work out what it does. This last example is an example of the application of the Single Responsibility Principle in which classes only do one thing.

One result of high cohesion is that classes are only likely to change if some aspect of the business domain changes and then it will be a simple matter to identify the class that needs to be changed.

Composition

Composition in Java is the design technique to implement the HAS-A relationship in classes. Object composition may also be used as an alternative to inheritance (IS-A) in Java for code reuse through a technique called ‘delegation’.

Java composition is achieved by using instance variables that refer to other objects. For example, an Employee is assigned to a JobRole.



Figure 1-1 Simple Composition

Here is some example code to illustrate Java composition:

```

package hr;
public class JobRole {
    private String name;
    private long salary;
    private int id;
    // Constructors and other methods...
}

package hr;
import java.time.LocalDate;
public class Employee {
    private JobRole jobRole; // HAS-A relationship
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;
    // Other variables and methods...
}
  
```

Simple composition occurs when two classes have a strong relationship such as that illustrated above between Employee and JobRole. Sometimes this relationship is discovered during analysis and design in which case the developer simply implements the design by including a member variable of the appropriate type. On other occasions it may emerge during coding where the developer realises that a member variable is complex enough to be an object and have its own methods and variables.

Composition may involve a one-to-one relationship, as above, or a one to many relationship.

A JobRole is likely to require more than one skill. If skill is complex enough to be an object in its own right, let's say it has different levels and may require some academic or vocational qualifications, then there would be a one-to-many relationship between JobRole and Skill.

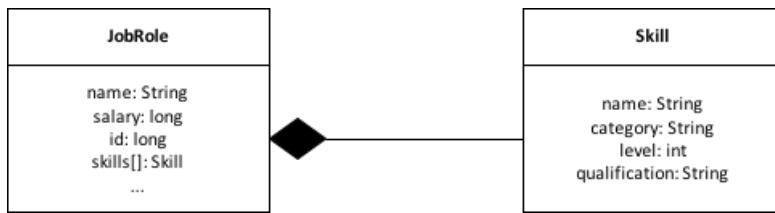


Figure 1-2 One to Many Composition

Here is some example code to illustrate Java one to many composition:

```

package hr;
public class JobRole {
    private String name;
    private long salary;
    private int id;
    private Skill[] skills;
    // Constructors and other methods...
    public addSkill(String name, int level){ ... }
}

package hr;
public class Skill {
    private String name;
    private String category;
    private int level;
    private String qualification;
    // Constructors and other methods...
}
  
```

A more complex example of composition that includes delegation is provided by **PaymentType** which defines the method and business rules for paying entities, such as **Employee**. In our design there are already multiple ways of paying people – **Salary**, **HourlyRate**, **Commission** and **FixedPrice** – and in the future, there could be more required. There are also multiple entities that will be paid – **Employee**, **Supplier**, **Contractor**, **Agency** and **Broker**.

Imagine designing an inheritance hierarchy which contains every possible combination of these! Much more flexibility is provided by delegation which uses composition to create a combined object, for example a salaried employee.

This is often called the Delegation design pattern as it is a well-established solution to a common problem.

So that the different variations of **PaymentType** can be substituted for each other, they must be part of the same inheritance hierarchy. However, the entities that will be paid may be completely unrelated.

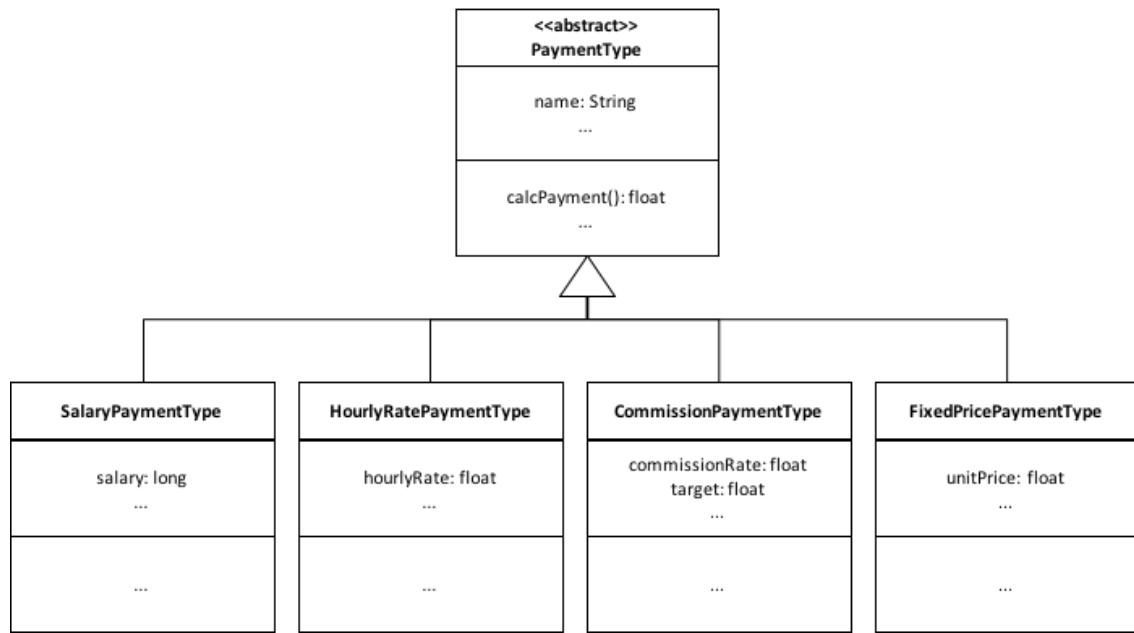


Figure 1-3 Delegate Pattern Inheritance Hierarchy

Defining the abstract class `PaymentType` allows us to include this as a variable in our various entities that are to be paid:

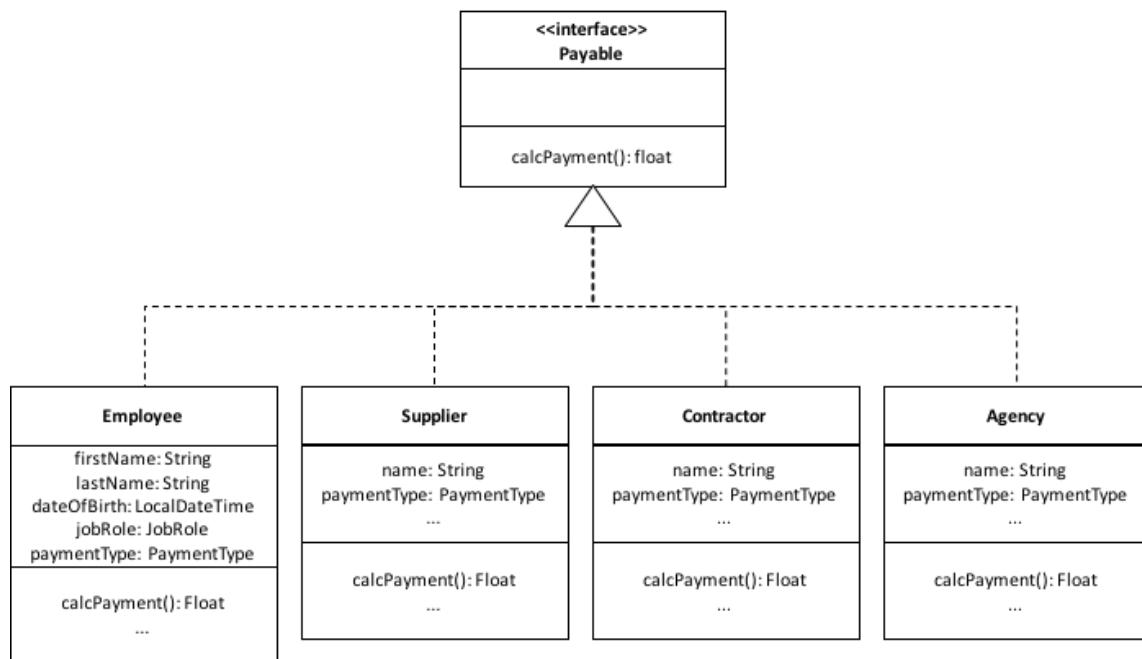


Figure 1-4 Use of Composition in Delegation Pattern

Note that an interface has been defined with the `calcPayment()` method which ensures that all the objects contain the method.

When the `calcPayment()` method of `Employee`, `Supplier` etc. is called, the object delegates to the object referenced by its member `PaymentType` variable by calling its `calcPayment()` method. The delegating object does not need to know which sub class is being referenced.

The PaymentType object may be provided as a constructor argument, if it is immutable, or via a setter method if it may be changed through the life of the delegating object (Employee, Supplier etc.) in the application.

Here is the Employee class which accepts a PaymentType via the constructor:

```
package hr;
import java.time.LocalDate;
public class Employee implements Payable {
    private String firstName;
    private String lastName;
    private String lastName;
    private JobRole jobRole;
    private PaymentType paymentType;
    public Employee(String firstName, String lastName, String lastName,
                    JobRole jobRole, PaymentType paymentType) {
        this.firstName = firstName;
        ...
        this.paymentType = paymentType;
    }
    ...
    public float calculatePayment() { // Implementing Payable interface
        return paymentType.calculatePayment() // Delegate to paymentType
    }
    ...
}
```

As long as there is no setter then the paymentType cannot be changed but to allow for changes in another class, perhaps Contractor, a setter method could be provided. The delegation works in the same way.

Composition includes any design where a class holds a reference to another class. Usually it is not used for reference variables of types provided by the Java API such as ArrayList or String. Many other design patterns such as Decorator are based on composition.

Access Modifiers

Access modifiers are the keywords that define which other classes can access methods and instance variables.

The four access modifiers are:

- private
- protected
- public
- (default)

These all change the default level of access.

The default access level does not use a keyword and is assigned to a method or instance variable when neither private, protected, nor public is used, and the area is left blank.

NOTE

There was a default keyword introduced in Java 8 for interfaces. That keyword is NOT an access modifier.

Access modifiers are important, allowing the implementation details to be hidden in a class as part of proper encapsulation.

private

The most restrictive modifier. Any method or instance variable that is marked as private can only be accessed by other methods in the same class. Subclasses cannot access instance variables or methods that are private.

For example:

```
private String surname;  
private int calculate() { }
```

Default or package

The default, or "friendly" access level is the second most restrictive. It is often referred to as package. This allows access to methods and instance variables from code in the same package (folder).

For example:

```
String surname;  
int calculate() { }
```

protected

The third most restrictive modifier. It adds the ability of subclasses outside of the package to access its methods or instance variables to the package level, so relatives can access too.

For example:

```
protected boolean processing;  
protected int calculate() { }
```

public

Public is the least restrictive modifier. No restriction is applied. Any method can access a public method or instance variable regardless package, superclass or location. It is effectively universal, open access.

For example:

```
public int houseNumber;  
public String houseName() { }
```

The accessibility of class members with the four access modifiers is shown in the following table:

From:	private	(default)	protected	public
Same class	✓	✓	✓	✓
Other class in same package		✓	✓	✓
Other package sub class			✓	✓
Other package				✓

Overloading and Overriding

These two techniques are similar in name but completely different in use and effect. They may be used in combination if the situation is appropriate. Overloading is used in a single class to provide multiple signatures for the same method. Overriding works with two classes in the same inheritance hierarchy and allows a sub class to have a different implementation for an identical method with the same signature.

It is possible to make the mistake of overloading a method when the intention was to override. This is done by unintentionally changing the signature in the sub class. The `@Override` annotation makes the compiler check for this error.

Overloading

Overloading a method allows two or more methods in the same class to share the same name. Method overloading can sometimes be convenient and saves having to think up new names for methods that do the same thing but with different types and or numbers of arguments.

```
public class Calc {
    public int add( int a, int b ){
        return a + b;
    }
    public int add( int a, int b, int c ){
        return a + b + c;
    }
}
```

The compiler determines the method being called by matching the types and number of arguments in the parameter list.

NOTE

The names of the arguments are irrelevant to the compiler when deciding which method to call.

In java, method overloading is not possible by simply changing the return type of the method because of the ambiguity that may arise.

Let's see how ambiguity may occur:

```
class Calc {
    int add(int a, int b){ return a + b; }
    double add(int a, int b){ return a + b; }
}
class TestOverloading {
    public static void main(String[] args){
        System.out.println(Calc.add(11, 11)); //ambiguity
    }
}
```

Here the compiler is not even given a variable to which to assign the return value so has no way to distinguish between these two methods.

The result is a compilation error.

The java main method may be overloaded:

```
class TestOverl oadi ng{
    public static void mai n(String[] args){
        System.out.println("mai n wi th String[]");
    }
    public static void mai n(String args){
        System.out.println("mai n wi th String");
    }
    public static void mai n(){
        System.out.println("mai n wi thout args");
    }
}
```

However, the JVM will call the main with a String array as its only argument. The other methods may be called from other code.

Type promotion may be used to determine which method the compiler will call.

Here is a summary of primitive type promotion:

Data type	May be promoted to:
byte	short
short	int
char	int
int	long, float or double
long	float or double
float	double

Here is an example of method overloading involving type promotion:

```
class Cal c {
    void sum(int a, long b){
        System.out.println(a + b);
    }
    void sum(int a, int b, int c){
        System.out.println(a + b + c);
    }
    public static void mai n(String args[]){
        Cal c cal c = new Cal c();
        obj .sum(20, 20); // second int literal will be promoted to long
        obj .sum(20, 20, 20);
    }
}
```

Where matching type arguments are found, no promotion is necessary.

```
class Calc {
    void sum(int a, int b){
        System.out.println(a + b);
    }
    void sum(long a, long b){
        System.out.println(a + b);
    }
    public static void main(String args[]){
        Calc calc = new Calc();
        obj.sum(20, 20); // sum() with int arguments will be called
    }
}
```

Even with type promotion there is scope for ambiguity

```
class Calc {
    void sum(int a, long b){
        System.out.println(a + b);
    }
    void sum(long a, int b){
        System.out.println(a + b);
    }
    public static void main(String args[]){
        Calc calc = new Calc();
        obj.sum(20, 20); // ambiguity
    }
}
```

In this case, there is a compiler error.

When multiple overloaded methods are present, Java looks for the closest match first.

The compiler tries to find the following:

- Exact match by type
- A matching superclass type
- Primitive type promotion
- Converting to an autoboxed type
- Varargs

If no match can be found there will be a compiler error.

Overriding

Where a class inherits a method from a super class, then it is possible to override the method, as long as the method is not marked as final.

This means that the sub class method has a different implementation and so the behaviour can be changed as required.

There are some rules for overriding:

- The argument list must be identical to the overridden method
- The return type must be the same or a subtype of the return type of the overridden method
- The access cannot be more restrictive than the overridden method
- Instance methods can be overridden only if they are inherited by the subclass
- `final` methods cannot be overridden
- `static` methods cannot be overridden but can be re-declared
- If a method cannot be inherited, then it cannot be overridden
- A subclass cannot inherit a method if it is inaccessible
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not
- An overriding method must not throw checked exceptions that are new or broader than the ones declared by the overridden method
- Constructors cannot be overridden

Here is a class that will be subclassed:

```
package finance;
public class PaymentType {
    // member variables and constructors
    protected float calcPayment() { ... }
    float depreciate(float cost, float value, int period) { ... }
    private equals(float amount, int level) { ... }
}
```

Here is a subclass that overrides `calcPayment()`:

```
package hr;
public class SalaryPaymentType {
    public float calcPayment() { ... }
}
```

Note that `SalaryPaymentType` does not inherit nor have access to the `equalise()` or `depreciate()` methods and therefore is unable to override them.

When overriding it may be required to access the super class version of the method. If so, the `super` keyword may be used:

```
package finance;
public class FractionalPaymentType {
    private float fraction;
    public float calcPayment() {
        return super.calcPayment() * fraction;
    }
}
```

Abstract Classes

Abstract classes are used to provide a base class that can be subclassed but not instantiated. This makes sense if it is so general that it does not have any meaning in the real world and more specific detail is required to make it useful.

All classes (and interfaces) can be completely empty and will compile. Abstract classes can also contain abstract and concrete methods as well as instance variables. These will be inherited and the subclass could override the methods.

Abstract classes may subclass other abstract classes as well as concrete classes. They may also implement interfaces.

Only abstract classes (and interfaces) may contain abstract methods so any concrete class inheriting an abstract method must provide implementation code for it.

```
package finance;
public abstract class PaymentType {
    public abstract float calculatePayment();
}
```

This class has an abstract method and so any concrete (non-abstract) class inheriting from it needs to provide some implementation code.

```
package hr;
public class SalaryPaymentType {
    public float calculatePayment(){
    }
}
```

This method does not do anything but the presence of the curly braces is good enough for the compiler.

If an abstract class inherits an abstract method from another class or an interface then it does not need to implement it. However, as soon as a concrete class appears in the hierarchy as a sub class, it must implement the abstract method and provide an implementation.

Static and Final

The static keyword may be used with members (variables and methods) and inner classes to make them belong to the class rather than an instance of the class.

The final keyword means that the class, method or variable can never be modified.

In combination, static and final are used to create constants where a meaningful name can be given to a number or other value that will be used in code, e.g. PI is a static final constant in the Math class.

Static

This means there is no need to instantiate the class to access the member or inner class.

For example, a class with a static variable has only one copy of that variable irrespective of how many instances there are, even if there are none.

```
public class HrUtilities {  
    public static int employeeCount;  
    public static calculateHolidays(LocalDateTime startDate, Grade grade) {  
        // implementation code  
    }  
}
```

employeeCount in this example is initialised to zero when the class is loaded by the JVM and can be read and set by any code using the following syntax:

```
System.out.println(HrUtilities.employeeCount);  
HrUtilities.employeeCount = 1792;
```

All code within the application can see and set the value so if a new employee is added then this variable can be incremented.

Confusingly, static variables may also be accessed using a reference variable of the class type:

```
HrUtilities hru = new HrUtilities();  
System.out.println(hru.employeeCount);
```

Although this looks very much as if we are accessing an instance variable called employeeCount, it is really a class variable of HrUtilities.

Static methods work in a similar way.

A very common use is with Singleton pattern or Factory Method Pattern classes.

Singleton classes do not have public constructors but expose a public static function, often called `getInstance()`, that returns a reference to a shared object. With singletons there is only one object however many times you call `getInstance()`.

Factory Method objects decide at run time which exact class will be used for your object. Calendar, for example, exposes a public static method called `getInstance()` that returns a

sub class of the abstract Calendar class, usually a GregorianCalendar object. LocalDate also has Factory methods such as of(year, month, day) which returns an object of type LocalDate.

Utility classes usually only have static methods. For example the Files class in the java.nio package has static methods for dealing with Path objects that represent files and directories. The Files class cannot be instantiated and is never used to represent anything in the business domain but rather is a single point for all the commands users of the file system may wish to issue. These include copy, move, delete, create, read, write and several commands to test the attributes of a file or directory. All these methods can be used without needing to instantiate a class using the new keyword. In fact, Files cannot be instantiated as its declaration contains the keyword final (which is explained in the next section).

Probably the first place Java programmers encounter the static keyword is the main method which is called by the JVM.

```
public static void main(String[] args) { ... }
```

NOTE

JVM will only call a main method in a class if it matches the correct signature.

However, the public and static keywords may appear in any order so the following are equivalent:

- **public static void main(String[] args) { ... }**
- **static public void main(String[] args) { ... }**

Also, the name of the argument does not have to be 'args' and alternative array declaration syntax may be used including varargs, so the following are also legitimate:

- **public static void main(String[]x1_7) { ... }**
- **public static void main(String args[]) { ... }**
- **public static void main(String... args) { ... }**

The main method has to be public as it is being called by the JVM and not from another class in the same package as the class containing main. It also has to be static as there is no instance of the class at the point when main is called by the JVM.

Another time you will see the static keyword is when it is used in combination with the import keyword. We will see how this works in the section called Imports below.

There is also a construct called a static initialiser block which is a block of code within a class but outside of any method or constructor which is prefixed with the static keyword:

```
package hr;
import java.time.LocalDate;
public class HrUtilities {
    public static LocalDate today;
    public static int lengthOfMonth;
    static{
        today = LocalDate.now();
```

```
    lengthOfMonth = today.lengthOfMonth();
}
...
}
```

Static initialiser blocks are run when the class is loaded by the JVM. Since there is no accessible constructor where code can be provided to run at class load, this is the only option for programmers to have code run at this time.

Finally, a word about where static members may be used and what they have access to. static variables and methods are available all the time, without any need to instantiate a class. Therefore, static members may be accessed from other static code. Static code means either a static method, an in-line initialisation of a static variable or a static initialiser. Static members may also be accessed by non-static code. Non-static code means instance methods, in-line instance variable declarations, constructors and non-static initialiser blocks. In summary, static members are accessible to all code since they are always available from the start of the application.

However, the same is not true of non-static (instance) members. These only exist once an instance has been created. So it is not possible for a static method like main, for example, to access a non-static variable or method in the same class:

```
package hr;
import java.time.LocalDate;
public class HrTest {
    public static void main(String[] args){
        System.out.println(getDate()); // LINE 5
    }
    private LocalDate getDate(){
        return LocalDate.now();
    }
}
```

This causes a compiler error at line 5 as getDate() is not static but is being called from static code. When main() runs there is no instance of HrTest. If line 5 were replaced with the following code there would be no error:

```
System.out.println(new HrTest().getDate()); // LINE 5
```

This is because an instance has been created anonymously and the instance method called.

In summary, instance code can access static and instance members but static code can only access other static members.

Final

Final may be applied to classes, methods and variable declarations to limit the use of the code unit being declared.

If a class is declared final then it may not be sub-classed.

The following code will cause a compiler error:

```
public final class HrUtilities { ... }
public class HrUtilSub extends HrUtilities { ... }
```

At the method level, final simply means the class cannot be overridden. There may be many subclasses that inherit this method but none may change the implementation by overriding.

The following code will cause a compiler error:

```
public abstract class BusinessUnit {
    ...
    private ArrayList<Post> posts;
    public final void addPost(Post){ ... }
    ...
}
public class Team {
    ...
    public void addPost(Post){ ... }
    ...
}
```

NOTE

Although methods declared as final cannot be overridden they may still be overloaded using different parameter types or numbers.

At the level of declaring variables, final means that the content of a variable may not change.

As a simple example, the following would cause a compiler error:

```
public abstract class BusinessUnit {
    ...
    public static final int MAX_POSTS = 50;
    ...
    public void increaseMaxPosts(int increment) {
        MAX_POSTS += increment;
    }
    ...
}
```

In this example, MAX_POSTS is intended to be a constant which is why it is named with upper case letters.

The above example used a primitive int variable. The use is almost identical for a wrapper class such as Integer. If MAX_POSTS had been declared as an Integer variable there would still be a compiler error but the reason is slightly different.

Integer and all wrapper classes (Short, Byte, Float etc.) are immutable. When you change the value of a wrapper class it involves multiple operations. The primitive is unboxed, the calculation is performed with primitive arithmetic and the result is boxed into a new object (instance of the wrapper class). The new object is now assigned to the original variable and the old object is de-referenced and marked for garbage collection. This

means that the contents of the reference variable would have to change as it now needs to point to a new object.

The same is also true for String, which is also immutable, so the following causes a compiler error:

```
public abstract class BusinessUnit {  
    ...  
    private final String name;  
    ...  
    public void extendName(String nameExtension) {  
        name += nameExtension;  
    }  
    ...  
}
```

However, newcomers to Java are sometimes surprised to find that the following code does not result in an error:

```
public class Employee {  
    ...  
    public float salary;  
    ...  
}  
public class Test {  
    public static void main(String[] args) {  
        final Employee employee = new Employee(...);  
        employee.salary = 15000.00f;  
    }  
}
```

Even though the variable called employee is declared final, its properties can be changed. The above example breaks the rules of encapsulation by making salary public. However, the same logic would apply if it were properly encapsulated and a setter were provided to make the change.

The final keyword affects the variable being declared not any objects that the variable might point to. A reference variable declared final may only ever point (refer) to one object. Reference variable contain references and this is what cannot be changed if the variable is declared as final.

The final keyword may be used wherever a variable is being declared. This includes method parameters which become local variables inside the method. Since it is considered bad practice to change the values of method parameters, some people would prefer them all to be declared as final to prevent this:

```
public void setSalary(final float salary) { ... }
```

We will see a little more about the use of final relating to inner and anonymous classes later in this chapter and encounter the term ‘effectively final’ where the compiler makes a variable final automatically under certain circumstances.

Imports

The import keyword tells the java compiler that you intend to use a member of a package in your code and allows the member to be named without prefixing it with the package name. A member of a package is a class, interface, enum, Exception or Error which has been declared using a package statement.

For example:

```
package hr;
import java.util.ArrayList;
public class Employee {
    private ArrayList<Skill> skills;
    ...
}
```

This allows you to use the class ArrayList as if it were local to the hr package.

An alternative to the code shown above without using import would be:

```
package hr;
public class Employee {
    private java.util.ArrayList<Skill> skills;
    ...
}
```

This code is more long-winded and harder to read and this is the reason for the import keyword. The bytecode produced by the compiler is the same. Importing names does not use more memory and if an imported class is never used then the compiler will simply ignore the import.

NOTE

Classes in the java.lang package, such as String and Integer, do not need to be imported. Classes from every other package, including those you have written, do need to be imported to avoid fully qualifying them in your code.

It is possible to import all the classes, interfaces and enums in a package using the * wildcard:

```
import java.util.*;
```

This means that all the classes in the java.util package may be used without prefixing with the package name. This is therefore an alternative to the following:

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Random;
```

It is shorter to use the wildcard but less clear in its intention, particularly with a package like util where the members do not have much in common, as with the three listed above.

The other problem with using the wildcard is that there may be a clash between two classes with the same name but which are in different packages being imported.

For example, both java.util and java.sql contain a class called Date. Therefore the following code would expose this clash and cause a compiler error:

```
package hr;
import java.util.*;
import java.sql.*;
public class DateTest {
    private Date date;
    ...
}
```

One other point to note is that the import statements can only exist between the package statement, if it exists, and the class (or interface or enum) declaration. It is illegal to have an import statement before a package statement or after a class statement. package and import statements apply to the whole file and may affect multiple classes if present.

The import statement provides a way to treat members of a package as if they were local to the package of the classes in the file containing the import statement. This may be one or all members of the package.

Another use of import, in combination with the static keyword, is to treat the static members of another class as if they were members of the class being declared. For example, here the static members of java.lang.Math may be used as if local to the HrUtilities class:

```
package hr;
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
public class HrUtilities {
    public static float calculateCircleArea(float radius) {
        return PI * radius * radius;
    }
    public static float calculateHypotenuse(float left, float right) {
        return sqrt((left * left) + (right * right));
    }
}
```

There is also a wildcard available for this type of import:

```
import static java.lang.Math.*;
```

This statement imports all the static members (methods and variables) of the java.lang.Math class and allows them to be used as if they were local static members of the class being declared. So, the two lines:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

Could be replaced with:

```
import static java.lang.Math.*;
```

The last point to note here is that although these imports are called ‘static imports’, the syntax is always ‘import static’.

The instanceof Operator

`instanceof` is a comparison operator just like `<` and `==` and like these other more mathematical comparison operators it returns Boolean true or false.

The left operand is an object and the right operand is a class name.

Here is an example:

```
a instanceof B
```

This expression returns true if the reference to which `a` points is an instance of class `B`, a subclass of `B` (directly or indirectly), or a class that implements the `B` interface (directly or indirectly).

Let's have a look at an example with several classes and interfaces:

```
package hr;
public abstract class BusinessUnit { ... }
public class Team extends BusinessUnit { }
public class Board extends BusinessUnit { }
```

You see that `Team` is a subclass of `BusinessUnit` but not of `Board` so the following expression returns true:

```
BusinessUnit bu = new Team();
boolean b = bu instanceof BusinessUnit; // true
```

Whereas the following expression returns false:

```
b = bu instanceof Board; // false
```

What about the following?

```
b = bu instanceof Team; // ??
```

This is true of course! We know that `bu` is an instance of the `Team` class.

The best way to think about this is as a test for the IS-A relationship which exists between subclasses and interface implementors at any level. This means that if a superclass of `x` IS-A `y` then `x` IS-A `y`. Also if a superclass of `x` implements the `z` interface then `x` IS-A `z`. That means that every class must pass the `instanceof` test if `Object` is the right-hand operand.

The above tests compile ok even though we knew in advance that the second example would return false. However it was possible that what was contained in the `bu` variable was a reference to something that passed the `instanceof` test with `Board`.

You can make it impossible though:

```
Team t = new Team();
b = t instanceof Board; // Compiler error!
```

This gives a compiler error as a variable declared as of type `Team` can never contain a reference to an object of type `Board` as no subclass of `Team` can be a subclass of `Board`.

However, if you are testing against an interface then the compiler cannot tell. Interfaces can be implemented further down the hierarchy in the future

The result is that you can use the instanceof operator to test if a variable may contain a reference to an object. If you try to assign a variable to an object that is not compatible (passes the IS-A or instanceof test) then an exception will be thrown at runtime by the JVM.

When assigning an object referenced by one variable (variable1) to another variable (variable2) that is below it in the class inheritance hierarchy, it is necessary to use an explicit cast:

```
BusinessUnit bu = new Test();
Test t = bu; // Compiler error
```

This does not work because the compiler is not convinced that bu contains a reference to an object of type Test even though the assignment is on the line above. The compiler considers the possibility that you may assign a reference to an object of another class, Board for instance, and then the assignment to t would cause an exception. So you have to use the explicit cast to Test (or a subclass) as follows:

```
Test t = (Test)bu; // Ok now
```

This code may be interpreted as a promise by the programmer that whatever is in bu is compatible with Test.

However, the following code compiles ok but still throws an exception:

```
BusinessUnit bu = new Board();
Test t = (Test)bu; // Runtime exception
```

As always, the compiler cannot run through all the code and try every possible combination of inputs to methods and events etc. and only complain if it finds a problem. (That is what testing is for!) So, the compiler gives an error if it is definite or likely that there is a problem.

Virtual Methods

A virtual method is where a method is called on an instance of a superclass but the implementation that is called depends on which subclass the object belongs to.

Let's say we want to calculate the ongoing annual cost of a list of organisational assets. The hierarchy looks like this:

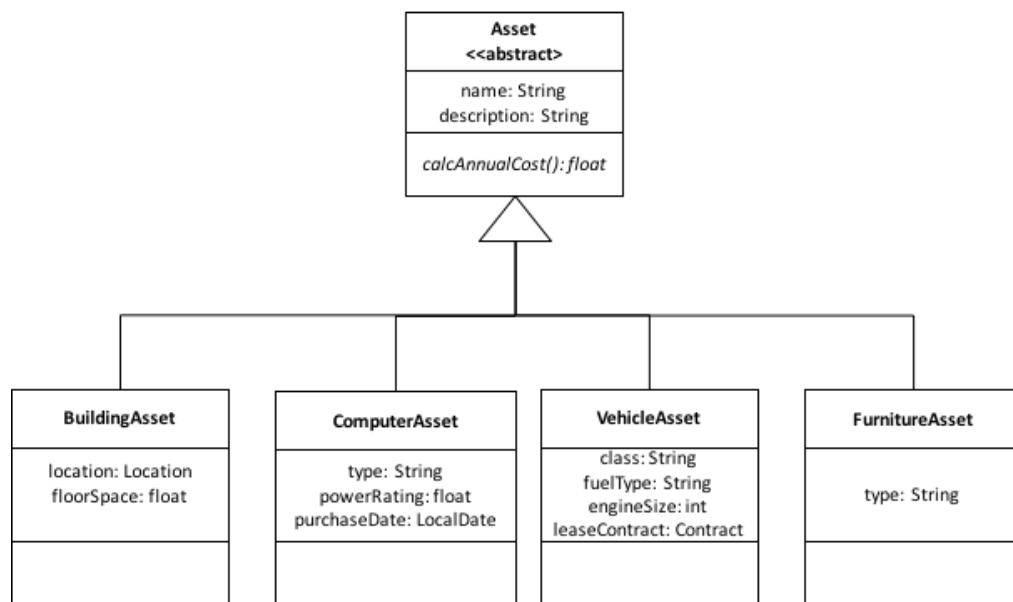


Figure 1-5 Asset Inheritance Hierarchy

Each concrete asset class inherits the `calcAnnualCost()` which returns a float. However, the calculations are very different and so this method is overridden in each subclass with the details of these differences.

Here is some code illustrating the Asset inheritance hierarchy shown above:

```

package finance;
import java.time.LocalDate;
public abstract class Asset {
    private String name;
    private String description;
    public Asset(String name, String description) {
        this.name = name;
        this.description = description;
    }
    public abstract float calcAnnualCost();
}

package finance;
public class BuildingAsset extends Asset {
    private Location location;
    private float floorSpace;
    public BuildingAsset(String name, String description) {
        super(name, description);
    }
}
  
```

```

        }
    public float calcAnnualCost() {
        System.out.println("BuildingAsset calcAnnualCost()");
        return 1.0F;
    }
}

package finance;
public class ComputerAsset extends Asset {
    private String type;
    private float powerRating;
    private LocalDate purchaseDate;
    public ComputerAsset(String name, String description) {
        super(String name, String description);
    }
    public float calcAnnualCost() {
        System.out.println("ComputerAsset calcAnnualCost()");
        return 2.0F;
    }
}

package finance;
public class VehicleAsset extends Asset {
    private String class;
    private String fuelType;
    private int engineSize;
    private Contract leaseContract;
    public VehicleAsset(String name, String description) {
        super(String name, String description);
    }
    public float calcAnnualCost() {
        System.out.println("VehicleAsset calcAnnualCost()");
        return 3.0F;
    }
}

package finance;
public class FurnitureAsset extends Asset {
    private String type;
    public FurnitureAsset(String name, String description) {
        super(String name, String description);
    }
    public float calcAnnualCost() {
        System.out.println("FurnitureAsset calcAnnualCost()");
        return 4.0F;
    }
}

```

Note that for simplicity the overridden methods simply print out the name of the class and method and return a fixed value: 1.0F for BuildingAsset, 2.0F for ComputerAsset and so on. The extra fields are not set or used in the calculation but you can see which method is called by the output.

Now we will have some code to calculate the costs:

```
package finance;
import java.util.ArrayList;
public class AssetTest {
    public static void main(String[] args) {
        ArrayList<Asset> assets = new ArrayList<>();
        assets.add(new BuildingAsset("HQ", "Headquarters"));
        assets.add(new ComputerAsset("Server1", "Local Storage Server"));
        assets.add(new VehicleAsset("JAG111", "CEO's Car"));
        assets.add(new FurnitureAsset("Table1", "Boardroom table"));
        float costs = 0;
        for(Asset asset : assets) {
            costs += asset.calcAnnualCost();
        }
        System.out.print("Cost for " + assets.size() + " assets: ");
        System.out.println(costs);
    }
}
```

This example only has one level of inheritance but the same logic applies with multiple levels.

The key point is that the variable called asset points to objects of different classes but we know they must be compatible with the Asset class and therefore must implement the calcAnnualCost() method. So, it is safe to call that method and the compiler allows it. This truly devolves the implementation to the subclass and anyone calling the method can guarantee its behaviour as far as the rules of overriding allow, i.e. what parameter types it will accept for input and what type it will return.

Variable Inheritance

Methods are inherited and may be overridden in subclasses and will be called as ‘virtual methods’ as described above. This is different for variables. Overriding a method is about changing its implementation. Since variables do not have an implementation it is hard to see why this is even a point for discussion but it is important not to get confused about how variable inheritance works.

Non-private variables may be inherited but may not be overridden. However, most classes are encapsulated which means the variables are private and cannot be inherited. If a variable is inherited and the subclass also contains a variable of the same name then this is called hiding.

Here is an example:

```
package demo;
public class SuperClass {
    public int number = 1;
}

package demo;
public class SubClass extends SuperClass {
    public int number = 2;
```

}

Ok, now let's create a couple of objects and see what happens:

```
package demo;
public class Test {
    public static void main(String[] args) {
        Super superClass = new SuperClass();
        Sub subclass = new SubClass();
        System.out.println("superClass number = " + superClass.number);
        System.out.println("subClass number = " + subclass.number);
        // That much is fairly predictable
        System.out.println("subClass super.number = "
                           + subclass.super.number);
        // You might even have guessed how that would work but...
        superClass = subclass; // OK because subclass IS-A SuperClass
        System.out.println("superClass number = " + superClass.number);
        // Now we get the superclass's number variable (1) printed!
    }
}
```

Ok, so what just happened? The number printed out depends on the variable's type not the object. This is not what happens with virtual methods and is hard to understand at first.

It is important to realise that the sub class has inherited a field called number with a value of 1. Then it has declared a second variable with the same name. The compiler

Since there really are two number variables, the compiler makes a decision as to which of them to print and it does so based on the variable class.

Just to reiterate, this is not good practice in any sense but it is important to understand as programmers often encounter less-than-perfect code in the work of others (and sometimes in the earlier work of their less-experienced selves!)

@Override Annotation

We have already seen how to override a method and discussed how this works. Java provides a way to indicate explicitly in the code that a method is being overridden. As you have seen it is very easy to make a mistake and accidentally fail to override properly. For example you might change a parameter type and accidentally overload the method. Other errors are available!

In Java, when you see code that begins with an @ symbol, it means that is an annotation. An annotation is extra information about the program, and it is a type of metadata. It can be used by the compiler or even at runtime.

The @Override annotation is used to annotate your code to indicate that you intend this method to override one in a superclass (or implement a method from an interface). You may not think of implementing an interface method as overriding, but it is.

Of course annotations are also very useful for anyone reading your code – including you when you come back many months in the future to make a change!

Here is an example:

```
package finance;
public class SoftwareAsset extends ComputerAsset {
    @Override
    public float calculateAnnualCost() { ... }
}
```

This example works fine. Remembering that white space is ignored by Java, you sometimes see the following layout:

```
package finance;
public class SoftwareAsset extends ComputerAsset {
    @Override public float calculateAnnualCost() { ... }
}
```

Both of the above will compile but the following will give a compiler error:

```
package finance;
public class SoftwareAsset extends ComputerAsset {
    @Override
    public float calculateAnnualCost(float depreciationPercentage) { ... }
}
```

This code overloads the method rather than overriding it so the compiler complains. This annotation can help avoid many mysterious bugs due to the superclass method running where you thought it should call the subclass overridden method.

It is legitimate to override a method from any superclass at any level above in the inheritance hierarchy.

Overriding only applies to methods so if you use @Override where there is no method immediately afterwards, the compiler will give an error.

Overriding the equals, hashCode, and toString methods from Object

All classes in Java inherit from `java.lang.Object`, either directly or indirectly, which means that all classes inherit any methods defined in `Object`.

Three of these methods are key to the behaviour of classes and it is therefore common for subclasses to override them with a custom implementation.

First, we will look at `toString()` which is the method called when you print out a class. Then we will see how `equals()` and `hashCode()` are used, particularly in collections. We will also discuss how `equals()` and `hashCode()` work together and how you can ensure consistency of behaviour by coordinating their overridden implementations.

`toString()`

You may remember when studying for the OCA, you learned that Java automatically calls the `toString()` method if you try to print out an object. Some classes supply a human-readable implementation of `toString()` but Java does only provides a basic default that is not very helpful for the human reader.

Here are some examples:

```
package demo;
public class MyClass {
    private String name;
    public MyClass(String name) {
        this.name = name;
    }
}

package demo;
public class Test {
    public static void main(String[] args) {
        MyClass myClass = new MyClass("Class1");
    }
}
```

In this case, the output is something like:

```
demo.MyClass@7852e922
```

Not very human-friendly!

On the other hand, many Java API classes, like `ArrayList`, have overridden `toString()` to produce something useful:

```
package demo;
import java.util.ArrayList;
public class TestArrayList {
```

```

public static void main(String[] args) {
    ArrayList<String> strings = new ArrayList<>();
    strings.add("Do");
    strings.add("Re");
    strings.add("Mi ");
    strings.add("Fa");
    strings.add("So");
    System.out.println(strings);
}
}

```

Now the output looks like this:

```
[Do, Re, Mi , Fa, So]
```

So, the MyClass class above could be improved to show the name variable when printed:

```

package demo;
public class MyClass {
    private String name;
    public MyClass(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return name;
    }
}

```

When the Test class is run again, this time it produces:

```
Class1
```

equals()

You know that Java uses `==` to compare primitives and for checking if two variables refer to the same object. That is not the same as checking if they are the same. It is perfectly possible to have two String objects with identical contents but the `==` test will not show this.

Checking if two objects are equivalent is done using the `equals()` method, but this will only produce a sensible result if the programmer overrides `equals()` in a meaningful way.

`String` does have an overridden `equals()` method. It checks that the values are the same.

However, `StringBuilder` does not override the `equals()` method provided by `Object`. It simply checks if the two objects being referred to are the same, i.e. it does the same as `==`.

There is a bit more to writing your own `equals()` method than there was to overriding `toString()`. For example, what data type does `equals()` accept as the argument?

Well, you would think that if you're comparing two objects of the same type (like `Employee`) then you're going to want to pass an object of that type into the method...

But hold on! We're overriding the equals() method so it has to have the same argument types and number, doesn't it? In the Object class, where the equals() method is first defined, it takes an argument of type Object. So that means that your overridden method also has to accept Object. This means that you might not get an object of the right class.

Let's start with a simple example:

```
package hr;
import java.time.LocalDate;
public class Employee {
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;
    private JobRole jobRole;
    public Employee(String firstName, String lastName,
                    LocalDate dateOfBirth) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }
}
```

Ok, the first thing to ask here is “what makes two Employee objects the same?”. After all that is what the equals() method does but it needs to make sense in the context of the application.

For example, we could look at the firstName and lastName fields to see if they are the same. But in a large organisation there may be people with the same name. What about if we include dateOfBirth? That will probably do it. What about the JobRole? Well, in this application we might want to look at an employee in multiple roles, perhaps over time. So the business rule is that two Employee objects are the same irrespective of JobRole.

In real life there would probably be an employee ID of some sort which would make comparison a lot easier but not in this example.

Our first attempt at overriding equals() might look like this:

```
public boolean equals(Employee employee) {
    return (this.firstName.equals(employee.firstName) &&
            this.lastName.equals(employee.lastName) &&
            this.dateOfBirth.equals(employee.dateOfBirth));
```

Unfortunately, this works fine as seen here:

```
package demo;
import java.time.LocalDate;
public class TestEmployee {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Al i", "Klein",
            LocalDate.of(1967, 7, 11));
        System.out.println("employee1: " + employee1);
        Employee employee2 = new Employee("Al i n", "Klein",
            LocalDate.of(1967, 7, 11));
        System.out.println("employee2: " + employee2);
        Employee employee3 = new Employee("Al i", "Klein",
            LocalDate.of(1967, 7, 11));
        System.out.println("employee3: " + employee3);
        System.out.println("employee1.equals(employee2): " +
            employee1.equals(employee2));
        System.out.println("employee1.equals(employee3): " +
            employee1.equals(employee3));
    }
}
```

This compares Employee objects when the caller supplies an Employee object as the argument. The output is:

```
employee1: [Al i, Klein, 1967-07-11]
employee2: [Al i n, Klein, 1967-07-11]
employee3: [Al i, Klein, 1967-07-11]
employee1.equals(employee2): false
employee1.equals(employee3): true
```

You may have noticed that we are able to print out the employee objects and so there must be an overridden `toString()` method in the Employee class. It is shown here:

```
public String toString(){
    return "[" + firstName + ", " + lastName + ", "
        + dateOfBirth + "]";
}
```

The (at present unseen) problem here arises because this is not an override but rather an overload of the `equals` method.

Let's first see why this is a problem.

Here is some more code that adds employee1 and employee2 to an ArrayList:

```

package demo;
import java.time.LocalDate;
import java.util.ArrayList;
public class TestEmployee {
    public static void main(String[] args) {
        Employee employee1 = new Employee("Al i", "Klein",
                                         LocalDate.of(1967, 7, 11));
        System.out.println("employee1: " + employee1);
        Employee employee2 = new Employee("Al i n", "Klein",
                                         LocalDate.of(1967, 7, 11));
        System.out.println("employee2: " + employee2);
        Employee employee3 = new Employee("Al i", "Klein",
                                         LocalDate.of(1967, 7, 11));
        System.out.println("employee3: " + employee3);
        System.out.println("employee1.equals(employee2): " +
                           employee1.equals(employee2));
        System.out.println("employee1.equals(employee3): " +
                           employee1.equals(employee3));
        ArrayList<Employee> employees = new ArrayList<>();
        employees.add(employee1);
        employees.add(employee2);
        System.out.println("employees.contains(employee2): " +
                           employees.contains(employee2));
        System.out.println("employees.contains(employee3): " +
                           employees.contains(employee3));
    }
}

```

So, you might expect that both calls to the ArrayList's contains() method would return true as employee1 and employee2 are equal according to our definition, which is contained in the logic of our equals() method.

In fact the output is:

```

employee1: [Al i , Klein, 1967-07-11]
employee2: [Al i n, Klein, 1967-07-11]
employee3: [Al i , Klein, 1967-07-11]
employee1.equals(employee2): false
employee1.equals(employee3): true
employees.contains(employee2): true
employees.contains(employee3): false

```

The problem here is that the ArrayList's contains() method uses the equals() method of the class it is searching for but it uses the one that takes an Object parameter because all objects have one of these, including our Employee class. Object's equals() method's behaviour is the same as the == operator. We need to override equals() properly.

So, the first thing to do is to change the signature to:

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

But of course, the Object class does not have the fields we are using to test equality which are firstName, lastName and dateOfBirth.

That means we are going to have to down cast the parameter as an Employee to avoid a compiler error:

```
Employee employee = (Employee) object;
```

But there is a potential danger here. If a non-Employee object is passed in as a parameter, there will be a runtime exception (a ClassCastException) thrown by the JVM.

We can put a test in to prevent this using the instanceof operator:

```
if (!(object instanceof Employee)) {
    return false;
}
```

It would also be possible to pass in a null to the equals() method but the above test checks for that too as null can never pass the instanceof test.

So, the Employee class now looks like this (note the @Override annotations):

```
package demo;
import java.time.LocalDate;
public class Employee {
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;
    private JobRole jobRole;
    public Employee(String firstName, String lastName,
                   LocalDate dateOfBirth) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }
    @Override
    public String toString() {
        return "[" + firstName + ", " + lastName + ", "
               + dateOfBirth + "]";
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Employee)) {
            return false;
        }
        Employee employee = (Employee) object;
        return (this.firstName.equals(employee.firstName)
               && this.lastName.equals(employee.lastName)
               && this.dateOfBirth.equals(employee.dateOfBirth));
    }
}
```

It is also possible to test for self-comparison (`object == this`) and return true. This might give a performance boost. It has not been included in the code here but you may see it.

And the result when the TestEmployee class is run again is:

```
empl oyee1: [Al i , Ki ei n, 1967-07-11]
empl oyee2: [Al i n, Ki ei n, 1967-07-11]
empl oyee3: [Al i , Ki ei n, 1967-07-11]
empl oyee1.equals(empl oyee2): false
empl oyee1.equals(empl oyee3): true
empl oyees.contains(empl oyee2): true
empl oyees.contains(empl oyee3): true
```

NOTE

There is a contract defined in the Java standard for overriding the equals() method.

Java provides the following rules in the contract for correctly overriding the equals() method:

- **It is reflexive:** For any non-null reference value x , $x.equals(x)$ should return true
- **It is symmetric:** For any non-null reference values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true
- **It is transitive:** For any non-null reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true
- **It is consistent:** For any non-null reference values x and y , multiple invocations of $x.equals(y)$ consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified
- **For any non-null reference value x , $x.equals(null)$ should return false**

In the OCP Java Programmer II exam, you should be able to recognise correct and incorrect equals() methods. You will not be expected to specify which rule is being violated though.

The code for Employee shown above obeys all of these rules.

hashCode()

The two methods hashCode() and equals() are closely linked. In fact, you can think of hashCode() as just a weaker form of equals(). It is used in Java API collections that (usually) have 'hash' in the name. For example, HashMap (see later chapter on collections) has entries where a key is used to find the data. The keys are hashed to make the HashMap more efficient.

The upshot is that whenever you override equals(), you must also override hashCode() if you want it to work correctly with Java API classes that rely on hashing..

A hash code is a special number that puts instances of a class into a finite number of categories and the hashCode() method returns an int to represent this code.

Imagine that you have a pile of employee records, and know that you will need to find the right record as quickly as possible when it is needed.

If you do nothing you will have to search through the whole pile when you need a file and that could be very slow. You could sort them in alphabetical order (of lastName) and stick something (a file divider maybe) in to indicate where each letter starts.

Effectively you will have put them into 26 piles, assuming you have at least one employee whose name starts with each letter of the alphabet.

In this case, the significant field is lastName and we only need the first letter. The code could look like this:

```
@Override  
public int hashCode(){  
    return (int)(lastName.toLowerCase().charAt(0));  
}
```

This simply returns the integer value of the first character (in lower case) of the lastName field. Without the call to toLowerCase() there could be 52 buckets.

In the case of employee Ali Klein, the result is:

```
employee1: [Ali, Klein, 1967-07-11]  
...  
employee1.hashCode(): 107
```

107 is the decimal equivalent of character 'k'.

Using this system there will always be 26 piles of employee files. These are very often described as 'buckets' when discussing hashing algorithms and the implementation of the hashCode() method.

There is a trade-off here between the number of potential buckets and the number of items in each bucket.

It is generally accepted that the more buckets you have the better for the speed of operation of data structures like HashMap that rely on hashing algorithms. This is not easily provable without a good understanding of how your Class will be used, what the distribution of data will be and some advanced mathematics.

The generally accepted strategy is to maximise buckets and return a hash code that is almost unique.

If there is a unique ID then this can be returned. If it is not an int then it may need to be converted. This field should also be the sole basis for equals() as well.

If there is no unique ID then you can use the significant fields of the object to calculate a hash code. The significant fields are the ones used in the equals() method.

The mechanism for this is to convert each significant field into an int and then combine these into a single int which is returned as the hash code.

Here is a table summarising how to do convert to an int:

Field Type	Formula to convert to int
boolean	(field ? 1 : 0)
byte, char or short	(int)field
long	(int)(field ^ (field >>> 32))
float	Float.floatToIntBits(field)
double	double d = Double.doubleToLongBits(field) ...then... (int)(d ^ (d >>> 32))
Object reference	Use the object's hashCode() method

To combine the ints together, create an int variable and initialise to a constant non-zero value. Then successively multiply it by a prime number and add the int value of the field from the table above. 31 is often used as the prime number as the compiler can easily optimise its multiplication to $((x \ll 5) - x)$ which is faster.

Here is some code to do this for the Employee class. The equals() method is also shown here:

```

@Override
public boolean equals(Object obj) {
    if (! (obj instanceof Employee)) {
        return false;
    }
    Employee employee = (Employee) obj;
    return (this.firstName.equals(employee.firstName) &&
            this.lastName.equals(employee.lastName) &&
            this.dateOfBirth.equals(employee.dateOfBirth));
}

@Override
public int hashCode() {
    int hashCode = 5;
    hashCode = 31 * hashCode + firstName.hashCode();
    hashCode = 31 * hashCode + lastName.hashCode();
    hashCode = 31 * hashCode + dateOfBirth.hashCode();
    return hashCode;
}

```

The equals() and hashCode() methods use exactly the same fields to determine their respective return values. We could have used a subset of the fields used for equals() but this would decrease the number of buckets. This is important for the Java specification hashCode() contract. For example, using fields in hashCode() that have not been used in equals() may break the contract.

NOTE

There is a contract defined in the Java standard for overriding the hashCode() method.

Java provides the following rules in the contract for correctly overriding the hashCode() method:

- **It must be consistent:** For any non-null reference value x, multiple invocations of x.hashCode() consistently returns the same value.
- For two non-null reference values x and y, if x.equals(y) returns true then x.hashCode() == y.hashCode() must be true
- For two non-null reference values x and y, if x.equals(y) returns false then x.hashCode() == y.hashCode() may be true or false

For the OCP Java Programmer II exam, you should be able to recognise hashCode() implementations that are inconsistent with the equals() method for the same class.

One final comment here is to say that for immutable objects where the hash code will be frequently evaluated, there may be a benefit in caching it in an instance variable so it is only evaluated once. Here is a revised Employee class showing this functionality:

```
package demo;
import java.time.LocalDate;
public class Employee {
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;
    private JobRole jobRole;
    private final int hashCode;
    public Employee(String firstName, String lastName,
                    LocalDate dateOfBirth) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
        int temp = 5;
        temp = 31 * temp + firstName.hashCode();
        temp = 31 * temp + lastName.hashCode();
        temp = 31 * temp + dateOfBirth.hashCode();
        hashCode = temp;
    }
    ...
    @Override
    public int hashCode(){
        return hashCode;
    }
}
```

As you can see, the variable hashCode is now an instance variable that is initialised in the constructor and never changes. It is declared as final to emphasise this point.

Enums

It is a common requirement to have a type that can only have a finite set of values. An enumeration or enum is like a fixed set of constants.

In Java, an enum is a class that represents this concept. It is much better than declaring several constants because it provides type-safe checking as well.

With numeric constants, you could potentially provide an invalid value and not find out until runtime. With enums, it is impossible to create an invalid enum type without getting a compiler error.

Enumerations show up whenever you have a set of items whose types are known at compile time. Common examples are categories or lists that change very infrequently. If new items are added or any changes made then the enum has to be recompiled.

One example is a list of job role categories for dealing with employees.

These are:

1. Director
2. Manager
3. Supervisor
4. Technician
5. Administrator

Declaring an enum is similar to declaring a class, just use the enum keyword instead of the class keyword. Then list all of the valid types for that enum.

```
public enum JobCategory {
    DIRECTOR, MANAGER, SUPERVISOR, TECHNICIAN, ADMINISTRATOR
}
```

As an enum is like a set of constants, it is conventional to use uppercase letters that are normally used for constants.

Behind the scenes, an enum is a type of class that mainly contains static members. It also includes some helper methods like name() that you will see shortly. Using an enum is easy:

```
JobCategory jc = JobCategory.TECHNICIAN;
System.out.println(jc);
System.out.println(jc == JobCategory.TECHNICIAN);
```

This prints out:

```
JobCategory.TECHNICIAN
true
```

As you can see, enums print the name of the enum when `toString()` is called. They are also comparable using `==` because they are like static final constants.

An enum has a method called `values()` that returns an array of all of the values. You can use this like any normal array, including in an iteration loop:

```
for(JobCategory jobCat : JobCategory.values()) {  
    System.out.println(jobCat.name() + " " + jobCat.ordinal());  
}
```

The output is:

```
DIRECTOR 0  
MANAGER 1  
SUPERVISOR 2  
TECHNICIAN 3  
ADMINISTRATOR 4
```

You can't compare an int and enum value for equality because an enum is a type and not an int.

```
if (JobCategory.DIRECTOR == 0) { ... } // DOES NOT COMPILE
```

You can also create an enum from a String. This is helpful when working with String parameters for example. However, the String passed in must match exactly so you may have to use toUpperCase().

```
String string1 = "DIRECTOR";  
String string2 = "Manager";  
JobCategory jobCategory1 = JobCategory.valueOf(string1);  
JobCategory jobCategory2 = JobCategory.valueOf(string2); // Exception
```

The first valueOf() statement works and assigns the proper enum value to jobCategory1. The second statement encounters a problem. There is no enum value with the literal string "Manager" so the JVM throws an IllegalArgumentException.

```
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant enums.JobCategory.Manager
```

You cannot extend an enum as you can a class or interface.

```
public enum ExtendedJobCategory extends JobCategory { ... }  
// DOES NOT COMPILE
```

The values declared in an enum are all that are allowed. You cannot add more at runtime by extending the enum.

Now that we've covered the basics, we look at using enums in switch statements and how to add extra functionality to enums.

Enums in switch statements

Enums are like constants and as such may be used in switch statements. Pay attention to the case values in this code as they just have the constant and not the enum name:

```
JobCategory jobCategory = JobCategory.TECHNI CI AN;
switch(jobCategory) {
    case DIRECTOR:
        // Code for DIRECTOR
        break;
    case MANAGER:
        // Code for MANAGER
        break;
    case SUPERVISOR:
        // Code for SUPERVISOR
        break;
    case TECHNICIAN:
        // Code for TECHNICIAN
        break;
    case ADMINISTRATOR:
        // Code for ADMINISTRATOR
        break;
    default:
        // Default code
}
```

You cannot use the enum name as a prefix in a case value, nor use the ordinal number so neither of the following would compile as part of the example code above:

```
case JobCategory.ADMINISTRATOR:
case 4:
```

Enum Constructors

Enums can also have constructors, which means you can add values to the constant to represent anything you like about it.

For example, let's say our JobCategory system has grades that are used to calculate the pay scale. The grades are integer values between 1 and 5:

```
public enum JobCategory {
    DIRECTOR(5), MANAGER(4), SUPERVISOR(3), TECHNICIAN(2),
    ADMINISTRATOR(1);

    private int grade;
    private JobCategory(int grade){
        this.grade = grade;
    }
    public int getGrade(){
        return grade;
    }
}
```

Now, each enum value is followed by an int in brackets and there is a semicolon after the list of values. The semicolon is required if there is anything else apart from the values and may be present even if there is nothing else.

Apart from that, there is now a variable called grade and a constructor that assigns its parameter to the variable.

NOTE

Enum constructors must be private or the code will not compile.

The constructor gets called once for each value when any of the values is first used. There would be no point in the constructor running more than once since the input values cannot change.

The example above has a getter method to provide visibility to the grade variable.

It is possible to have other methods just like a regular class.

It is also possible to have the methods behave differently for different values just like subclasses overriding a superclass method:

```
public enum JobCategory {  
    DIRECTOR(5) { public int getMaxScalePoints() { return 12; }  
                 public int getMinScalePoints() { return 3; } },  
    MANAGER(4),  
    SUPERVISOR(3),  
    TECHNICIAN(2) { public int getMaxScalePoints() { return 15; }  
                  public int getMinScalePoints() { return 2; } },  
    ADMINISTRATOR(1);  
    private int grade;  
    private JobCategory(int grade){  
        this.grade = grade;  
    }  
    public int getGrade(){  
        return grade;  
    }  
    public int getMaxScalePoints() { return 8; }  
    public int getMinScalePoints() { return 1; }  
}
```

In this case, all classes inherit getMaxScalePoints() and getMinScalePoints() which return 8 and 1 respectively but they are overridden by DIRECTOR and TECHNICIAN which have different rules for calculating pay scales.

Methods belonging to the superclass (JobCategory) may also be abstract in which case they must be overridden.

Nested Classes

A nested class is a class that is declared within another class.

An inner class is a nested class that is not static. There are four types of nested classes:

- A member inner class is a class defined at the same level as instance variables and non-static methods and is usually just called an inner class
- A local inner class is defined within a method like a local variable
- An anonymous inner class is a special case of a local inner class that does not have a name and may be assigned to a variable or passed to a method
- A static nested class is a static class that is defined at the same level as static variables and is attached to the outer class as a matter of convenience, very much like static methods and constants

There are a few reasons for using inner classes. They can encapsulate helper classes by restricting them to the containing class. They can make it easy to create a class that will be used in only one place. They can make the code easier to read. Of course, they can also make the code harder to read when used improperly.

Member Inner Classes

A member inner class (as the name suggests) is defined at the member level of a class. This is the same level as the methods, instance variables, and constructors, i.e. within the code block defined by the class itself.

Member inner classes:

- can be declared public, private, or protected or use default access
- can extend any class and implement interfaces
- can be abstract or final
- cannot declare static fields or methods
- can access members of the outer class including private members

The last point means that the inner class can access the outer class as if it were a method or other code block within the outer class.

Here is an example:

```
package demo;
public class Outer {
    private int repeat = 5;
    protected class Inner {
        public void go() {
            for (int i = 0; i < repeat; i++){
                System.out.println("Inner.go()");
            }
        }
    }
    public void callInner() {
```

```
    Inner inner = new Inner();
    inner.go();
}
public static void main(String[] args) {
    Outer outer = new Outer();
    outer.callInner();
}
}
```

The inner class code can access private variables, like `repeat` in this case, because it is doing so from within the outer class, just like an outer class method.

Like other non-static members of the outer class, the inner class can only be used by creating an instance.

There are two ways this could happen. One way is for some code to create an instance of the outer class and then use it (usually via a reference variable) to create an instance of the inner class. The other way is to write code within the outer class, perhaps in a method, that creates an instance of the inner class. That code can only work when an instance of the outer class is created so the usual rules apply that you can't create an instance of a non-static inner class from a static method of the outer class.

The syntax for creating an inner class using a reference to an instance of the outer class is pretty strange:

```
Outer outer = new Outer();
Inner inner = outer.new Inner();
```

This strange syntax with the `new` keyword attached to the reference variable as if it were some kind of member of the object being referenced is only used in this situation. It is because the inner class instance is attached to the instance of the outer class of which it is a member.

NOTE

Compiling the Outer.java class with which we have been working creates two class files. Outer.class (as usual but for the inner class, the compiler creates Outer\$Inner.class).

This shows that the inner class is a real class and will be loaded along with all the other class files by the JVM if used in your code.

Inner classes can have the same variable names as outer classes and so there is a special way of specifying which class's variable you want to access.

It is also possible (but not advisable) to nest multiple classes and access a variable with the same name in each:

```
public class A {
    private int x = 10;
    class B {
        private int x = 20;
        class C {
            private int x = 30;
            public void printAll() {
                System.out.println(x);
                System.out.println(this.x);
                System.out.println(B.this.x);
                System.out.println(A.this.x);
            }
        }
    }
    public static void main(String[] args) {
        A a = new A();
        A.B b = a.new B();
        A.B.C c = b.new C();
        c.printAll();
    }
}
```

Local Inner Classes

A local inner class is a nested class defined and used within a method. Like local variables, local inner class declarations do not exist until the method is invoked, and they go out of scope when the method returns.

This means that you can create instances only from within the method. Those instances can still be returned from the method. This is exactly the same as how local variables work.

Local inner classes have the following properties:

- They do not have an access modifier (private, public etc.)
- They cannot be declared static and cannot declare static fields or methods
- They have access to all fields and methods of the enclosing class
- They do not have access to local variables of a method unless those variables are final or effectively final (a new concept in Java 8)

Here is a simple code example:

```
public class Outer {
    private int length = 7;
    public void calculate() {
        final int width = 10;
        class LocalInner {
            public void multiplyMethod() {
                System.out.println(length * width);
            }
        }
        LocalInner localInner = new LocalInner();
        localInner.multiplyMethod();
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.calculate();
    }
}
```

Local variable references are only allowed if they are final or effectively final. As stated above, this is a new concept in Java 8.

So, why is this important? The compiler is generating a class file from your inner class. A separate class has no way to refer to local variables. If the local variable is final, Java can handle it by passing it to the constructor of the inner class or by storing it in the class file. This creates a copy of the variable with no link back to the original local variable.

If it weren't final, the value could change after the copy was made which could lead to inconsistent behaviour or just confusing code.

Until Java 7, the programmer had to type the final keyword. In Java 8, the 'effectively final' concept was introduced. If the code still compiles with the keyword final inserted before the local variable, the variable is effectively final.

Remember that final variables can be declared and initialised on different lines. Final means only set once and never changed.

NOTE

Local variables need to be effectively final to be accessed from within a local inner class contained within the same method.

Method parameters are also only accessible if effectively final as they become local variables in the method.

Although it is unusual to change the value of method parameters within the method, it is legal and renders them not effectively final and therefore not accessible within a local inner class.

Method parameters may be marked with the final keyword, although this is also quite unusual.

Here is some sample code to illustrate the concept of variables being effectively final:

```
public void testFinal (int x, int y) {
    int i = 5;
    int j = i;
    j++;
    int k;
    if(i == 7){
        k = 11;
    }else{
        k = 13;
    }
    int m = 17;
    class LocalInner{ }
    m = 19;
    x = y;
}
```

First of all, there are six variables that are local to the testFinal() method: x, y, i, j, k and m. Don't forget the method parameters.

Of these none is marked final so we have to determine if they are effectively final. The compiler also has to do this so you could just compile it unless you are in the OCP Java Programming II exam at the time of course...

x and y are passed in as parameters to the argument so they don't need to be initialised. You need to check if they are changed and x is reassigned the value of y on the very last line. So, x is not effectively final but y is ok.

i is ok. It is declared and initialised in-line and then never changed although it is used (read) a few times. j is also declared and initialised in-line but then it is changed on the next line.

k is also fine. Even though there are two lines making the assignment they are mutually exclusive so only one can possibly run and so it will be initialised ok.

m is not ok as it is modified after being initialised.

Of course, it is not an error to have variables that are not effectively final in a method that declares an inner class. It is only an error if you try to access one of these variables from within the inner class.

Since the above code declares an empty class, it would compile without error.

Another point to note here is that since you can hide variables by redefining them, you could have another local or instance variable in the inner class with the same name as a local variable in the method declaring the inner class. In this case you are dealing with the inner class's variable and it is irrelevant whether the outer method's variable is effectively final or not.

Local variables, whether effectively final or not, need to be initialised and you will get a compiler error to this effect if they are not.

It is irrelevant whether the variable declaration takes place before or after the class declaration. However, the class cannot be instantiated before it is declared.

Anonymous Inner Classes

An anonymous inner class is a local inner class that does not have a class name although it may be assigned to a variable.

It is declared and instantiated all in one statement using the new keyword.

Anonymous inner classes must extend an existing class or implement an existing interface and so may be assigned to a variable of that type.

They are useful when you have a short implementation that will not be used anywhere else.

Here is an example:

```
public class TestAnonymous {
    abstract class Super {
        abstract int superMethod();
    }
    public int calc(int x) {
        Super anon = new Super() {
            int superMethod() {
                return 3;
            }
        };
        return x - anon.superMethod();
    }
}
```

After the anonymous inner class has been defined there is a semicolon after the closing curly bracket. This is because this is the end of a statement in which a variable has been declared and assigned.

Anonymous inner classes may also be passed as the parameter to a method call and not even assigned to a variable.

Lambda expressions are implemented as anonymous inner classes and so you will see a lot more examples of inner classes in Java 8 than in previous versions.

A point to note here is that the code above appears to instantiate an abstract class, which of course is impossible. What is actually happening is that an abstract class is being subclassed and it is the subclass that is being instantiated. The new keyword is slightly confusing here. The same would apply if it were an interface.

Static Nested Classes

The final type of nested class is not an inner class.

A static nested class is a static class defined at the member level. It can be instantiated without an object of the enclosing class, so it can't access the instance variables without an explicit object of the enclosing class.

For example, new OuterClass().var allows access to the instance variable var.

In other words, it is like a regular class except for the following:

- The nesting creates a namespace because the enclosing class name must be used to refer to it just like static variables and methods
- It can be made private or use one of the other access modifiers to encapsulate it so its access may be restricted just like static variables and methods
- The enclosing class can access the private fields and methods of the static nested class by using the nested class name for statics and an instance of the nested class for non-statics

```
package demo;
public class StaticOuter {
    public static class StaticNested {
        private int value = 6;
        private static int number = 13;
    }
    public static void main(String[] args) {
        StaticNested nested = new StaticNested();
        System.out.println(nested.value);
        System.out.println(StaticNested.number);
    }
    public void testMethod(){
        System.out.println(StaticNested.number);
        System.out.println(new StaticNested().value);
    }
}
```

From a different class, the instantiation looks like this:

```
package demo;
public class TestStaticNested {
    public static void main(String[] args){
        StaticOuter.StaticNested nested = new StaticOuter.StaticNested();
    }
}
```

And, as you might expect, from here there is no access to either of the private variables: number and value.

From a different package, you will need to import of course:

```
package other;
import demo.StaticOuter.StaticNested;
public class TestImportStaticNestedClass {
    StaticNested nested;
}
```

Because `StaticNested` is a static member of `StaticOuter`, you could use a static import with the same effect:

```
package other;  
import static demo.StaticOuter.StaticNested;  
...
```

or:

```
import static demo.StaticOuter.*
```

Summary

The instanceof operator compares an object to a class or interface type and returns true or false. It also looks at subclasses and subinterfaces. x instanceof Object always returns true unless x is null. If the compiler can determine that there is no way for instanceof to return true, it will generate a compiler error.

Virtual method invocation means that Java will look at subclasses when finding the right method to call. This is true, even from within a method in the superclass. There is no such thing as a virtual variable though.

The methods `toString()`, `equals()`, and `hashCode()` are implemented in `Object` and so all classes can override them to change their behaviour. `toString()` is used to provide a human-readable representation of the object. `equals()` is used to specify which significant instance variables should be considered for equality. `equals()` is required to return false when the object passed in is null or is of the wrong type. `hashCode()` is used to provide a grouping in some collections. `hashCode()` is required to return the same number when called with objects that are the same according to `equals()`.

The enum keyword is short for enumerated values or a list of values. Enums can be used in switch statements. In a switch statement, the enum value is placed in the case. Enums are allowed to have instance variables, constructors, and methods. Enums can also have value-specific methods. The enum itself declares that method as well. It can be abstract, in which case all enum values must provide an implementation. Alternatively, it can be concrete, in which case enum values can choose whether they want to override the default implementation.

There are four types of nested classes. Member inner classes require an instance of the outer class to use. They can access private members of that outer class.

Local inner classes are classes defined within a method. They can also access private members of the outer class. Local inner classes can also access final or effectively final local variables.

Anonymous inner classes are a special type of local inner class that does not have a name. Anonymous inner classes are required to extend exactly one class by name or implement exactly one interface.

Static nested classes can exist without an instance of the outer class and behave just like any other static member of the outer class.

This chapter also contained a review of access modifiers, overloading, overriding, abstract classes, static, final, and imports. It also introduced the optional `@Override` annotation for overridden methods or methods implemented from an interface.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 2

Java Design Patterns

Introduction

What do we mean when we say we want to write good code? Is it possible to differentiate good code from bad code and if so, can you quantify the difference?

Previously you may have focused on developing Java code that compiles and executes properly at runtime but this chapter is about finding best practices for designing Java classes and writing applications that lead to code that is easier to understand, more maintainable, and that you and other developers can use effectively in the future.

Conforming to design principles and using design patterns enables you to create potentially complex class models that interact properly with other developers' code. The better your software is designed, the better it may adapt to changes in requirements, over the course of the project lifespan and beyond. Many of the established Java libraries that you rely on to build your own applications may have started life as simple projects aiming to solve a commonly reoccurring problem.

This chapter is about teaching you powerful techniques for writing software so that you can build on what others have learned while avoiding the mistakes and pitfalls that have been identified and documented by others.

Designing Interfaces

An interface is an abstract data type, similar to a class that defines a list of public abstract methods that any class implementing the interface must provide.

Since any class implementing the interface must implement its methods, there is an IS-A relationship between class and interface. This means that a variable may be declared of the interface's type and used to reference an instance of the class.

Typically, interfaces have been used as a way of separating method signatures from implementation code.

However, since Java 8, interfaces can now declare methods with implementation code thanks to two improvements. First there are default methods, which let you declare methods with implementation code inside an interface. This provides a mechanism to evolve the Java API (and your class libraries) in a backward-compatible way. For example, you'll see that the List interface now supports a sort method that is defined as follows:

```
default void sort(Comparator<? super E> c) {  
    Collections.sort(this, c);  
}
```

Default methods can also serve as a multiple inheritance mechanism for behaviour. Classes can implement multiple interfaces and therefore inherit the signature of methods from multiple hierarchies. Java 8 allows classes to inherit implementation code from multiple hierarchies as well. Note that Java 8 has explicit rules to prevent inheritance issues common in C++ (where multiple inheritance is allowed) such as the diamond problem.

Second, interfaces can now also have static methods. The Java API often defines both an interface and a companion utility class defining static methods for working with instances of the interface. For example, Java has the Collection interface and the Collections class.

These static methods can now live within the interface. For instance, the Stream interface in Java 8 declares a static method like this:

```
public static <T> Stream<T> of(T... values) {  
    return Arrays.stream(values);  
}
```

An interface may also include constant public static final variables. These are defined in the usual way and are accessible from classes that implement the interface. This approach is not recommended and is often described as an anti-pattern.

An interface may extend another interface, and in doing so it inherits all of its abstract methods. Interfaces and classes may not extend each other and interfaces may not implement other interfaces.

Interfaces may also not be marked final or instantiated directly. There are additional rules for default methods, such as Java failing to compile if a class or interface inherits two default methods with the same signature and doesn't provide its own implementation.

An interface provides a way for one programmer or team to develop code that uses another's specification, without having access to the underlying implementation.

Interfaces can facilitate rapid application development by enabling development teams to create applications in parallel, rather than being directly dependent on each other.

For example, two teams can work together to develop a one-page standard interface at the start of a project. One team then develops code that uses the interfaces while the other team develops code that implements the interface. The development teams can then combine their implementations toward the end of the project, and as long as both teams developed with the same interface, they will be compatible.

Of course, testing will still be required to make sure that the class implementing the interface behaves as expected.

How a developer using just an interface build some code without access to a class that implements the interface.

There is a technique that is commonly used in this situation:

The developer using the interface can create a temporary mock object, sometimes referred to as dummy code, which simulates the real object that implements the interface with a simple implementation. The mock object does not need to be very complex, with one line per abstract method, for example, as it only serves as a placeholder for the real implementation. This allows the developer using the interface to compile, run, and test their code.

There are libraries such as Mockito that will automatically produce a mock object based on the interface.

Functional Programming

Java defines a functional interface as an interface that contains a single abstract method.

A lambda expression provides implementation code for a single method.

Therefore, functional interfaces may be implemented completely by lambda expressions. A lambda expression may be stored in a variable and passed as a parameter to a method. Many Java API methods define their parameters using functional interfaces and where previously an anonymous class would be used as the parameter, from Java 8, a lambda expression may be used and this dramatically improved readability.

Functional Interfaces

Here is an example of a functional interface and a class that implements it:

```
package finance;
@FunctionalInterface
public interface Payable {
    public float calculatePayment();
}

package finance;
public class Supplier implements Payable {
    private float outstandingPayments;
    public Supplier(float outstandingPayments) {
        this.outstandingPayments = outstandingPayments;
    }
    public float calculatePayment() {
        return outstandingPayments;
    }
}
```

In this example, the Payable class is a functional interface, because it contains exactly one abstract method, and the Supplier class is a valid class that implements the interface and provides code for the method. The `@FunctionalInterface` annotation declares that there is one and only one abstract method and compilation will fail if this is not the case.

It is recommended that you explicitly mark the interface with the `@FunctionalInterface` annotation so that other developers know which interfaces they can safely apply lambdas to without the possibility that they may stop being functional interfaces later in development.

An interface may be functional be several means:

1. It may just have a single abstract method
2. It may have inherited a single abstract method and not added anything
3. It may additionally have a default method
4. It may additionally have a final static constant

Remember, an interface having no abstract methods or more than one is not a functional interface.

Using Lambdas with Functional Interfaces

As stated earlier, a lambda expression is a block of code that gets passed around, like an anonymous method. In fact, it is actually an anonymous subclass with a single method.

Here is an example of a utility class with a method that takes two parameters, the List interface and a functional interface that can be used to filter the list. Of course, you can't instantiate an interface so the caller must pass instance of an implementer of the interface. The utility class method then uses the interface's method to process the class.

The class in this example is going to be a list of Employee objects. The functional interface therefore needs to test something about the Employee objects and include or exclude them from the result.

Here is the Employee class:

```
package hr;
import java.time.LocalDate;
public class Employee {
    private String firstName;
    private String lastName;
    private LocalDate dateOfBirth;
    private float grade;
    private String department;
    public Employee(String firstName, String lastName,
                   LocalDate dateOfBirth, float grade, String department) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getDepartment() {
        return department;
    }
    public float getGrade() {
        return grade;
    }
    // Other variables and methods...
}
```

Here is the utility class:

```
package hr;
import java.util.List;
import java.util.ArrayList;
public class Utility {
    public static <T> List<T> filter(List<T> list, Test<T> test) {
        List<T> filtered = new ArrayList<>();
        for(T t : list){
            if(test.test(t)){
                filtered.add(t);
            }
        }
    }
}
```

```
        }  
    }  
    }  
    return filtered;
```

And here's some code to test its operation:

```
package hr;
import java.util.List;
import java.util.ArrayList;
import java.time.LocalDate;
public class TestUtility{
    public static void main(String[] args){
        ArrayList<Employee> employees = new ArrayList<>();
        employees.add(new Employee("Alice", "Stelle",
                LocalDate.of(1961, 7, 10), 5.0F, "IT"));
        employees.add(new Employee("Baz", "Tyco",
                LocalDate.of(1967, 1, 19), 2.5F, "Sales"));
        employees.add(new Employee("Cat", "Polli",
                LocalDate.of(1950, 12, 3), 5.2F, "Sales"));
        employees.add(new Employee("Dot", "Lisle",
                LocalDate.of(1958, 3, 9), 7.0F, "Finance"));
        employees.add(new Employee("Effy", "Khalt",
                LocalDate.of(1973, 9, 23), 3.0F, "Sales"));
        List<Employee> filtered = Utility.<Employee>filter(employees,
                e->e.getDepartment() == "Sales");
        for(Employee employee : filtered){
            System.out.println(employee);
        }
        filtered = Utility.<Employee>filter(employees,
                e->e.getGrade() > 5.0F);
        for(Employee employee : filtered){
            System.out.println(employee);
        }
    }
}
```

In this code the lambda expressions:

```
e->e.getDepartment() == "Sales"
```

and

e->e. getGrade() > 5. OF

each provide the code for the test() method. The first only accepts Employee objects from the Sales department. The second checks for employees whose grade is higher than 5.

Lambda Expression Syntax

The two lambda expressions shown above are very simple and use the minimum syntax.

If there is a single parameter and the data type is clear in the interface, as in the example, then the lambda starts with the parameter name and an arrow:

e->

If there are zero or more than one parameter or if the data types need to be specified then round brackets are required:

()->
(Employee e)->
(x, y)->

If any of the data types is specified then they all must be.

Our two lambda expressions are also simple in that all they do is return a value so the code (after the arrow) effectively consists of a single return statement. In this case the return keyword may be omitted and no curly brackets are required:

```
e.getDepartment() == "Sales"  
e.getGrade() > 5.0F
```

These two examples result in a boolean true or false and this is the return value for the lambda expression.

If there are multiple statements then curly brackets and a return statement (if the lambda returns a value) are required.

Predicate Functional Interface

The example given above is very common. Effectively you need to check if an object passes a test and return true or false. Java includes a number of useful functional interfaces in a package called `java.util.function`. One of these is called `Predicate` and provides the functionality we need. It takes one parameter and returns a boolean, just like our `Test` interface above. We could substitute `Predicate` for `Test` in the code above and, apart from an import statement, our code would continue to work identically.

There are many common situations (use cases) where a `Predicate` is required and so classes and methods in the Java API have been designed to use it.

There are many use cases where a functional interface with a different signature from `Predicate` is required.

A very common one is called `Function`. It accepts a parameter and returns a result, just like many mathematical functions. A common use case for `Function` is to transform an item. Perhaps messages are being received and need to be modified before being processed or re-transmitted.

`Predicate` and `Function` are extremely flexible.

Predicate can accept a parameter of any type and returns a boolean. Function can accept a parameter of any type and return a value of any type.

These functional interfaces are so flexible because they are based on generics.

The signature of Predicate is:

```
@FunctionalInterface  
public interface Predicate<T>
```

The <T> means you have to provide the data type (e.g. Employee) when using the interface.

The signature of Function is:

```
@FunctionalInterface  
public interface Function<T, R>
```

Here the <T> is the data type the function takes as an argument and <R> is the data type of the return value.

Predicate can be used wherever filtering is required and Function may be used to transform data. Both of these are widely used in the Java API and will be used in future sections of this course such as those focusing on collections and streams.

Polymorphism

Polymorphism is the ability of a single interface or superclass to support multiple underlying forms. The underlying forms are classes that implement interfaces or extend a class (or both).

In Java, this allows multiple types of objects to be passed as arguments to the same method. It also allows multiple types of object to be held in a single collection.

The method or collection treats the object as if it were of the type of the defined data type of the argument or collection element although it could be of any subclass. This means that it is only possible to call methods that are present in the super class.

This makes sense because any subclass or implementer of an interface must have these methods.

However the code that is executed is that contained within the class definition of the object.

This is where the term polymorphism comes from because it means that each object can behave differently and appropriately for its own class.

Here is an interface:

```
package finance;
public abstract interface Payable{
    public abstract void pay();
}
```

Here are two classes that implement the interface:

```
package finance;
public class Supplier implements Payable{
    private String name;
    private float balance;
    public Supplier(String name, float balance){
        this.name = name;
        this.balance = balance;
    }
    public void pay(){
        Payment payment = new Payment(name, balance);
        payment.makePayment();
    }
}
```

```

package hr;
import finance.Payable;
public class Contractor implements Payable{
    private String name;
    private float hours;
    private float rate;
    public Contractor(String name, float hours, float rate){
        this.name = name;
        this.hours = hours;
        this.rate = rate;
    }
    public void pay(){
        Payroll.getInstance().addPaymentLine(name, hours * rate);
    }
}

```

And here is some code that uses the `pay()` method in both classes:

```

package finance;
import java.util.ArrayList;
import hr.Contractor;
public class MakePayments{
    public static void main(String[] args){
        ArrayList<Payable> duePayments = new ArrayList<>();
        duePayments.add(new Supplier("Office Supplies", 545.20F));
        duePayments.add(new Contractor("Jan Smith", 27.0F, 18.0F));
        duePayments.add(new Contractor("CarService.com", 291.85F));
        for(Payable p : duePayments){
            p.pay();
        }
    }
}

```

Casting Object References

In the previous example, we created instances of `Contractor` and `Supplier` classes and accessed them via the `Payable` interface in the form of an `ArrayList` of type `Payable` and then an iterator for loop variable of type `Payable`.

By using the interface reference type you lose access to more specific methods defined in the subclass that would still be usable within the object. When the compiler sees the variable type it will only allow you to use methods that belong to that type irrespective of the actual object being referenced. This is a good policy because the only methods that can guarantee to run are those of the variable type.

If you want to use subclass methods you will need to cast the object to a variable of the appropriate type. This is called ‘down casting’ because the subclass is ‘below’ the superclass or interface in the inheritance hierarchy.

To down cast a `Payable` as a `Contractor` for instance, you could use the following code:

```

Payable p = new Contractor("Jan Smith", 27.0F, 18.0F);
Contractor c = (Contractor)p;

```

The explicit cast using (Contractor) before the variable being assigned, is required for down casts because the compiler does not know if this is a safe operation.

However you can still make a mistake and try casting an incompatible object being referenced by a Payable variable as a Contractor if your logic allows something else (maybe a Supplier) to be referenced by the Payable variable.

The following code will compile:

```
Payable p = new Supplier("Office Supplies", 545.20F);
Contractor c = (Contractor)p;
```

Supplier does not have an IS-A relationship with Contractor so this will fail at runtime and the JVM will throw a ClassCastException.

Of course, there are some casts that could never work such as:

```
String s = new String("Office Supplies");
Contractor c = (Contractor)s;
```

The compiler will not allow this as Contractor and String are totally unrelated.

You can always assign the contents of a subclass variable to a superclass:

```
Supplier s = new Supplier("Office Supplies", 545.20F);
Payable e = s;
```

This does not require an explicit cast and can never cause a ClassCastException exception to be thrown.

The rules are:

1. Casting an object from a subclass to a superclass doesn't require an explicit cast
2. Casting an object from a superclass to a subclass requires an explicit cast
3. The compiler will not allow casts to unrelated types
4. Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class

Java Design Principles

Java design principles are the best practices that have emerged from the experience of many developers over many years. The aim is to use this experience to facilitate the Java software design process.

In this section, we will discuss design principles for creating Java classes and why those principles lead to better and more manageable code.

Following good design principles leads to:

- More logical and concise code
- Code that is easier to read and understand
- Classes that are easier to reuse in other relationships and applications
- Code that is easier to maintain and that adapts more readily to changes in the application requirements
- Reduced total cost of ownership of software
- Faster development of new systems

Throughout this section, we will refer to the decision of how to structure class relationships as the underlying data model.

In software development, a data model is the representation of our objects and their properties within our application and how they relate to items in the real world.

For example, the data models of HR and Finance throughout this course contain attributes that support the operations required by the system.

Encapsulating Data

One fundamental principle of object-oriented design is the concept of encapsulating data.

In software development, encapsulation is the idea that although classes use fields and methods to provide functionality, the users of the class only have access via a defined interface (the public methods) and do not want or need visibility of how the class works.

In Java, it is commonly implemented with private instance members that have public methods to retrieve or modify the data, commonly referred to as getters and setters, respectively. Getters and setters are only required if the calling code sets those attributes.

NOTE

The OCP Java Programming II exam may ask questions about classes that are not encapsulate and which have public variables that can be accessed directly.

Even if a class allows attributes to be changed using a setter, any business rules about valid domain values for that attribute can be enforced in a setter method that would be impossible to enforce in a public variable.

Classes that have no setters are called ‘immutable’ as they cannot be changed.

There is a design principle called JavaBeans that provides rules for encapsulating classes as follows:

- Properties (fields) are private
- Properties start with a lower-case letter (lower camel-case)
- Getter method begins with is or get for boolean properties but NOT Boolean (wrapper class)
- Getter method begins with get for all other properties
- Setter method begins with set
- Follow is/get/set with the property name starting with an upper-case letter (so that the method name is still lower camel-case)

IS-A Relationships

Remember the instanceof operator can be used to determine when an object is an instance of a particular class, superclass, or interface. In object-oriented design, we describe the property of an object being an instance of a data type as having an IS-A relationship. Checking for the IS-A relationship is also known as the inheritance test.

The fundamental result of the is-a principle is that if A IS-A B, then any instance of A can be treated like an instance of B. This holds true for a child that is a subclass of any parent, be it a direct subclass or a distant child. As we discussed with polymorphism, objects can take many different forms.

When constructing an inheritance-based data model, it is important to apply the IS-A relationship regularly, so that you are designing classes that conceptually make sense.

For example, let's say that we have a class Intern that extends the Employee class.

The Intern class shares all the attributes and methods of an Employee class. Any processing of an Employee within the HR system applies equally to an Intern. Therefore it is safe to say the Intern IS-A Employee.

However, we now need to design a new class to represent a client contact. The ClientContact class has many of the attributes of an Employee. The name and address are practically identical. However there are many operations and quite a few attributes in the Employee class that are completely irrelevant to client contacts. After careful examination it is apparent that ClientContact does not have an IS-A relationship with Employee.

The HAS-A Relationship

In object-oriented design, we often want to model the relationship between two objects. For example, an employee has a home address. A postal address is too complicated to be a simple text or numeric value and so is modelled as a class. The PostalAddress class is then the data type of a member variable of the Employee class.

We refer to the HAS-A relationship as the property of an object having a named data object or primitive as a member. The HAS-A relationship is also known as object composition.

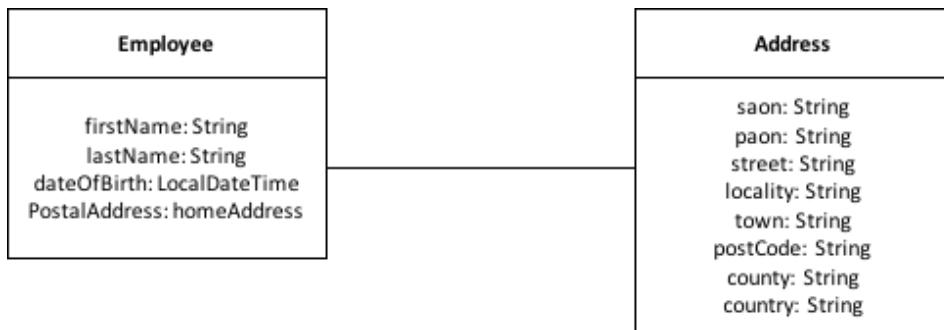


Figure 2-1 Object Composition

In this example, Employee and Address are both classes with different attributes and values. While they obviously fail the IS-A test, since a Address is not an Employee and Employee is not an Address. However Employee HAS-A Address.

Inheritance goes one step further by allowing us to say that any subclass of Employee, such as Intern, must also have an Address.

More generally, if a class HAS-A object as a protected or public member, then any subclass of the superclass must also have that object as a member. Note that this does not hold true for private members as they are not inherited in Java.

Design Patterns

A design pattern is an established general solution to a commonly occurring software development problem. The purpose of a design pattern is to leverage the experience and knowledge of developers who have come before you in order to solve some of the repeating problems that you may encounter.

It also usefully gives developers a common vocabulary with which they can discuss common problems and solutions.

For example, if you say that you are writing getters and setters or implementing the singleton pattern, most developers will understand the structure of your code without having to get into the low-level details.

In this section, we are primarily focused on creational patterns, a type of software design pattern that manages the creation of objects within an application. Obviously, you already know how to create objects in Java with the new keyword, as shown in the following code:

```
Employee e = new Employee(...);
```

This is straightforward enough providing there is only one option for which class you are going to use. In a more complex and flexible system you may not know until runtime which specific class will be required.

For example, you may be onboarding a new employee but are they an Employee, Intern or Apprentice? It may not be possible to hard-wire this into your code and you may need to rely on some programming logic to choose the appropriate class at runtime.

Singleton Pattern

The first creational pattern we will discuss is the singleton pattern.

The problem solved by the Singleton Pattern is how to share the same single object in memory between all the classes in an application.

An example is the Payroll class. At any one time there is a single Payroll object and all classes and users make additions to it until the payroll deadline when all the payments are sent to the bank.

Obviously it makes sense only to have one Payroll object but all classes need access to it so how can we make sure they all get a reference to the same object?

Here is a simple implementation:

```
package payroll;
import java.util.ArrayList;
import java.time.LocalDateTime;
public class Payroll {
    private static Payroll payroll = null;
    private ArrayList<Payment> payments = new ArrayList<>();
    protected Payroll(){}
    public static synchronized Payroll getInstance(){
        if(payroll == null){
            payroll = new Payroll();
        }
        return payroll
    }
    public void addPayment(String name, float amount){
        payments.add(new Payment(name, amount, LocalDateTime.now()));
    }
    // Other methods...
}
```

This example highlights the key features of a lazily-instantiated singleton.

First to note is that it has a protected constructor. This prevents direct instantiation. the only way to obtain a reference to an object of this type is via the static getInstance() method.

The getInstance() method only creates an object if the private static variable called payroll is null. This is called ‘lazy’ instantiation rather than ‘eager’ as it is only done where necessary, just in time for its first use. Thereafter any class requesting an instance will receive a reference to the same object thus ensuring it is a singleton.

Also note the synchronized modifier used with the getInstance() method. This ensures that only one thread can access this code. This prevents two threads simultaneously running the instantiation line and overwriting the reference variable payroll. If this were to happen, one thread would have access to its own Payroll object and all other threads would share another single object.

Immutable Objects

The next creational pattern we will discuss is the Immutable Object pattern.

An object is considered immutable if its state cannot change after it is constructed. Use of immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

The immutable object pattern is a creational pattern based on the idea of creating objects whose state does not change after they are created and can be easily shared across multiple classes. Immutable objects rely on full encapsulation of state, with the additional condition that no setter methods exist that modify the object. Since the state of an immutable object never changes, they are inherently thread-safe.

Examples of immutable objects from the Java API include String and Integer.

Immutable objects greatly simplify your program, since they:

- are simple to construct, test, and use
- are automatically thread-safe and have no synchronisation issues
- do not need an implementation of clone
- allow hashCode to use lazy initialisation, and to cache its return value
- don't need to be copied defensively when used as a field (like ArrayList for example)
- make good Map keys and Set elements (these objects must not change state while in the collection)
- have their class invariant established once upon construction, and it never needs to be checked again
- are never left in an undesirable or indeterminate state for example when an exception is thrown

Immutable objects have many positive qualities. They are among the simplest and most robust types of class you can design. With immutable classes, entire categories of problems are cut out of your code.

To make a class immutable, simply follow these guidelines:

- make the class final to ensure the class cannot be overridden, or use static factories and keep constructors private
- make fields private and final
- only allow state to be set in the constructor
- do not provide any methods which can change the state of the object
- if the class has any mutable class fields then they must be defensively copied when they pass between the class and its caller (via a getter for example)

Here is an example of a class definition that we want to be immutable:

```
package hr;
import java.util.ArrayList;
public class JobRole{
    private int id;
    private String name;
    private ArrayList<Skill> skills;
    public JobRole(int id, String name, ArrayList<Skill> skills){
        this.id = id;
        this.name = name;
        this.skills = skills;
    }
    public int getId(){
        return id;
    }
    public String getName(){
        return name;
    }
    public ArrayList<Skills> getSkills(){
        return skills;
    }
}
```

The problem with this is that `ArrayList` is not immutable. In fact `ArrayList` would be pretty useless if it were immutable as the whole point is that you can add and remove elements.

Looking at the constructor first, it accepts an `ArrayList<Skill>` as its third argument. This is a reference to an object. If the caller subsequently modifies the `ArrayList` that has been passed in, by adding a Skill to it for example, then this will modify the `JobRole` object's list of skills to include the new Skill object. This is clearly not what was intended.

The same would apply to any caller using the `getSkills()` method. This returns a reference to the `ArrayList` which could be modified once the reference is obtained and again this would violate the immutability of `JobRole`.

The only way to avoid this is to make copies of immutable objects when they pass between the immutable object and its callers.

The constructor becomes:

```
public JobRole(int id, String name, ArrayList<Skill> skills){
    this.id = id;
    this.name = name;
    this.skills = new ArrayList<Skill>(skills);
}
```

The `getSkills()` method becomes:

```
public ArrayList<Skills> getSkills(){
    return new ArrayList<Skills>(skills);
}
```

This is known as 'defensive copying'.

Builder Pattern

This pattern aims to solve the problem of having many fields where some are optional.

The Builder pattern may be used with mutable or immutable objects but can be designed to ensure any objects created are always in a consistent state.

The problem with multiple optional fields arises when designing the constructors.

Let's take the example of PostalAddress which has the following fields:

- saon (Secondary Addressable Object Name such as Flat 3)
- paon (Primary Addressable Object Name such as 371)
- street (such as High Street)
- locality (district of a town)
- town
- postcode
- county
- country

Of these, only paon, street, town and postcode are mandatory for construction. The country field defaults to the locale of the user. The others are all optional as not all addresses have them.

If we were to provide constructors to allow for this functionality we would require one constructor for the four mandatory fields and fifteen others if we wish to allow for all combinations of the optional fields.

This is a maintenance nightmare and PostalAddress is a fairly simple (and frequently used) class compared to some others you may come across.

One option is to provide a constructor for the mandatory fields and JavaBeans style setters for the others with a default initialisation for any fields where this is appropriate. This is ok but obviously would not work for immutable objects and may leaves open the possibility of the object being in a half-formed state.

The builder pattern can help as it provides an ancillary object to give all the values to which then returns the main object from (usually) a build() method.

Here is some example code for the PostalAddress class:

```
package general;
public class PostalAddress{
    private String saon;
    private String paon;
    private String street;
    private String locality;
    private String town;
    private String postCode;
    private String county;
    private String country;
    private PostalAddress(Builder builder){
```

```

        thi s. saon = buil der. saon;
        thi s. paon = buil der. paon;
        thi s. street = buil der. street;
        thi s. local i ty = buil der. local i ty;
        thi s. town = buil der. town;
        thi s. postCode = buil der. postCode;
        thi s. county = buil der. county;
        thi s. country = buil der. country;
    }
public static class Buil der{
    // Mandatory fields
    private String paon;
    private String street;
    private String town;
    private String postCode;

    // Optional fields
    private String saon = "";
    private String locality = "";
    private String county = "";
    private String country = "United Kingdom";
    public Buil der(String paon, String street,
                    String town, String postCode){
        thi s. paon = paon;
        thi s. street = street;
        thi s. town = town;
        thi s. postCode = postCode;
    }
    public Buil der saon(String saon){
        thi s. saon = saon;
    }
    public Buil der locality(String locality){
        thi s. locality = locality;
    }
    public Buil der county(String county){
        thi s. county = county;
    }
    public Buil der country(String country){
        thi s. country = country;
    }
    public Postal Address buil d(){
        return new Postal Address(thi s);
    }
}
}
}

```

An example of the use is shown here:

```

Postal Address stayahead = new Postal Address.Buil der(
    "6", "Long Lane",
    "London", "EC1A 9HF")
    .saon("3rd Floor"
    .locality("Barbican")
    .buil d();

```

This can be modified to allow for immutable or mutable objects and various combinations of mandatory and optional fields and default values.

It is also possible to validate the fields for formatting or other domain constraints as they are input and even to normalise the values before setting them – e.g. capitalising postal towns in the UK.

Other business rules can be applied during the operation of a builder and a meaningful exception raised if the build() method is called when a business rule is being broken.

Factory Pattern

This pattern is also known as the Factory Method pattern and is one of the most popular creational patterns. It is used to construct objects so that they can be decoupled from the implementing system. This is useful where the implementation is not known when writing the code. In this case, instantiation using the new keyword is impractical.

There are numerous examples in the Java API. For example, when creating a connection to a database a factory method is provided by the DriverManager class. This returns a Connection object which is provided by the vendor of the database being used (Oracle, IBM, Microsoft etc.)

Factory pattern is popular because it enforces the principle of coding to an interface rather than an implementation. The JDBC example above illustrates this point. Code written using JDBC interfaces like Connection and ResultSet is extremely flexible. The underlying DMMS being used could be switched with no changes to the code.

The definition of Factory Method provided in the original Gang of Four book (Gamma et al) Design Patterns states:

"Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses"

The following diagram illustrates this definition of the Factory Method pattern. The Creator hides the creation and instantiation of the Product from the client. This is a benefit to the client as they are now insulated from any future changes - the Creator will look after all of their creation needs, allowing to decoupling. Furthermore, once the Creator and the Product conform to an interface that the client knows, the client doesn't need to know about the concrete implementations of either. The factory method pattern really encourages coding to an interface in order to deal with future change.

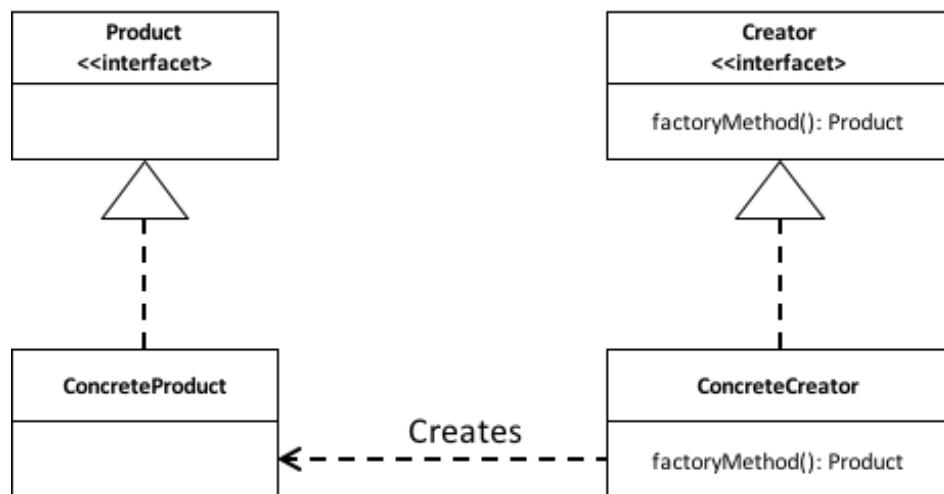


Diagram 2-2 Factory Method Design Pattern

The reason for using the Factory Method pattern is that it allows for the case where the client doesn't know what concrete classes it will be required to create at runtime, but just wants to get a class that implements a particular interface. Factories are used in logging frameworks, and in a lot of scenarios where the client doesn't need to know about the concrete implementations. It's a very good approach to de-coupling classes.

Here is some simple example code based on logging errors:

First the product interface:

```
//interface (Product)
public interface Logger {
    public void log(String message);
}
```

Then an implementation of the product:

```
//concrete implementation of the Logger (Product)
public class XMLLogger implements Logger {
    public void log(String message) {
        //log to xml System.out.println("Logging");
    }
}
```

Here is the abstract factory:

```
//the abstract Creator
public abstract class AbstractLoggerCreator {
    //the factory method
    public abstract Logger createLogger();
    //the operations that are implemented for all LoggerCreators
    //like anOperation() in the diagram
    public Logger getLogger(){
        //depending on the subclass, we'll get a particular logger.
        Logger logger = createLogger();
        //could do other operations on the logger here
        return logger;
    }
}
```

And the concrete factory:

```
//ConcreteCreator
public class XMLLoggerCreator extends AbstractLoggerCreator{
    @Override
    public Logger createLogger() {
        XMLLogger logger = new XMLLogger();
        return logger;
    }
}
```

And here is some code that uses the factory:

```
public class Client {
    private void someMethodThatLogs(AbstractLoggerCreator logCreator){
        Logger logger = logCreator.createLogger();
```

```
    logger.log("message");
}
public static void main(String[] args){
    //for this example, create an XMLLoggerCreator directly,
    //but this would normally be passed to constructor for use.
    AbstractLoggerCreator creator = new XMLLoggerCreator();
    Client client = new Client();
    client.someMethodThatLogs(creator);
}
}
```

Summary

The main focus of this chapter was how to write better code. We looked at techniques for designing class structures that scale naturally over time, integrate well with other applications, and are easy for other developers to read and understand.

We started off with a brief review of interfaces, showing how to declare, implement, and extend them. We then moved on to functional programming and reviewed the various syntax options available for defining functional interfaces and writing lambda expressions.

Given the prevalence of lambda expressions throughout Java 8, you really need to get to grips with writing and using lambda expressions.

We concluded the discussion with a review of the generics-based Predicate interface and showed how it can be used in place of your own functional interface. We will return to lambdas and streams later in the course.

We then introduced the concept of polymorphism, which is central to object-orientation and the Java language, and showed how objects can be accessed in a variety of forms. We covered when casts are needed for accessing objects, and how to spot the difference between compile-time and runtime cast problems.

In the design principles section, we looked at how to encapsulate classes in Java properly, allowing you to enforce class invariants in your data model. We then described the IS-A and HAS-A principles and showed how you can apply them to your data model.

We finished off by exploring design patterns and focusing on four well-known design patterns. Design patterns provide you with a way to solve a problem that you encounter using solutions that other developers have already built and generalised.

The singleton pattern is used for managing a single shared instance of an object within an application.

The Immutable Object pattern is useful for creating read-only objects that cannot be modified by other classes.

The Builder pattern solves the problem of how to create complex objects cleanly, and it is often used in conjunction with the immutable object pattern.

Finally, the Factory (Method) pattern is useful for creating various objects without exposing the underlying constructors and complex rules for selecting a particular object subtype.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 3

Generics and Collections

Introduction

Generics is a system of type specifications by which you can ensure type safety in your code and make sure that classes and more commonly collections only contain the types you want. The reason for the name of this system is because it allows a type or method to operate on objects of various types while providing compile-time type safety, making Java a fully statically typed language.

Therefore you (or the Oracle Java team) don't have to write a different ArrayList class to hold objects of different types, or Comparator interface to compare different objects. Instead you can write a generic class or collection that can be given the class name in code and the type safety will be enforced by the compiler.

Imagine doing this without generics and you would end up with many more classes and interfaces such as the following imagined list:

- StringArrayList
- IntegerArrayList
- CustomerArrayList
- StringComparator
- IntegerComparator
- CustomerComparator

Instead we just have one ArrayList and one Comparator and they are genericised. You provide the class name to make them type-safe:

- ArrayList<String>
- ArrayList<Integer>
- ArrayList<Customer>
- Comparator<String>
- Comparator <Integer>
- Comparator <Customer>

Collections, as mentioned above, have to work with many classes and provide a way of dealing with them in bulk. There are different families of collections that organise their objects in different ways such as those that implement the Set interface which only allow unique values or Map classes that have a unique key to quickly find entries.

Collections, generics and functional interfaces work well together. In this chapter we will focus on the main operations and uses of collections and generics, including some functional interfaces.

Later in the course you will come across some more specialised collections that are used in concurrent code for example but everything in this chapter applies equally to all collections.

Java Collections

The Java Collections Framework includes classes that implement List, Map, Queue, and Set. Previously, in the Java Programming 1 course, you saw one such class, ArrayList, that implements the interface List. You also learned about arrays and how they work such as int[] or Customer[].

Arrays are not part of the Collections Framework. However, since sorting and searching are similar between lists and arrays, both are covered here and are required for the OCP Java Programming II exam.

In the following sections, we will review arrays, ArrayLists, wrapper classes, autoboxing, the diamond operator, searching, and sorting.

Arrays and ArrayList

An ArrayList is a List object that contains other objects. An ArrayList cannot contain primitives, only reference variables.

An array is a built-in Java data structure (also an object) that contains other objects or primitives. The following code reviews how to use an array and ArrayList:

```
List<String> stringList = new ArrayList<>();
stringList.add("First");
stringList.add("Second");
String[] stringArray = new String[list.size()];
stringArray[0] = stringList.get(1);
stringArray[1] = stringList.get(0);
for (int i = 0; i < array.length; i++)
    System.out.print(array[i] + "-");
```

The output is:

```
Second-Fir
```

This code reminds us that Java uses zero-based indexes. It also reminds us that the method for accessing elements in ArrayLists is:

```
E get(int index)
```

Also that to get the number of elements in an ArrayList with:

```
int size()
```

By contrast, we access elements in arrays using square brackets and check the number of elements with the length property.

There's a simpler way to convert between arrays and Lists but it results in a link being created that affects the operation of both:

```
String[] array = { "keyboard", "mouse" };
List<String> list = Arrays.asList(array);
list.set(1, "pen");
```

```
array[0] = "new";
String[] array2 = (String[]) list.toArray();
list.remove(1);
```

The last line results in:

UnsupportedOperationException

The second line of code converts an array to a List. Note that it happens to be an implementation of List that is not an ArrayList but we don't know this from looking at the code.

Remember that a List is like a resizable array. It makes sense to convert an array to a List. It doesn't make sense to convert an array to a Set, which has very different behaviour from an array.

You still can do so if you want although it takes an extra step. You'll need to convert the array to a List first and then the List to a Set.

Note that you can change the elements in either the array or the List. Changes are reflected in both, since they are backed by the same data. This is the link.

Implementations of List may of course add their own behaviour. The implementation returned by the asList() method has the added restriction of not being resizable but still honours all of the other methods in the interface.

This includes the toArray() method which, as you might have guessed, converts the List back to an array. The last line demonstrates that list is not resizable (because it is backed by an underlying array).

Searching and Sorting

One other area to review is searching and sorting.

There is a utility class called Arrays that has a sort() method. It will sort the array in-place (i.e. it changes the array rather than returning a sorted copy) and does so in the natural order of the datatype contained. In this case we are dealing with integers so it sorts in increasing numerical order.

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

First, we sort the array. This is critical because binary search assumes the input is sorted. The first call to binarySearch() prints the index at which a match is found.

The next call returns one less than the negated index of where the requested value would need to be inserted to match the sort order. The missing number 3 would need to be inserted at index 1 (between 1 and 6). Negating that gives us -1 and subtracting 1 gives us -2. This seems complex but it is the only effective way to say both that the search item was not found and provide where it would go.

Let's see the same code but this time with a List:

```
List<Integer> list = Arrays.asList(9, 7, 5, 3);
Collections.sort(list); // [3, 5, 7, 9]
System.out.println(Collections.binarySearch(list, 3)); // 0
System.out.println(Collections.binarySearch(list, 2)); // -1
```

NOTE

For List objects, you call sort() and binarySearch() on Collections rather than Collection.

Before Java 8, Collection could not have concrete methods because it is an interface.

Some concrete methods were added in Java 8. We will see some of these later in the course.

You will see searching and sorting again later in this chapter and find out about how it works with the Comparable interface.

Wrapper Classes and Autoboxing

As a brief review, each primitive has a corresponding wrapper class, as shown here:

Primitive type	Wrapper type	Example instantiation
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte)100)
short	Short	new Short((short)100)
int	Integer	new Integer(100)
long	Long	new Long(100)
float	Float	new Float(100.0)
double	Double	new Double(100.0)
char	Character	new Character('X')

Autoboxing automatically converts a primitive to the corresponding wrapper classes when needed if the generic type is specified in the declaration. As you might expect, unboxing automatically converts a wrapper class back to a primitive.

Let's try an example, which also shows a 'gotcha' you may come across.

Look at this code and work out what the result is:

```
List<Integer> numberList = new ArrayList<Integer>();
numberList.add(1);
numberList.add(new Integer(3));
numberList.add(new Integer(5));
numberList.remove(1);
numberList.remove(new Integer(5));
System.out.println(numbers);
```

The answer is it leaves just [1]. To understand why that is, we need to look at the first call to `remove()`. We have added three `Integer` objects to `numberList` (which only allows `Integer` objects). The first one on line uses autoboxing of the literal integer 1 to do so.

After adding the other two `Integer` objects, `numberList` contains [1, 3, 5].

The first `remove()` line contains the trick. The `remove()` method is overloaded. One signature takes an `int` as the index of the element to remove. The other takes an `Object` that should be removed. Java sees a matching signature for `int`, so it doesn't need to autobox the call to the method.

Now `numberList` contains [1, 5]. Line 8 calls the other `remove()` method, and it removes the matching object, which leaves us with just [1].

Java will also convert the wrapper classes to primitives via unboxing:

```
int num = numbers.get(0);
```

The Diamond Operator

Java has come a long way. Before generics arrived with Java 5, you had to write code like the following and hope that other programmers would remember only to put `String` objects in your `ArrayList`:

```
List names = new ArrayList();
```

There was no way of knowing `names` was only supposed to contain `String` objects rather than `StringBuilder` or some other class.

In Java 5, you could use generics and write the following code which would also be enforced by the compiler:

```
List<String> names = new ArrayList<String>();
```

In Java 7 you can even miss out the contents of the second diamond brackets and it will be inferred:

```
List<String> names = new ArrayList<>();
```

This may seem like a small benefit but, as usual, there is more complex code to come! The following two examples show how diamond operator inference can really improve the readability of your code:

Which do you prefer:

```
HashMap<String, HashMap<String, String>> myMap = new HashMap<String,  
HashMap<String, String>>();
```

or:

```
HashMap<String, HashMap<String, String>> myMap = new HashMap<>();
```

Diamond operator inference works even if the declaration is on one line and the instantiation on another line. However, if the two lines are not near each other you might conclude that it is better not to use inference as anyone reading your code would have to check back to see what class was being used.

Using Generics

Why do we need generics? Well, remember when we said that we had to hope the caller didn't put something in the list that we didn't expect?

The following does just that:

```
static void printNames(List list) {
    for (int i = 0; i < list.size(); i++) {
        String name = (String) list.get(i);
        System.out.println(name);
    }
}

public static void main(String[] args) {
    List names = new ArrayList();
    names.add(new StringBuilder("Monitor"));
    printNames(names);
}
```

This code throws a `ClassCastException`. The `main` method adds a `StringBuilder` to `list`. This is legal because a non-generic list can contain anything.

However, the `printNames()` method is written to expect a specific class to be in there (it casts the return value of the `get()` method to a `String`).

This assumption is incorrect and so the code throws a `ClassCastException` stating that `java.lang.StringBuilder` cannot be cast to `java.lang.String`.

Generics fixes this by allowing you to write and use parameterised types. You specify that you want an `ArrayList` of `String` objects. Now the compiler has enough information to prevent you from causing this problem in the first place:

```
List<String> names = new ArrayList<String>();
names.add(new StringBuilder("Monitor"));
```

Now the compiler will alert anyone making a call to `printList()` if they try to put anything else in apart from a `String`.

Generic Classes

You can introduce generics into your own classes. The syntax for introducing a generic is to declare a formal type parameter in angle brackets.

For example, the following class named `Container` has a generic type variable declared after the name of the class:

```
public class Container<T> {
    private boolean empty = true;
    private T contents = null;
    public T unload() throws EmptyContainerException {
        if(contents == null) {
            throw new EmptyContainerException;
```

```
    }
    T temp = contents;
    contents = null;
    empty = true;
    return contents;
}
public void load(T contents) {
    this.contents = contents;
}
}
```

The generic type T is available anywhere within the Crate class.

When you instantiate the class, you tell the compiler what T should be for that particular instance.

Using the Container is easy.

When you want a Container object you just say what data type you want to store in it by using the diamond brackets:

```
Container<Employee> employeeContainer = new Container<>();
```

NOTE

Naming Conventions for Generics:

A type parameter could be named anything you want but it might be confusing.

The convention is to use single uppercase letters so it's obvious that they aren't real class names.

The following are the conventional letters to use:

- E for an element
- K for a map key
- V for a map value
- N for a number
- T for a generic data type
- S, U, V, and so forth for multiple generic types

Generic Interfaces

Just like a class, an interface can declare a formal type parameter. For example, the following Shippable interface uses a generic type as the argument to its ship() method:

```
public interface Transferrable<T> {
    void transfer(T t);
```

```
}
```

There are three ways a class can approach implementing this interface. The first is to specify the generic type in the class.

The following concrete class says that it deals only with Employee objects. This allows it declare the transfer() method with an Employee parameter:

```
class TransferEmployeeContainer implements Transferrable<Employee> {
    public void transfer(Employee t) { ... }
}
```

The next way is to create a generic class. The following concrete class allows the caller to specify the type of the generic:

```
class TransferrableContainer<U> implements Transferrable<U> {
    public void ship(U t) { ... }
}
```

In this example, the type parameter could have been named anything, including T. We used U in the example so that it isn't confusing as to what T refers to.

The final way is to not use generics at all. This is the old way of writing code. It generates a compiler warning about Transferrable being a raw type, but it does compile. Here the transfer() method has an Object parameter since the generic type is not defined:

```
class TransferrableCreate implements Transferrable{
    public void transfer(Object t) { ... }
}
```

Generic Methods

So far we've looked at formal type parameters declared on the class or interface level. It is also possible to declare them on the method level. This is often useful for static methods since they aren't part of an instance that can declare the type. However, it is also allowed on non-static methods as well.

In this example, the method uses a generic parameter:

```
public static <T> Container<T> transfer(T t) {
    System.out.println("Preparing " + t);
    return new Container<T>();
}
```

The method parameter is the generic type T. The return type is a Container<T>. Before the return type, we declare the formal type parameter of <T>.

Unless a method is obtaining the generic formal type parameter from the class/interface, it is specified immediately before the return type of the method. This can lead to some interesting-looking code!

```
public static <T> void sink(T t) { }
public static <T> T identity(T t) { return t; }
public static T noGood(T t) { return t; } // DOES NOT COMPILE
```

The first line shows the formal parameter type immediately before the return type of void. The second line shows the return type being the formal parameter type. It looks odd, but it is correct. The third line omits the formal parameter type, and therefore it does not compile.

Bounds

By now, you might have noticed that generics are quite restrictive which is good when that's what you want but they are not very flexible. Bounded wildcards solve this by restricting what types can be used in that wildcard position.

A bounded parameter type is a generic type that specifies a bound for the generic.

A wildcard generic type is an unknown generic type represented with a question mark (?). You can use generic wildcards in three ways, as shown in the table below:

Bound type	Syntax	Example
Unbounded wildcard	<?>	List<?> l =new ArrayList<String>();
Wildcard with an upper bound	<? extends type>	List<? extends Exception> l =new ArrayList<RuntimeException>();
Wildcard with a lower bound	<? super type>	List<? super Exception> l =new ArrayList<Object>();

Unbounded Wildcards

An unbounded wildcard represents any data type. You use ? when you want to specify that any type is OK with you. Let's suppose that we want to write a method that looks through a list of any type:

```
public static void printList(List<Object> list) {
    for (Object x: list) {
        System.out.println(x);
    }
}
public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords); // DOES NOT COMPILE
}
```

Something is wrong. It's true that a String is a subclass of an Object, however, List<String> cannot be assigned to List<Object>.

Ok, perhaps we don't really need a `List<Object>`. What we really need is a `List` of “whatever.” That's what `List<?>` is. The following code does what we expect:

```
public static void printList(List<?> list) {
    for (Object x: list) {
        System.out.println(x);
    }
}

public static void main(String[] args) {
    List<String> keywords = new ArrayList<>();
    keywords.add("java");
    printList(keywords);
}
```

The `printList()` method takes any type of list as a parameter. The `keywords` list is of type `List<String>`. That matches the requirement so this code compiles.

Upper-Bounded Wildcards

Let's try to write a method that adds up the total of a list of numbers. We've established that a generic type can't just use a subclass:

```
ArrayList<Number> list = new ArrayList<Integer>(); // DOES NOT COMPILE
```

Instead, we need to use a wildcard:

```
List<? extends Number> list = new ArrayList<Integer>();
```

The upper-bounded wildcard says that any class that extends `Number` or `Number` itself can be used as the formal parameter type:

```
public static long total(List<? extends Number> list) {
    long count = 0;
    for (Number number: list){
        count += number.longValue();
    }
    return count;
}
```

Something interesting happens when we work with upper bounds or unbounded wildcards. The list becomes logically immutable. Immutable means that the object cannot be modified.

Actually, you can remove elements from the list, but you cannot use `add()` or `set()` as they would not be type safe as the compiler cannot tell what the type is going to be at run time.

Lower-Bounded Wildcards

Let's try to write a method that adds a string “Speaker” to two lists:

```
List<String> strings = new ArrayList<String>();
strings.add("Speaker");
```

```
List<Object> objects = new ArrayList<Object>(strings);
addSound(strings);
addSound(objects);
```

The problem is that we want to pass a `List<String>` and a `List<Object>` to the same method.

To solve this problem, we need to use a lower bound:

```
public static void addSound(List<? super String> list) { // Lower bound
    list.add("Speaker");
}
```

With a lower bound, we are telling Java that the list will be a list of `String` objects or a list of some objects that are a superclass of `String`. Either way, it is safe to add a `String` to that list.

Lists, Sets, Maps, and Queues

A collection is a group of objects contained in a single object. The Java Collections Framework is a set of classes in `java.util` for storing collections. There are four main interfaces in the Java Collections Framework:

- **List:** A list is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an int index.
- **Set:** A set is a collection that does not allow duplicate entries.
- **Queue:** A queue is a collection that orders its elements in a specific order for processing. A typical queue processes its elements in a first-in, first-out order, but other orderings are possible.
- **Map:** A map is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

Here are some of the commonly used collections:

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

Not all collections in the Collections Framework actually implement the `Collection` interface. In other words, not all collections pass the IS-A test for `Collection`.

Specifically, none of the Map-related classes and interfaces extend from `Collection`.

So while `SortedMap`, `Hashtable`, `HashMap`, `TreeMap`, and `LinkedHashMap` are all thought of as collections, none are actually extended from `Collection-with-a- capital-C`.

To make things a little more confusing, there are really three overloaded uses of the word "collection":

- collection (lowercase c), which represents any of the data structures in which objects are stored and iterated over
- `Collection` (capital C), which is actually the `java.util.Collection` interface from which the `Set`, `List`, and `Queue` interfaces extend
- `Collections` (capital C and ends with s) is the `java.util.Collections` class that holds a number of static utility methods for use with collections

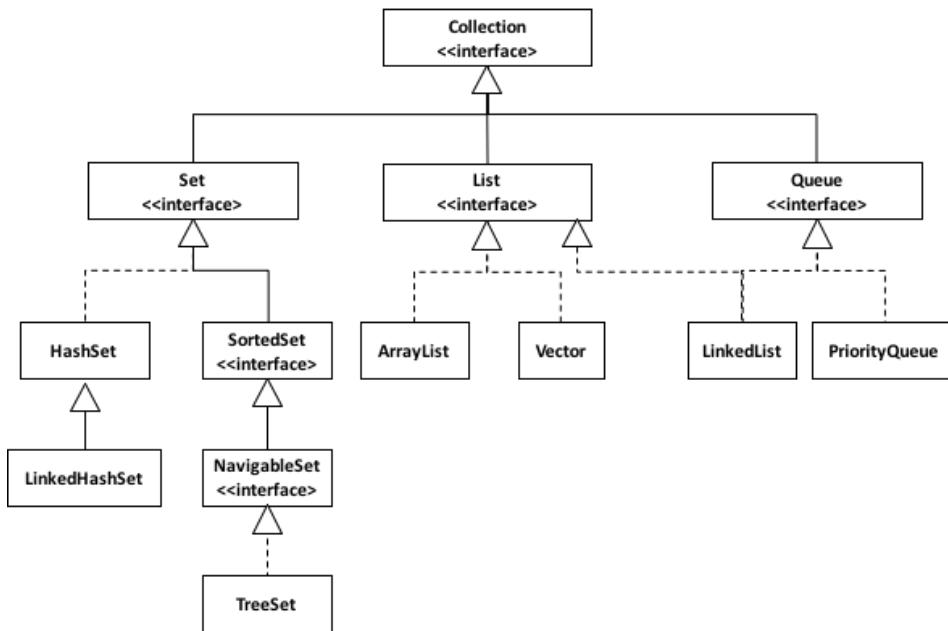


Figure 3-1 Collections Inheritance Hierarchy

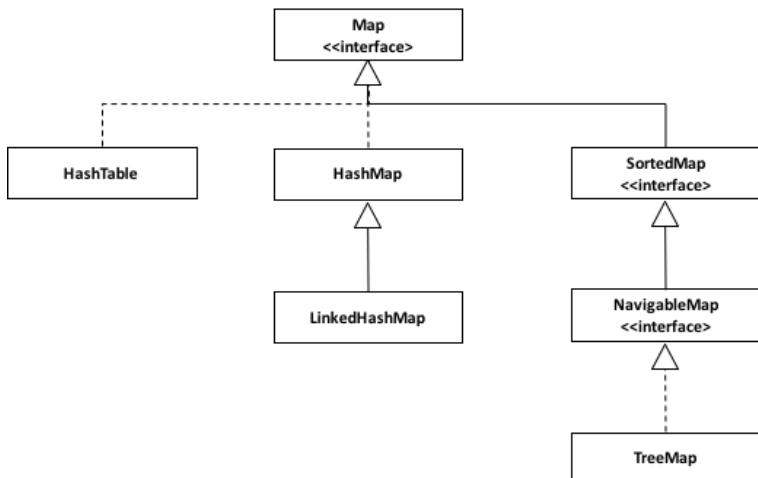


Figure 3-2 Map Inheritance Hierarchy

Common Collection Methods

The Collection interface contains useful methods for working with lists, sets, and queues. We will also cover maps. In the following sections, we will discuss the most common ones. We will cover stream in the next chapter and a couple others added in Java 8 at the end of this chapter.

add()

The add() method inserts a new element into the Collection and returns whether it was successful. The method signature is

```
boolean add(E element)
```

Remember that the Collections Framework uses generics. You will see E appear frequently. It means the generic type that was used to create the collection. For some collection types, add() always returns true.

For other types, there is logic as to whether the add was successful. The following shows how to use this method:

```
List<String> list = new ArrayList<>();
System.out.println(list.add("Hello")); // true
System.out.println(list.add("Hello ")); // true

Set<String> set = new HashSet<>();
System.out.println(set.add("Hello ")); // true
System.out.println(set.add("Hello ")); // false
```

A List allows duplicates, making the return value true each time. A Set does not allow duplicates. When we tried to add a duplicate, the set (HashSet) returns false from the add() method.

isEmpty() and size()

The isEmpty() and size() methods look at how many elements are in the Collection. The method signatures are

```
boolean isEmpty()
int size()
```

The following shows how to use these methods:

```
System.out.println(list.isEmpty()); // true
System.out.println(list.size()); // 0
```

clear()

The clear() method provides an easy way to discard all elements of the Collection. The method signature is

```
void clear()
```

The following shows how to use this method:

```
list.add("Desktop"); // [Desktop]
list.add("Notebook"); // [Desktop, Notebook]
System.out.println(list.isEmpty()); // false
System.out.println(list.size()); // 2
list.clear(); // []
System.out.println(list.isEmpty()); // true
System.out.println(list.size()); // 0
```

contains()

The contains() method checks if a certain value is in the Collection. The method signature is

```
boolean contains(Object object)
```

The following shows how to use this method:

```
List<String> binds = new ArrayList<>();
binds.add("Notebook"); // [Notebook]
System.out.println(binds.contains("Notebook")); // true
System.out.println(binds.contains("Desktop")); // false
```

This method calls equals() on each element of the ArrayList to see if there are any matches.

List Interface

You use a list when you want an ordered collection that can contain duplicate entries. Items can be retrieved and inserted at specific positions in the list based on an int index much like an array. Lists are commonly used because there are many situations in programming where you need to keep track of a list of objects.

For example, you might make a list of committee meetings. Each entry records the type and name of the committee together with the date, time and location of the meeting. Over time, the same committee meets many times. It makes sense to have duplicate entries for the same committee.

Sometimes, the order of elements in a list is not the main factor. For example, a shopping list has a random order, probably the order of the list happens to be the order in which we thought of the items. You probably won't do the shopping in that particular order, but it is a perfectly workable record of what you want to buy.

Classes implementing the List interface may have many additional methods but we will start with those they all must have (in the List interface) and some of the other most common ones.

The main thing that all List implementations have in common is that they are ordered and allow duplicates. Beyond that, they each offer different functionality. We will look at the implementations of some useful ones and their methods.

List Implementations

An ArrayList is like a resizable array. When elements are added, the ArrayList automatically grows. When you aren't sure which collection to use, use an ArrayList.

ArrayList is the most popular collection for Java programmers.

The main benefit of an ArrayList is that you can look up any element in constant time. Adding or removing an element is slower than accessing an element. This makes an ArrayList a good choice when you are doing more reading than writing (or the same amount).

A LinkedList is slightly different as you can see from the illustration of the Collection inheritance hierarchy illustrated above. It implements both List and Queue where the other concrete objects implement a single interface. It has all of the methods of a List but also has methods for adding or removing from the beginning and/or end of the list.

The main benefits of a LinkedList are that you can access, add, and remove from the beginning and end of the list in constant time. The trade-off is that dealing with an arbitrary index takes linear time. This makes a LinkedList a good choice when you'll be using it as Queue.

There are also two old implementations. Vector used to be the only choice if you wanted a list. ArrayList essentially replaced it In Java 1.2 as they are 100% identical except that Vector is thread safe and, as a result, slower. You only need to know about Vector in case you have some old code that might refer to it.

A Stack is a data structure where you add and remove elements from the top of the stack. Think about a to-do list as an example. Stack extends Vector and is still supported for compatibility with old code. If you need a stack, use an ArrayDeque instead which we will see in the Queue section later.

Big O Notation

Big O notation is used to talk about the performance of algorithms. Obviously you would prefer code to take 1 second than 10 seconds to do the same thing. This is called an order of magnitude difference (x 10).

Big O notation assumes the worst-case response time. If you write an algorithm that could take a while or be instantaneous, big O uses the longer one. It uses an ‘n’ to reflect the number of elements or size of the data you are talking about. The following lists the most common big O notation values that you will see and what they mean:

- O(1)—constant time: It doesn’t matter how large the collection is, the answer will always take the same time to return.
- O(log n)—logarithmic time: Logarithms grow much more slowly than the data size and is better than linear time but worse than constant time. Binary search runs in logarithmic time because it doesn’t need to look at the majority of the elements for large collections.
- O(n)—linear time: The performance will increase linearly with respect to the size of the collection. Looping through a list and returning the number of elements matching “Desktop” will take linear time as the algorithm needs to check every element.
- O(n²)—n squared time: Code that has nested loops where each loop goes through the data takes n squared time. An example would be matching every pair of computers together to see if they are compatible. This kind of algorithm visits each element multiple times.

List Methods

The methods in the List interface work with indexes. In addition to the inherited Collection methods, the method signatures that you need to know are shown in this table:

Method	Description
void add(E element)	Adds element to end
void add(int index, E element)	Adds element at index and moves the rest toward the end
E get(int index)	Returns element at index

<code>int indexOf(Object o)</code>	Returns first matching index or -1 if not found
<code>int lastIndexOf(Object o)</code>	Returns last matching index or -1 if not found
<code>void remove(int index)</code>	Removes element at index and moves the rest toward the front
<code>E set(int index, E e)</code>	Replaces element at index and returns original

The following statements demonstrate these basic methods for adding and removing items from a list:

```
List<String> list = new ArrayList<>();
list.add("USB"); // [USB]
list.add(0, "SD"); // [SD, USB]
list.set(1, "CD"); // [SD, CD]
list.remove("SD"); // [CD]
list.remove(0); // []
```

The `ArrayList` list starts out empty. Then we add an element to the end of the list. The next line adds an element at index 0 that pushes the original index 0 to index 1 and automatically grows the `ArrayList`. We then replace the element at index 1 with a new value. The first call to `remove()` removes the element matching “SD” and the next one removes the element at index 0 and so list is empty again.

Let's look at one more example that queries the list:

```
List<String> list = new ArrayList<>();
list.add("USB"); // [USB]
list.add(0, "SD"); // [SD, USB]
list.add("CD"); // [SD, USB, CD]
String storage = list.get(0); // "SD"
int index = list.indexOf(USB); // 1
```

The output would be the same if you changed the code to use `LinkedList`, `Vector`, or `Stack`. Although the code would be less efficient, it wouldn't be noticeable until you ran it with very large lists.

Iterating through a List

You will have used an enhanced for loop like this to run through a list or array:

```
for (String string: list) {
    System.out.println(string);
}
```

You'll see another longer way to do this in old code written before Java 5:

```
Iterator iter = list.iterator();
```

```
while(iterator.hasNext()) {  
    String string = (String) iterator.next();  
    System.out.println(string);  
}
```

The old way requires casting because it predates generics. It also requires checking if the Iterator has any more elements followed by requesting the next element.

There is a hybrid way where you still use Iterator with generics. You get rid of the cast but still need to handle the looping logic yourself.

```
Iterator<String> iterator = list.iterator();  
while(iterator.hasNext()) {  
    String string = iterator.next();  
    System.out.println(string);  
}
```

Pay attention to the methods used here. `hasNext()` checks if there is a next value, i.e. it tells you whether `next()` will execute without throwing an exception. The `next()` method actually moves the Iterator to the next element.

Set Interface

You use a set when you don't want to allow duplicate entries. For example, you might want to keep track of the unique products that you want to sell at a supermarket. You aren't concerned with the order in which you see these products, but it doesn't make sense to have them more than once. You just want to make sure that you see the ones that are important to you and remove them from the set of current products to put on the shelves.

This diagram shows how you can picture a Set. The main thing that all Set implementations have in common is that they do not allow duplicates. Beyond that, they each offer different functionality. We will look at some of the most common implementations and how to write code using Set.

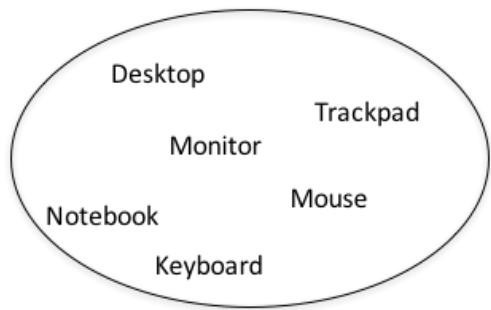


Figure 3-3 Set of Products

Set Implementations

A HashSet stores its elements in a hash table. This means that it uses the hashCode() method of the objects to retrieve them more efficiently. We covered how hashCode() works in chapter 1 you may remember.

The main benefit is that adding elements and checking if an element is in the set both have constant, O(1) time. The trade off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a set anyway, making HashSet the most common set.

A TreeSet stores its elements in a sorted tree structure. The main benefit is that the set is always in sorted order. The trade off is that adding and checking if an element is present are both O(log n).

TreeSet implements a special interface called NavigableSet, which lets you slice up the collection as shown in this table:

Method	Description
E lower(E e)	Returns greatest element that is < e, or null if no such element

E floor(E e)	Returns greatest element that is $\leq e$, or null if no such element
E ceiling(E e)	Returns smallest element that is $\geq e$, or null if no such element
E higher(E e)	Returns smallest element that is $> e$, or null if no such element

Set Methods

The Set interface has the same methods as List but sets behave differently with respect to the Collection methods.

Here is an example with HashSet:

```
Set<Integer> set = new HashSet<>();
boolean b1 = set.add(66); //true
boolean b2 = set.add(10); //true
boolean b3 = set.add(66); //false
boolean b4 = set.add(8); //true
for (Integer integer: set){
    System.out.print(integer + ","); // 66, 8, 10
}
```

The add() methods are the same as for List objects, they return true unless the element is already in the set. This means we get false when adding 66 again because it is already in the set which preserves uniqueness. When we print the elements of the set, they are retrieved in an arbitrary order. In this case, it happens neither to be sorted order, nor the order in which we added the elements.

Remember that the equals() method is used to determine equality. The hashCode() method is used to know which bucket to look in so that Java doesn't have to look through the whole set to find out if an object is there. The best case is that hash codes are unique, and Java has to call equals() on only one object. The worst case is that all implementations return the same hashCode(), and Java has to call equals() on every element of the set anyway.

Now let's look at the same example with TreeSet:

```
Set<Integer> set = new HashSet<>();
boolean b1 = set.add(66); //true
boolean b2 = set.add(10); //true
boolean b3 = set.add(66); //false
boolean b4 = set.add(8); //true
for (Integer integer: set){
    System.out.print(integer + ","); // 8, 10, 66
```

This time, the elements are printed out in their natural sorted order. Numbers implements the Comparable interface in Java, which is used for sorting. Later in the chapter, we will look at how to create your own Comparable objects.

Queue Interface

You use a queue when elements are added and removed in a specific order. Queues are typically used for sorting elements prior to processing them. For example, if you have a number of payments to be made on a ‘first come, first served’ basis then a Queue can do this.

Unless stated otherwise, a queue is assumed to be FIFO (first-in, first-out). Some queue implementations change this to use a different order. The other common format is LIFO (last-in, first-out.)

All queues have specific requirements for adding and removing the next element. Beyond that, they each offer slightly different functionality. We will look at the implementations that you are likely to need to know and the available methods that go with these.

Comparing Queue Implementations

You saw LinkedList earlier in the List section. In addition to being a list, it is a double-ended queue. A double-ended queue is different from a regular queue in that you can insert and remove elements from both the front and back of the queue.

For example, you may wish to give some elements special treatment, “Don Corleone, sir, come right to the front! Everyone else get to the back of the line!”

The main benefit of a LinkedList is that it implements both the List and Queue interfaces. The trade off, as usual, is efficiency. LinkedList is just not as efficient as a “pure” queue.

An ArrayDeque is a “pure” double-ended queue. It was introduced in Java 6, and it stores its elements in a resizable array. The main benefit of an ArrayDeque is that it is more efficient than a LinkedList.

The Java people at Oracle say that Deque is supposed to be pronounced “deck,” but many programmers say it wrong as “d-queue” as it makes more sense so please, take your choice.

Working with Queue Methods

The ArrayDeque contains many methods. Luckily, there are only seven new methods that you need to know in addition to the inherited Collection ones. These methods are shown in this table:

Method	Description	Queue	Stack
boolean add(E e)	Adds an element to the back of the queue and returns true or throws an exception	Yes	No
E element()	Returns next element or throws an exception if empty	Yes	No
boolean offer(E e)	Adds an element to the back of the queue and returns whether successful	Yes	No

E remove()	Removes and returns next element or throws an exception if empty queue	Yes	No
void push(E e)	Adds an element to the front of the queue	Yes	Yes
E poll()	Removes and returns next element or returns null if empty queue	Yes	No
E peek()	Returns next element or returns null if empty queue	Yes	Yes
E pop()	Removes and returns next element or throws an exception if empty queue	No	Yes

Except for push, all are in the Queue interface as well. The push() method is what makes it a double-ended queue.

There are essentially two types of method. One type throws an exception when something goes wrong. The other uses a different return value when something goes wrong. The offer/poll/peek methods are more common. This is the standard language programmers use when working with queues.

Let's look at an example that uses some of these methods:

```
Queue<Integer> queue = new ArrayDeque<>();
System.out.println(queue.offer(31)); // true
System.out.println(queue.offer(3)); // true
System.out.println(queue.peek()); // 31
System.out.println(queue.poll()); // 31
System.out.println(queue.poll()); // 3
System.out.println(queue.peek()); // null
```

Having said that ArrayDeque is a double-ended queue, how do we insert an element at the other end, as we did with the Stack? The answer is we just call the push() method. It works just like offer() except at the other end of the queue.

With LIFO (stack), use push/poll/peek.

With FIFO (single-ended queue), use offer/poll/peek.

Now let's rewrite the previous example using the stack functionality:

```
ArrayDeque<Integer> stack = new ArrayDeque<>();
stack.push(31);
stack.push(3);
System.out.println(stack.peek()); // 3
System.out.println(stack.poll()); // 3
System.out.println(stack.poll()); // 31
System.out.println(stack.peek()); // null
```

The difference between whether an ArrayDeque is being used as a stack or a queue is critical when interpreting code. A queue is like a line of people waiting for the bus, you join at the back and the person at the front gets on the bus first. A stack is like a stack of

dirty plates to be washed up. You put the plates on top and take the one on top and wash it up first. Since the stack is implemented using ArrayDeque, we refer to “top” and “front” interchangeably.

A LinkedList works the exact same way as ArrayDeque, so the code would look exactly the same apart from the declaration.

You use a map when you want to identify values by a key. For example, when you use the contact list in your phone, you look up “George” rather than looking through each phone number in turn.

Map Interface

A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects. You're probably quite familiar with Maps since many languages support data structures that use a key/value or name/value pair. The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys. Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

You can visualise a Map as shown this diagram:

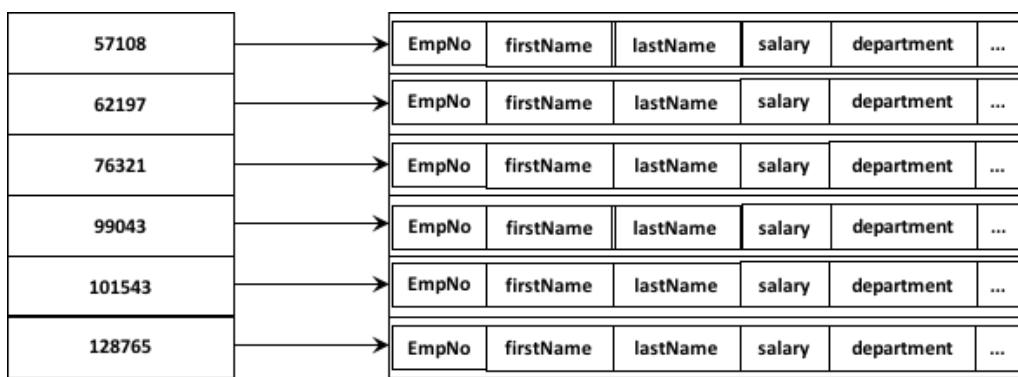


Figure 3-6 Map Object

You use a map when you want to identify values by a key. For example, when you use the contact list in your phone, you look up “Sharon” rather than looking through each phone number in turn.

HashMap

The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order when you iterate through it, then HashMap is the way to go; the other maps add a little more overhead.

Where the keys end up in the Map is based on the key's hashCode, so, like HashSet, the more efficient your hashCode() implementation, the better access performance you'll get. HashMap allows one null key and multiple null values in a collection.

A HashMap stores the keys in a hash table. This means that it uses the hashCode() method of the keys to retrieve their values more efficiently.

The main benefit is that adding elements and retrieving the element by key both have constant time. The trade off is that you lose the order in which you inserted the elements. Most of the time, you aren't concerned with this in a map anyway. If you were, you could use LinkedHashMap.

Hashtable

Like Vector, Hashtable has existed from prehistoric Java times. Note the naming inconsistency: HashMap vs. Hashtable. Where's the capitalisation of 't'? Anyway, just as Vector is a synchronised counterpart to the sleeker, more modern ArrayList, Hashtable is the synchronised counterpart to HashMap.

Remember that you don't synchronise a whole class, so when you say that Vector and Hashtable are synchronised, it just means that the key methods of the class are synchronised. Another difference, though, is that while HashMap lets you have null values as well as one null key, a Hashtable doesn't allow null keys or values.

LinkedHashMap

Like its Set counterpart, LinkedHashSet, the LinkedHashMap collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

TreeMap

You can probably guess by now that a TreeMap is a sorted Map. And you already know that, by default, this means "sorted by the natural order of the elements."

Like TreeSet, TreeMap lets you define a custom sort order (via a Comparator) when you construct a TreeMap that specifies how the elements should be compared to one another when they're being ordered. As of Java 6, TreeMap implements NavigableMap.

A TreeMap stores the keys in a sorted tree structure. The main benefit is that the keys are always in sorted order. The tradeoff is that adding and checking if a key is present are both $O(\log n)$.

Map Methods

Given that Map doesn't extend Collection, there are more methods specified in the Map interface. Since there are both keys and values, we need generic type parameters for both. The class uses K for key and V for value. Most of the method signatures that you need to know how to use are shown in this table:

Method	Description
void clear()	Removes all keys and values from the map.
boolean isEmpty()	Returns whether the map is empty.

<code>int size()</code>	Returns the number of entries (key/value pairs) in the map.
<code>V get(Object key)</code>	Returns the value mapped by key or null if none is mapped.
<code>V put(K key, V value)</code>	Adds or replaces key/value pair. Returns previous value or null.
<code>V remove(Object key)</code>	Removes and returns value mapped to key. Returns null if none.
<code>boolean containsKey(Object key)</code>	Returns whether key is in map.
<code>boolean containsValue(Object)</code>	Returns value is in map.
<code>Set<K> keySet()</code>	Returns set of all keys.
<code>Collection<V> values()</code>	Returns Collection of all values.

Let's compare running the same code with two Map types. First, HashMap:

```
Map<String, String> map = new HashMap<>();
map.put("GT", "Guatamala");
map.put("HU", "Hungary");
map.put("ET", "Ethiopia");
map.put("NP", "Nepal");
map.put("LB", "Lebanon");
String country = map.get("HU"); // Hungary
for (String key: map.keySet()){
    System.out.print(key + " "); // GT ET LB HU NP
}
```

Java uses the hashCode() of the key to determine the order. The order here happens to not be sorted order, or the order in which we typed the values.

Let's look at the same code using TreeMap:

```
Map<String, String> map = new TreeMap<>();
map.put("GT", "Guatamala");
map.put("HU", "Hungary");
map.put("ET", "Ethiopia");
map.put("NP", "Nepal");
map.put("LB", "Lebanon");
String country = map.get("HU"); // Hungary
for (String key: map.keySet()){
    System.out.print(key + " "); // ET GT HU LB NP
}
```

TreeMap sorts the keys in natural order as we would expect. If we were to have called values() instead of keySet(), the order of the values would still correspond to the order of the keys.

With our same map, we can try some boolean checks:

```
System.out.println(map.contains("NP")); // DOES NOT COMPILE
System.out.println(map.containsKey("NP")); // true
System.out.println(map.containsValue("NP")); // false
System.out.println(map.size()); // 5
```

The first line gives a compiler error as contains() is a method on the Collection interface but not the Map interface. The next two lines show that keys and values are checked separately. Finally, we see that there are five elements (key/value pairs) in our map.

Comparing Collection Types

Let's start off with a brief review of the characteristics of the different interfaces:

Interface	Duplicates	Ordered	Keys	Add/remove in specific order
List	Yes	Yes	No	No
Map	Yes (values)	No	Yes	No
Queue	Yes	Yes	No	Yes
Set	No	No	No	No

Now let's look at some of the specific collection classes:

Data Type	Interface	Sorted	Uses hashCode()	Uses compareTo()
ArrayList	List	No	No	No
ArrayDeque	Queue	No	No	No
HashMap	Map	No	Yes	No
HashSet	Set	No	Yes	No
Hashtable	Map	No	Yes	No
LinkedList	List, Queue	No	No	No
Stack	List	No	No	No
TreeMap	Map	Yes	No	Yes
TreeSet	Set	Yes	No	Yes
Vector	List	No	No	No

It is important to know which data structures allow nulls. Most do allow nulls, so let's just look at the exceptions.

The data structures that involve sorting do not allow nulls. This makes sense. We can't compare a null and a String. They are completely different things. We wouldn't say that 5 is less than "Equador" It doesn't make any more sense to say that null is less than "Vietnam" either. This means that TreeSet cannot contain null elements. It also means that TreeMap cannot contain null keys. Null values are OK though.

Next comes ArrayDeque. You can't put null in an ArrayDeque because methods like poll() use null as a special return value to indicate that the collection is empty. Since null has that meaning, Java forbids putting a null in there. That would just be confusing.

Finally, Hashtable doesn't allow null keys or values. There's not really a good reason for this one. It's just because it is old and was written that way.

In handy list form, all data structures allow nulls except these:

- TreeMap—no null keys
- Hashtable—no null keys or values
- TreeSet—no null elements
- ArrayDeque—no null elements

Comparator and Comparable Interfaces

We discussed sorted order for the TreeSet and TreeMap classes.

Some things just have a natural order. For numbers, natural order is obvious, it is numerical order. For String objects, order is defined according to the Unicode character mapping. As far as the exam is concerned, that means numbers sort before letters and uppercase letters sort before lowercase letters. Mozambique comes after Morocco of course.

You can also give objects that you create a natural order. This would be hard for Java to work out for itself. What order would you say is natural for the Aircraft class? What about Vehicle? The answer is very context dependent and there may not be anything very useful. Even so it is a good idea to choose an order so that your classes will work with collections.

To allow you to choose the sort order, Java provides an interface called Comparable. If your class implements Comparable, it can be used in these data structures that require comparison.

There is also an interface called Comparator, which is used to specify that you want to use a different order than the object itself provides. This means your code can sort the same object many ways. There can still only be one natural order though.

It's easy to get these two interfaces mixed up. In this section, we will discuss both of them and point out the differences.

Comparable

The Comparable interface has only one method. In fact, this is the entire interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Notice the use of generics in the interface and method signatures.

This lets you avoid using a cast when implementing the compareTo() method. Any object can be Comparable. For example, we have a collection of Aircraft and want to sort them by maximum speed:

```
import java.util.Comparable;
public class Aircraft implements Comparable<Aircraft> {
    private String name;
    private int maxSpeed;
    public Aircraft(String name, int maxSpeed) {
        this.name = name;
        this.maxSpeed = maxSpeed
    }
}
```

```

public String getName() { return name; }
public int getMaxSpeed() { return maxSpeed; }
@Override
public String toString() {
    return name + ": " + maxSpeed;
}
@Override
public int compareTo(Aircraft a) {
    return maxSpeed - a.maxSpeed;
}
public static void main(String[] args) {
    List<Aircraft> aircraft = new ArrayList<>();
    aircraft.add(new Aircraft("Spitfire", 370));
    aircraft.add(new Aircraft("B-17", 287));
    aircraft.add(new Aircraft("Sea Fury", 460));
    aircraft.add(new Aircraft("Mosquito", 356));
    Collections.sort(aircraft); // sort by maxSpeed
    System.out.println(aircraft);
    // [B-17: 287, Mosquito: 356, Spitfire: 370, Sea Fury: 460]
}
}

```

The Javadocs page for Comparable gives the following method detail for compareTo:

Parameters:

o - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

NullPointerException - if the specified object is null

ClassCastException - if the specified object's type prevents it from being compared to this object.

NOTE

compareTo() and equals() Consistency:

If you write a class that implements Comparable, you introduce new business logic for determining equality. The compareTo() method returns 0 if two objects are equal, while your equals() method returns true if two objects are equal.

A natural ordering that uses compareTo() is said to be consistent with equals if, and only if, *x.equals(y)* is true whenever *x.compareTo(y)* equals 0.

You are strongly encouraged to make your Comparable classes consistent with equals because not all collection classes behave predictably if the compareTo() and equals() methods are not consistent.

Comparator

Sometimes you want to sort an object that did not implement Comparable. More commonly you may want to sort objects in different ways at different times. Even if the object implements Comparable, that only gives it one natural sort order.

Suppose that we want to sort Aircraft objects by name rather than maximum speed. You might think that is a more natural order but it really depends on the use of that object in the application.

No problem though as we can (fairly) easily sort in any order we like. Here's an example with a local inner class in the main method:

```
public static void main(String[] args) {
    // ...
    // Declare and populate aircraft ArrayList as before
    // ...
    Comparator<Aircraft> byName = new Comparator<>(){
        public int compare(Aircraft a1, Aircraft a2){
            return a1.getName().compareTo(a2.getName());
        }
    }
    Collections.sort(aircraft); // sort by maxSpeed (natural order)
    System.out.println(aircraft);
    // [B-17: 287, Mosquito: 356, Spitfire: 370, Sea Fury: 460]
    Collections.sort(aircraft, byName); // Now sort by name
    System.out.println(aircraft);
    // [B-17: 287, Mosquito: 356, Sea Fury: 460, Spitfire: 370]
```

First, we defined an inner class to implement Comparator and override compare(). Then we sorted without the comparator (natural order) and then with the comparator to see the difference in output.

Comparator is a functional interface since there is only one abstract method to implement. This means that we can rewrite the comparator in the previous example using a lambda expression:

```
Comparator<Aircraft> byName =
    (d1, d2) ->d1.getName().compareTo(d2.getName());
```

Or even rewrite the call to Collections.sort() as follows:

```
Collections.sort(aircraft,
    (d1, d2) ->d1.getName().compareTo(d2.getName()));
```

Searching and Sorting

You already know the basics of searching and sorting and now you know how the Comparable and Comparator interfaces help with sorting..

The sort method uses the compareTo() method to sort. It expects the objects to be sorted to implement the Comparable interface.

```
import java.util.*;
public class SortReports {
    static class Airport{
        private String iataCode;
        public Airport(String iataCode){
            this.iataCode = iataCode
        }
        public String getIataCode(){
            return iataCode;
        }
        public static void main(String[] args) {
            List<Airport> airports = new ArrayList<>();
            airports.add(new Airport());
            Collections.sort(airports); // DOES NOT COMPILE
        }
    }
}
```

Java knows that the Airport class is not a Comparable object. It knows the sort() method will fail, so it doesn't even let the code compile. You can fix this by passing a Comparator to sort() as the second argument. Remember that a Comparator is used when you want to specify sort order with or without using a compareTo() method to provide a default natural ordering of objects.

Simply substitute this code for the line that doesn't compile above:

```
Comparator<Airport> byIataCode =
    (a1, a2) ->a1.getIataCode().compareTo(a2.getIataCode());
Collections.sort(airports, byIataCode); // COMPILES AND SORTS OK
```

The sort() and binarySearch() methods allow you to pass in a Comparator object when you don't want to use the natural order or when you can't use the natural order because there isn't one. You can tell if there is a natural order because the object implements Comparable (it would pass the instanceof test).

We saw that the sort() method overload with a single Collection argument fails to compile if the Collection object is not Comparable. The binarySearch() method also has an overload which assumes the collection being passed in is Comparable and it will also fail to compile if this is not the case.

Both methods provide an overload that accepts a Comparator to indicate the sort order.

The binarySearch() method relies on the collection having been sorted otherwise the search returns an unpredictable result. You may already have guessed from this that if

you are going to use a Comparator with binarySearch() then you need to have sorted the collection using the same Comparator.

Comparators are helpful objects. They let you separate sort order from the object to be sorted.

Method References

Method references (new in Java 8) are a way to make the code shorter by eliminating some of the code that can be inferred and simply mentioning the name of the method. Like lambdas, it takes time to get used to the new syntax.

Suppose that we have the Aircraft class we saw previously with name and maxSpeed attributes along with a new helper class:

```
public class AircraftHelper {  
    public static int compareByMaxSpeed(Aircraft a1, Aircraft a2) {  
        return a1.getMaxSpeed() - a2.getMaxSpeed();  
    }  
    public static int compareByName(Aircraft a1, Aircraft a2) {  
        return a1.getName().compareTo(a2.getName());  
    }  
}
```

Now think about how we would write a Comparator for it if we wanted to sort by weight. Using lambdas, we'd have the following:

```
Comparator<Duck> byMaxSpeed =  
    (a1, a2) -> AircraftHelper.compareByMaxSpeed(d1, d2);
```

This looks ok but it can be simplified. The lambda takes two parameters and does nothing but pass those parameters to another method. Java 8 lets you remove that redundancy and simply write this:

```
Comparator<Aircraft> byMaxSpeed = AircraftHelper::compareByMaxSpeed;
```

The :: operator tells Java to pass the parameters automatically into compareByMaxSpeed. AircraftHelper::compareByMaxSpeed returns a functional interface and not an int. Remember that :: is an extension of the syntax of lambda expressions, and it is typically used for deferred execution.

There are four formats for method references:

- Static methods
- Instance methods on a particular instance
- Instance methods on an instance to be determined at runtime
- Constructors

In this section, you will see three functional interfaces in the examples.

Remember that Predicate is a functional interface that takes a single parameter of any type and returns a boolean.

Another functional interface we haven't talked about yet is Consumer, which takes a single parameter of any type and has a void return type.

Finally, the Supplier functional interface doesn't take any parameters and returns any type.

Static Methods

Let's look at some examples from the Java API. In each set, we show the lambda equivalent. Remember that none of these method references are actually called in the code that follows. They are simply available to be called in the future. Let's start with a static method:

```
Consumer<List<Integer>> methodRef1 = Collections::sort;
Consumer<List<Integer>> lambda1 = l -> Collections.sort(l);
```

Here we are calling a method with one parameter, and Java knows that it should create a lambda with one parameter and pass it to the method. The second line is equivalent to the first.

One question comes to mind here, we know the sort method is overloaded so how does Java know that we want to call the version that omits the comparator?

With both lambdas and method references, Java is inferring information from the context. In this case, we said that we were declaring a Consumer, which takes only one parameter. Java looks for a method that matches that description.

Specific Instance Method

Here is some example code calling an instance method on a specific instance:

```
String str = "Monitor";
Predicate<String> methodRef2 = str::startsWith;
Predicate<String> lambda2 = s -> str.startsWith(s);
```

The first line shows that we want to call string.startsWith() and pass a single parameter to be supplied at runtime. This would be a nice way of filtering the data in a list as we will see later.

Runtime Instance Method

In this example, we will call an instance method without knowing the instance in advance:

```
Predicate<String> methodRef3 = String::isEmpty;
Predicate<String> lambda3 = s -> s.isEmpty();
```

The first line says the method that we want to call is declared in String. It looks like a static method, but it isn't. Instead, Java knows that isEmpty is an instance method that does not take any parameters. Java uses the parameter supplied at runtime as the instance on which the method is called.

Constructor Method

The last type is a constructor reference:

```
Supplier<ArrayList> methodRef4 = ArrayList::new;
Supplier<ArrayList> lambda4 = () -> new ArrayList();
```

A *constructor reference* is a special type of method reference that uses new instead of a method, and it creates a new object. It expands like the method references you have seen so far. You'll see method references again in the next chapter when we cover more types of functional interfaces.

Java 8 Methods

There are a few new methods in Java 8 Collection classes that are really useful to know. Many of these work with Streams as well and so you will see them again later in the course.

Conditional Removal

Java 8 introduces a new method called `removeIf`. Before this, we had the ability to remove a specified object from a collection or a specified index from a list. Now we can specify what should be deleted using a block of code.

The method signature looks like this:

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

It uses a `Predicate`, which is a lambda that takes one parameter and returns a boolean. Since lambdas use deferred execution, this allows specifying logic to run when that point in the code is reached.

Here's an example:

```
List<String> list = new ArrayList<>();
list.add("PHX");
list.add("LHR");
list.add("LAX");
System.out.println(list); // [LHR, LAX]
list.removeIf(s -> s.endsWith("X"));
System.out.println(list); // [LHR]
```

The call to `removeIf()` shows how to remove all of the strings that end with the letter X. This allows us to make Los Angeles and Phoenix disappear!

There isn't much to `removeIf()` as long as long as you remember how `Predicate` works. The most important thing to remember about `removeIf()` is that it is one of two methods that are on a collection and it takes a lambda parameter.

Update All Elements

Another new method introduced on Lists is `replaceAll`. Java 8 lets you pass a lambda expression and have it applied to each element in the list. The result replaces the current value of that element.

The method signature looks like:

```
void replaceAll(UnaryOperator<E> o)
```

It uses a `UnaryOperator` (another functional interface) object, which takes one parameter and returns a value of the same type.

Here is an example:

```
List<Integer> list = Arrays.asList(1, 2, 3);
list.replaceAll(x -> x*2);
System.out.println(list); // [2, 4, 6]
```

The lambda uses deferred execution to increase the value of each element in the list.

Looping with Lambdas

Looping though a Collection is a very common requirement and so there is a new method provided to make this easier.

For example, we often want to print out the values one per line. There are a few ways you could do this. You could use an iterator, the enhanced for loop or there are several other approaches. Now there is a way you can do this with a lambda expression.

Cats like to explore, so let's join two of them as we learn a shorter way to loop through a Collection. We start with the traditional way:

```
List<String> movies = Arrays.asList("Legend", "Spectre", "Sicario",
                                    "Spy", "Room", "Pixels");
for(String movie: movies){
    System.out.println(movie);
}
```

This will work ok but you can do the same thing with lambdas in one line:

```
movies.forEach(c -> System.out.println(c));
```

This time, we've used a Consumer lambda object, which takes a single parameter and doesn't return anything. You won't see this approach used too often because it is common to use the equivalent method reference instead:

```
movies.forEach(System.out::println);
```

In the next chapter, you will learn about using the stream() method of collections to do powerful processing of large amounts of data with lambdas.

Java 8 Map API Methods

Java 8 added some new methods on the Map interface. The only one mentioned in the OCP Java Programmer II exam is merge() but there are two others that you will find useful.

These are: putIfPresent() and putIfAbsent().

Sometimes you need to update the value for a specific key in the map. There are a few ways that you can do this. The first is to replace the existing value unconditionally:

```
Map<String, String> movieStars = new HashMap<>();
movieStars.put("Legend", "Tom Hardy");
movieStars.put("Sicario", "Emily Blunt");
movieStars.put("Legend", "Emily Browning");
System.out.println(movieStars);
// {Sicario=Emily Blunt}, {Legend=Emily Browning}
```

There's a new method, called `putIfAbsent()`, that you can call if you want to set a value in the map, but this method skips it if the value is already set to a non-null value:

```
Map<String, String> movieStars = new HashMap<>();
movieStars.put("Legend", "Tom Hardy");
movieStars.put("Sicario", null);
movieStars.put("Legend", "Emily Browning");
movieStars.put("Sicario", "Emily Blunt");
System.out.println(movieStars);
// {Sicario=Emily Blunt}, {Legend=Tom Hardy}
```

This time Emily Browning fails to get in on the act as Tom Hardy is already the star of this film. Maybe we need a more flexible star system or there could be a few arguments as to who is the star (singular) of a film!

Emily Blunt rightly gets added as Sicario's star though as the value entry was null for this film.

merge

The `merge()` method allows adding logic to the problem of what to choose. Suppose that our studio is indecisive and can't pick a star for each film. They want to delegate the decision to a computer so no one gets the blame. The computer always decides that the person with the longest name wins.

We can write code to express this by passing a mapping function to the `merge()` method:

```
BiFunction<String, String, String> mapper =
    (v1, v2) -> v1.length() > v2.length() ? v1: v2;

Map<String, String> movieStars = new HashMap<>();
movieStars.put("Spy", "Melissa McCarthy");
movieStars.put("Burnt", "Bradley Cooper");

String spy = movieStars.merge("Spy", "Jude Law", mapper);
String burnt = movieStars.merge("Burnt", "Sienna Miller", mapper);

System.out.println(movieStars);
// {Spy=Melissa McCarthy, Burnt=Sienna Miller}
System.out.println(spy); // Jude Law
System.out.println(burnt); // Bradley Cooper
```

Look carefully! Sienna Miller beat Bradley Cooper even though both names are 14 characters long. That is because the formula used merges the second name unless the first is longer. Maybe we should have used `>=` but whichever we used, the stars will just argue anyway.

The same logic applies to null values (luckily or our lambda would be calling `length()` on a null and we'd get an exception. If the value is null the replacement is made without using the `merge()` at all.

Summary

Generics are type parameters for code. To create a class with a generic type parameter, add `<T>` after the class name. You can use any name you want for the type parameter. Single upper-case letters are common choices and there are some conventions (like `T` for type and `E` for element).

The diamond operator (`<>`) is used to tell Java that the generic type matches the declaration without specifying it again. The diamond operator can be used for local variables or instance variables as well as one-line declarations.

Generics allow wildcards: `<?>` is an unbounded wildcard that means any type. `<? extends Object>` is an upper bound that means any type that is `Object` or extends it. `<? extends MyInterface>` means any type that implements `MyInterface`. `<? super Number>` is a lower bound that means any type that is `Number` or a superclass. You cannot add or remove an item in a list with an unbounded or upper-bounded wildcard.

When working with code that doesn't use generics (also known as legacy code or raw types), Java gives a compiler warning. A compiler warning is different than a compiler error in that the compiler still produces a class file. If you ignore the compiler warning, the code might throw a `ClassCastException` at runtime. Unboxing gives a compiler error when generics are not used.

Each primitive class has a corresponding wrapper class. For example, `long`'s wrapper class is `Long`. Java can automatically convert between primitive and wrapper classes when needed. This is called autoboxing and unboxing. Java will only use autoboxing if it doesn't find a matching method signature with the primitive. For example, `remove(int n)` will be called rather than `remove(Object o)` when called with an `int`.

The Java Collections Framework includes four main types of data structures: lists, sets, queues, and maps. The `Collection` interface is the parent interface of `List`, `Set`, and `Queue`. The `Map` interface does not extend `Collection`. You need to recognise the following commonly used collections and their key aspects:

- **List**—An ordered collection of elements that allows duplicate entries
 - `ArrayList`—Standard resizable list.
 - `LinkedList`—Can easily add/remove from beginning or end.
 - `Vector`—Older, thread-safe version of `ArrayList`.
 - `Stack`—Older last-in, first-out class.
- **Set**—Does not allow duplicates
 - `HashSet`—Uses `hashcode()` to find unordered elements.
 - `TreeSet`—Sorted and navigable. Does not allow null values.
- **Queue**—Orders elements for processing
 - `LinkedList`—Can easily add/remove from beginning or end.
 - `ArrayDeque`—First-in, first-out or last-in, first-out. No null values.
- **Map**—Maps unique keys to values
 - `HashMap`—Uses `hashcode()` to find keys.
 - `TreeMap`—Sorted map. Does not allow null keys.
 - `Hashtable`—Older version of `HashMap`. Does not allow null keys or values.

The Comparable interface declares the `compareTo()` method. This method returns a negative number if the object is smaller than its argument, zero if the two objects are equal, and a positive number otherwise. `compareTo()` is declared on the object

that is being compared, and it takes one parameter. The Comparator interface defines the `compare()` method. A negative number is returned if the first argument is smaller, zero if they are equal, and a positive number otherwise. `compare()` can be declared in any code, and it takes two parameters. Comparator is often implemented using a lambda.

The Arrays and Collections classes have methods for `sort()` and `binarySearch()`. Both take an optional Comparator parameter. It is necessary to use the same sort order for both sorting and searching, so the result is not undefined. Collection has a few methods that take lambdas, including `removeIf()`, `forEach()`, and `merge()`.

A method reference is a compact syntax for writing lambdas that refer to methods. There are four types: static methods, instance methods referring to a specific instance, instance methods with the instance supplied at runtime, and constructor references.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 4

Functional Programming and Streams

Introduction

You have seen lambda expressions used in a number of situations and in the last chapter you saw how they can be replaced with method references when a single method call is involved.

You have also seen that functional interfaces are a powerful way to specify functionality. We have seen a few of these such as `Predicate<T>` that filters by running a test and returning true or false and `Function<T, R>` that transforms the input into some output.

Both lambdas and method references are used when implementing functional interfaces and act as implementation code that can be passed as an argument to a function like `Collections.sort()` to tell it how to behave.

At the end of the last chapter, you saw methods like `forEach()` and `merge()` working on collections to process the contents and provide output.

In this chapter, you will see how functional programming takes this one step further and works with the Streams API.

The Streams API was introduced in Java 8 specifically to allow for a functional programming approach to tasks that involve a lot of processing.

In this chapter, you will be introduced to many more functional interfaces and a group of `Optional` classes.

Finally, you will learn about the Stream pipeline which brings everything together.

Collections vs Streams

Collections are in-memory data structures which hold objects (elements) in a particular way depending on the collection type.

Each element in the collection is computed before it becomes a part of that collection. Streams on the other hand, are fixed data structures which compute the elements as required.

Streams can be seen as lazily constructed collections, where the values are computed when available or when requested by the caller. Collections contain eagerly computed values which are always present.

Collections are suitable for known amounts of data that will be processed together. Streams are suitable for unlimited and unknown amounts of data where processing can start as soon as some data is available rather than iterating over an entire collection.

Lambda Expression Variables

Earlier in the course you learned about the concept of ‘effectively final’ variables.”

These are local variables that can be accessed from an inner class because they don’t change. If you could add the final modifier to a local variable and the code would still compile then it’s effectively final.

Lambdas are anonymous local inner classes and so use the same access rules including allowing access to local variables only when they are final or effectively final.

Lambda expressions can access static variables, instance variables, effectively final method parameters, and effectively final local variables.

Here is some code to illustrate these different variable types:

```
interface Payable {
    String pay();
}

class Payments {
    String customer = "ACME Inc.";
    void MakePayments(boolean internal) {
        String supplier = "Nuts&Bolts";
        //supplier = "RawMat";
        showPayment(() -> customer);
        showPayment(() -> internal ? "Employee": "Contractor");
        showPayment(() -> supplier);
    }
    void showPayment(Payable p) {
        System.out.println(p.pay());
    }
}
```

The first call to showPayment() uses an instance variable (customer) in the lambda.

The second call uses a method parameter (internal). We know internal is effectively final since there are no reassessments to that variable.

The third and final call to showPayment() uses an effectively final local variable (supplier). If you were to uncomment the line that sets supplier = “RawMat” then supplier would no longer be effectively final. This would cause a compiler error when you try to access it to use it in the lambda.

The normal rules for access control still apply. For example, a lambda can’t access private variables in another class. Lambdas can access a subset of variables that are accessible, but never go beyond that.

Built-In Java Functional Interfaces

As you probably remember, a functional interface has exactly one abstract method. It can be annotated with `@FunctionalInterface` to emphasise this point and the compiler will check that no one has accidentally added a second abstract method.

All of the functional interfaces in the following table were introduced in Java 8 and are provided in the `java.util.function` package.

The convention here is to use the generic type T for type parameter. If a second type parameter is needed, the next letter, U, is used. If a distinct return type is needed, R for return is used for the generic type.

Functional Interface	Parameters	Return Type	Method
<code>Supplier<T></code>	None	T	get
<code>Consumer<T></code>	1 (T)	void	accept
<code>BiConsumer<T, U></code>	2 (T, U)	void	accept
<code>Predicate<T></code>	1 (T)	boolean	test
<code>BiPredicate<T, U></code>	2 (T, U)	boolean	test
<code>Function<T, R></code>	1 (T)	R	apply
<code>BiFunction<T, U, R></code>	2 (T, U)	R	apply
<code>UnaryOperator<T></code>	1 (T)	T	apply
<code>BinaryOperator<T></code>	2 (T, T)	T	apply

Many other functional interfaces are defined in the `java.util.function` package find you can see these by looking at the Javadocs Web page. Some are for working with primitives, and you'll see some of these later in the chapter.

Most of the time you don't need to assign the implementation of the interface to a variable or even specify the name of the functional interface. The interface name is implied, and it gets passed directly to the method that needs it.

It's important to know the names though so that you can better understand and remember what is going on.

You can name a functional interface anything you want. The only requirements are that it must be a valid interface name and contain a single abstract method.

Let's look at how to implement each of these interfaces. Since both lambdas and method references may be used, there is an implementation using both where possible.

Supplier

A Supplier is used when you want to generate or supply values without taking any input.

The Supplier interface is defined as

```
@FunctionalInterface
public interface Supplier<T> {
    public T get();
}
```

You may remember how to create a date using the now() factory method. You can use a Supplier to call this factory:

```
Supplier<LocalDate> s1 = LocalDate::now;
Supplier<LocalDate> s2 = () -> LocalDate.now();

LocalDate d1 = s1.get();
LocalDate d2 = s2.get();
System.out.println(d1);
System.out.println(d2);
```

This example prints a date such as 2017-09-26 twice. These lambdas are static method references.

The LocalDate::now method reference is used to create a Supplier to assign to an intermediate variable s1. A Supplier is often used in this way when constructing new objects.

Consumer and BiConsumer

You use a Consumer when you want to do something with a parameter but not return anything. BiConsumer does the same thing except that it takes two parameters. Omitting the default methods, the interfaces are defined as follows:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}
```

You used a Consumer in the last chapter with forEach. Here's the previous example being assigned to the Consumer interface:

```
Consumer<String> c1 = System.out::println;
Consumer<String> c2 = x -> System.out.println(x);
c1.accept("Monitor");
c2.accept("Desktop");
```

These two Consumer examples used a method reference to System.out::println which is an overloaded method with multiple signatures. Java uses the context of the lambda to determine which overloaded println() method it should call.

BiConsumer is called with two parameters. They don't have to be the same type. For example, we can put a key and a value in a map using this interface:

```
Map<String, Integer> map = new HashMap<>();
BiConsumer<String, Integer> b1 = map::put;
BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);
b1.accept("Basketball", 5);
b2.accept("Volleyball", 6);
System.out.println(map);
```

The output is {Basketball=5, Volleyball=6}, which shows that both BiConsumer implementations are called. This time we used an instance method reference since we want to call a method on the local variable map. It's also the first time that we passed two parameters to a method reference. The code to instantiate b1 is a good bit shorter than the code for b2. That's why method references are so popular.

Predicate and BiPredicate

Predicate is often used when filtering or matching which are both very common operations. A BiPredicate is just like a Predicate except that it takes two parameters instead of one. Omitting any default or static methods, the interfaces are defined as follows:

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

@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}
```

It should be old news by now that you can use a Predicate to test a condition:

```
Predicate<String> p1 = String::isEmpty;
Predicate<String> p2 = x -> x.isEmpty();
System.out.println(p1.test(""));
System.out.println(p2.test(""));
```

This prints true twice. More interesting is a BiPredicate. This example also prints true twice:

```
BiPredicate<String, String> b1 = String::endsWith;
BiPredicate<String, String> b2 =
        (string, prefix) -> string.endsWith(prefix);
System.out.println(b1.test("Volleyball", "ball"));
System.out.println(b2.test("Basketball", "ball"));
```

The method reference combines two techniques that you've already seen. `endsWith()` is an instance method. This means that the first parameter in the lambda is used as the instance on which to call the method. The second parameter is passed to the `endsWith()` method itself.

This is another example of how method references save a good bit of typing. However, they are less explicit, which means you really have to understand how they work.

Function and BiFunction

A Function is responsible for turning one parameter into a value of a potentially different type and returning it.

A BiFunction is responsible for turning two parameters into a value and returning it. Omitting any default or static methods, the interfaces are defined as the following:

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

@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

For example, this function converts a String to the length of the String:

```
Function<String, Integer> f1 = String::length;
Function<String, Integer> f2 = x -> x.length();
System.out.println(f1.apply("Lacrosse")); // 8
System.out.println(f2.apply("Tennis")); // 6
```

This function turns a String into an Integer. Well, technically it turns the String into an int, which is then autoboxed into an Integer. The types don't have to be different. The following combines two String objects and produces another String:

```
BiFunction<String, String, String> b1 = String::concat;
BiFunction<String, String, String> b2 =
        (string, toAdd) -> string.concat(toAdd);
System.out.println(b1.apply("Tennis ", "ace")); // Tennis ace
System.out.println(b2.apply("Lacrosse ", "star")); // Lacrosse star
```

The first two types in the BiFunction are the input types. The third is the result type. For the method reference, the first parameter is the instance that `concat()` is called on and the second is passed to `concat()`.

UnaryOperator and BinaryOperator

`UnaryOperator` and `BinaryOperator` are a special case of `Function`. They both require all type parameters to be the same type.

A UnaryOperator transforms its value into one of the same type. For example, incrementing by one is a unary operation. In fact, UnaryOperator extends Function.

A BinaryOperator merges two values into one of the same type. Adding two numbers is a binary operation. Similarly, BinaryOperator extends BiFunction. Omitting any default or static methods, the interfaces are defined as follows:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {}
```

They look like they don't have any abstract methods and so can't be functional interfaces. However, if you look at the Javadoc pages, you'll see that the apply() methods are actually declared on the Function and BiFunction superclasses respectively.

The methods become:

```
T apply(T t);
T apply(T t1, T t2);
```

The generic declarations on the sub interfaces are what force the type to be the same. For the unary example, notice how the return type is the same type as the parameter:

```
UnaryOperator<String> u1 = String::toUpperCase;
UnaryOperator<String> u2 = x -> x.toUpperCase();
System.out.println(u1.apply("London"));
System.out.println(u2.apply("London"));
```

This prints LONDON twice. We don't need to specify the return type in the generics because UnaryOperator requires it to be the same as the parameter. And here's a binary example:

```
BinaryOperator<String> b1 = String::concat;
BinaryOperator<String> b2 = (string, toAdd) -> string.concat(toAdd);
System.out.println(b1.apply("Tennis ", "ace")); // Tennis ace
System.out.println(b2.apply("Lacrosse ", "star")); // Lacrosse star
```

Notice that this does the same thing as the BiFunction example. The code is more succinct, which shows the importance of using the correct functional interface. It's nice to have one generic type specified instead of three.

Optional Objects

There are many situations when you don't know some data but are being asked to give a derived value. The only correct answer is 'unknown' but what does that mean in Java terms? If it's a number, it is probably initialised to zero or undefined if local in which case an exception will be thrown if it's used in a calculation. The same goes for String which is probably null.

Suppose that you are calculating quarterly profit and have the results for the first two months.

One of your monthly report slides has the quarterly average. If you put zero for the third month's profit you will get fired as it makes the figures look very bad.

Java 8 provides a way to store unknown values in the Optional type. An Optional is created using a factory. You can either request an empty Optional or pass a value for the Optional to wrap. Think of an Optional as a box that might have something in it or might instead be empty.

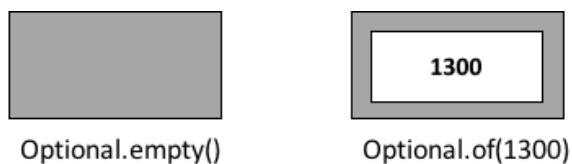


Figure 4-1 Optional Factory Methods

Here's how to code the average method:

```
public static Optional<Double> average(int... scores) {
    if (scores.length < 3) {
        return Optional.empty();
    }
    int sum = 0;
    for (int score: scores) {
        sum += score;
    }
    return Optional.of((double) sum / scores.length);
}
```

You should return an empty Optional with a call to the `Optional.empty()` factory method if you cannot calculate an average.

The `Optional.of()` factory method call creates an Optional to wrap the average once it has been calculated.

Calling the method shows what is in the Optional 'box':

```
System.out.println(average(9050, 11070, 8920)); // Optional [9680.0]
System.out.println(average(9050, 11070)); // Optional.empty
System.out.println(average()); // Optional.empty
```

Normally, you will want to check if a value is there and/or get it out of the box. Here's one way to do that:

```
Optional<Double> opt = average(90, 100);
if (opt.isPresent()){
    System.out.println(opt.get()); // 95.0
}
```

What if you didn't check first and the Optional turned out to be empty?

```
Optional<Double> opt = average();
System.out.println(opt.get()); // bad
```

This produces an exception since there is no value inside the Optional:

```
java.util.NoSuchElementException: No value present
```

When creating an Optional, it is common to want to use empty when the value is null. You can do this with an if statement or ternary operator:

```
Optional o = (value == null) ? Optional.empty(): Optional.of(value);
```

If value is null, o is assigned the empty Optional. Otherwise, we wrap the value. Since this is such a common pattern, Java provides a factory method to do the same thing:

```
Optional o = Optional.ofNullable(value);
```

That covers the static methods you need to know about Optional.

This table shows most of the useful instance methods on Optional:

Method	If Empty	If not Empty
get()	Throws an exception	Returns value
ifPresent(Consumer c)	Does nothing	Calls c.accept(value)
isPresent()	Returns false	Returns true
orElse(T other)	Returns other	Returns value
orElseGet(Supplier s)	Returns s.get()	Returns value
orElseThrow(Supplier s)	Throws exception from s.get()	Returns value

Streams

A stream in Java is a sequence of data. A stream pipeline is a set of operations that run on a stream to produce a result. Think of a stream pipeline as an assembly line in a factory.

The first person (or robot) spends their shift bolting two items together. The next person sprays the resulting product red. The next person checks the quality and rejects any that are no good. The next person puts it into a box. The last person puts the boxes on a pallet and they are driven away to the customer.

Each stage relies on the last and cannot begin until the previous one has been completed. However, person 2 does not have to wait until person 1 has bolted all the items in their pile together. As soon as one item is bolted by person 1, person 2 can start spraying it. All the stages can be going at the same time.

Another important feature of an assembly line is that each person touches each element to do their operation and then that piece of data is gone. It doesn't come back. The next person deals with it at that point. This is different than the lists and queues that you saw in the last chapter. With a list, you can access any element at any time. With a queue, you are limited in which elements you can access, but all of the elements are there. With streams, the data isn't generated up front—it is created when needed.

Many things can happen in the assembly line stations along the way. In programming, these are called stream operations. Just like with the assembly line, operations occur in a pipeline. Someone has to start and end the work, and there can be any number of stations in between.

There are three parts to a stream pipeline, as shown here:

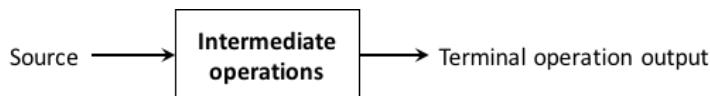


Figure 4-2 Stream Pipeline

These three parts are:

- Source: Where the stream comes from.
- Intermediate operations: Transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
- Terminal operation: Produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes (so you can only have one of these).

Stream Source

In Java, the Stream interface is in the `java.util.stream` package. There are a few ways to create a finite stream:

```
Stream<String> empty = Stream.empty(); // count = 0
Stream<Integer> singleElement = Stream.of(1); // count = 1
Stream<Integer> fromArray = Stream.of(1, 2, 3); // count = 2
```

The first line creates an empty stream. The second line shows how to create a stream with a single element. The third line creates a stream from an array. The method signature uses varargs, which lets you pass in a comma-separated list.

Since you often have data in collection such as a list, Java provides a convenient way to convert from a list to a stream:

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> fromList = list.stream();
Stream<String> fromListParallel = list.parallelStream();
```

The `stream()` method is a simple way to create a stream from a list.

The following line does the same thing, except that it creates a stream that is allowed to process elements in parallel.

This is a great feature because you can write code that uses parallelism before even learning what a thread is. Using parallel streams is like setting up multiple stations of workers who are able to do the same task. Bolting and spraying would be a lot faster if we could have six people at each station. There's an overhead cost involved in parallel streams so it isn't usually worth working in parallel for small streams.

You could do all this with lists but there is a difference, you can't create an infinite list like you can with streams, which makes streams more powerful:

```
Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

The first line generates a stream of random numbers. How many random numbers? However many you need. If you call `randoms.forEach(System.out::println)`, the program will print random numbers until you kill it. Later in the chapter, you'll learn about operations like `limit()` to turn the infinite stream into a finite stream.

The second line gives more control in terms of the result produced. The `iterate()` method takes a seed or starting value as the first parameter. This is the first element of the stream. The other parameter is a lambda expression (for Function) that gets passed the previous value and generates the next value. This will keep on producing odd numbers as long as you need it to.

Terminal Operations

You can perform a terminal operation without any intermediate operations but not the other way around. This is why we will talk about terminal operations first. Reductions are a special type of terminal operation where all of the contents of the stream are combined into a single primitive or Object. For example, you might have an int or a Collection.

The following table summarises the methods you can use:

Method(s)	Terminates for Infinite Streams	Return value	Reduction
allMatch() anyMatch() noneMatch()	Sometimes	boolean	No
collect()	Never	Varies	Yes
count()	Never	long	Yes
findAny() findFirst()	Always	Optional<T>	No
forEach()	Never	void	No
min() max()	Never	Optional<T>	Yes
reduce()	Never	Varies	Yes

count()

The count() method determines the number of elements in a finite stream. For an infinite stream, it hangs as it would need to count for ever.

The count() method is a reduction because it looks at each element in the stream and returns a single value. The method signature is this:

```
long count()
```

This example shows calling count() on a finite stream:

```
Stream<String> s = Stream.of("uno", "dos", "tres", "cuatro");
System.out.println(s.count()); // 4
```

min() and max()

The min() and max() methods allow you to pass a custom comparator and find the smallest or largest value in a finite stream according to that sort order. Like count(), min() and max() hang on an infinite stream because they cannot be sure that a smaller or larger value isn't coming later in the stream. Both methods are reductions because they return a single value after looking at the entire stream. The method signatures are as follows:

```
Optional<T> min(<? super T> comparator)
Optional<T> max(<? super T> comparator)
```

This example finds the number with the fewest letters in its name:

```
Stream<String> s = Stream.of("uno", "dos", "tres", "cuatro");
Optional<String> min = s.min((s1, s2) -> s1.length() - s2.length());
min.ifPresent(System.out::println); // uno
```

Notice that the code returns an Optional rather than the value. This allows the method to specify that no minimum or maximum was found. We use the Optional method and a method reference to print out the minimum only if one is found. As an example of where there isn't a minimum, let's look at an empty stream:

```
Optional<?> minEmpty = Stream.empty().min((s1, s2) -> 0);
System.out.println(minEmpty.isPresent()); // false
```

Since the stream is empty, the comparator is never called and no value is present in the Optional.

findAny() and findFirst()

The findAny() and findFirst() methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty Optional. This is the first method you've seen that works with an infinite stream. Since Java generates only the amount of stream you need, the infinite stream needs to generate only one element. findAny() is useful when you are working with a parallel stream. It gives Java the flexibility to return to you the first element it comes by rather than the one that needs to be first in the stream based on the intermediate operations.

These methods are terminal operations but not reductions. The reason is that they sometimes return without processing all of the elements. This means that they return a value based on the stream but do not reduce the entire stream into one value.

The method signatures are these:

```
Optional<T> findAny()
Optional<T> findFirst()
```

This example finds a letter:

```
Stream<String> finite = Stream.of("a", "b", "c");
Stream<String> infinite = Stream.generate(() -> "x");
s.findAny().ifPresent(System.out::println); // a
infinite.findAny().ifPresent(System.out::println); // x
```

Finding any one match is more useful than it sounds. Sometimes we just want to sample the results and get a representative element, but we don't need to waste the processing generating them all. After all, if you only need one element, why bother looking at more?

allMatch(), anyMatch() and noneMatch()

These three methods search a stream and return information about how the stream relates to the predicate. These may or may not terminate for infinite streams. It depends on the data. Like the find methods, they are not reductions because they do not necessarily look at all of the elements.

The method signatures are as follows:

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

This example checks whether aircraft engine names begin with letters:

```
List<String> list = Arrays.asList("7E", "CF6", "H75");
Stream<String> infinite = Stream.generate(() -> "X-250");
Predicate<String> pred = x -> Character.isLetter(x.charAt(0));
System.out.println(list.stream().anyMatch(pred)); // true
System.out.println(list.stream().allMatch(pred)); // false
System.out.println(list.stream().noneMatch(pred)); // false
System.out.println(infinite.anyMatch(pred)); // true
```

This shows that we can reuse the same predicate, but we need a different stream each time. `anyMatch()` returns true because two of the three elements match. `allMatch()` returns false because one doesn't match. `noneMatch()` also returns false because one matches. On the infinite list, one match is found, so the call terminates.

If we called `noneMatch()` or `allMatch()`, they would run until we killed the program.

forEach()

A looping construct is available. As expected, calling `forEach()` on an infinite stream does not terminate. Since there is no return value, it is not a reduction.

Before you use it, consider if another approach would be better. Developers who learned to write loops first tend to use them for everything. For example, a loop with an `if` statement should be a filter instead.

The method signature is the following:

```
void forEach(Consumer<? super T> action)
```

Notice that this is the only terminal operation with a return type of `void`. If you want something to happen, you have to make it happen in the loop. Here's one way to print the elements in the stream. There are other ways, which you will see later in the chapter.

```
Stream<String> s = Stream.of("Funk", "Heinkel", "Sauer");
s.forEach(System.out::print); // FunkHeinkelSauer
```

Remember that you can call `forEach()` directly on a Collection or on a Stream but you can't use a traditional for loop on a stream:

```
Stream s = Stream.of(1);
for (Integer i: s) {} // DOES NOT COMPILE
```

Although `forEach()` sounds like a loop, it is really a terminal operator for streams. Streams cannot use a traditional for loop to run because they don't implement the Iterable interface.

reduce()

The `reduce()` method combines a stream into a single object. As you can tell from the name, it is a reduction. The method signatures are these:

```
T reduce(T identity, BiFunction<T, T, T> accumulator)
Optional<T> reduce(BiFunction<T, T, T> accumulator)
<U> U reduce(U identity, BiFunction<T, ? super T, U>
               accumulator, BiFunction<U, U> combiner)
```

Let's take them one at a time. The most common way of doing a reduction is to start with an initial value and keep merging it with the next value. Think about how you would concatenate an array of Strings into a single String without functional programming. It might look something like this:

```
String[] array = new String[] { "a", "b", "c", "d" };
String result = "";
for (String s: array){
    result = result + s; System.out.println(result);
}
```

The initial value of an empty String is the identity. The accumulator combines the current result with the current String. With lambdas, we can do the same thing with a stream and reduction:

```
Stream<String> stream = Stream.of("a", "b", "c", "d");
String word = stream.reduce("", (s, c) -> s + c);
System.out.println(word); // abcd
```

Notice how we still have the empty String as the identity. We also still concatenate the Strings to get the next value. We can even rewrite this with a method reference:

```
Stream<String> stream = Stream.of("a", "b", "c", "d"); String word =
stream.reduce("", String::concat); System.out.println(word); // abcd
```

Let's try another one. Can you write a reduction to multiply all of the Integer objects in a stream? Think about it. One solution is shown here:

```
Stream<Integer> stream = Stream.of(3, 5, 6);
System.out.println(stream.reduce(1, (a, b) -> a*b));
```

We set the identity to 1 and the accumulator to multiplication. In many cases, the identity isn't really necessary, so Java lets us omit it.

When you don't specify an identity, an Optional is returned because there might not be any data. There are three choices for what is in the Optional:

- If the stream is empty, an empty Optional is returned.
- If the stream has one element, it is returned.
- If the stream has multiple elements, the accumulator is applied to combine them.

The following illustrates each of these scenarios:

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> empty = Stream.empty();
Stream<Integer> oneElement = Stream.of(3);
Stream<Integer> threeElements = Stream.of(3, 5, 6);
empty.reduce(op).ifPresent(System.out::print); // no output
oneElement.reduce(op).ifPresent(System.out::print); // 3
threeElements.reduce(op).ifPresent(System.out::print); // 90
```

Why are there two similar methods? Why not just always require the identity? Java could have done that.

However, sometimes it is nice to differentiate the case where the stream is empty rather than the case where there is a value that happens to match the identity being returned from calculation.

The signature returning an Optional lets us differentiate these cases. For example, we might return `Optional.empty()` when the stream is empty and `Optional.of(3)` when there is a value.

The third method signature is used when we are processing collections in parallel. It allows Java to create intermediate reductions and then combine them at the end. In our example, it looks similar. While we aren't actually using a parallel stream here, Java assumes that a stream might be parallel. This is helpful because it lets us switch to a parallel stream easily in the future:

```
BinaryOperator<Integer> op = (a, b) -> a * b;
Stream<Integer> stream = Stream.of(3, 5, 6);
System.out.println(stream.reduce(1, op, op)); // 90
```

collect()

The `collect()` method is a special type of reduction called a mutable reduction. It is more efficient than a regular reduction because we use the same mutable object while accumulating.

Common mutable objects include `StringBuilder` and `ArrayList`. This is a really useful method, because it lets us get data out of streams and into another form.

The method signatures are as follows:

```
<R> R collect(Supplier<R> supplier,
                BiConsumer<R, ? super T> accumulator,
                BiConsumer<R, R> combiner)
<R, A> R collect(Collector<? super T, A, R> collector)
```

Let's start with the first signature, which is used when we want to code specifically how collecting should work. Our abcd example from reduce can be converted to use collect():

```
Stream<String> stream = Stream.of("a", "b", "c", "d");
StringBuilder word = stream.collect(StringBuilder::new,
                                    StringBuilder::append, StringBuilder::append)
```

The first parameter is a Supplier that creates the object that will store the results as we collect data. Remember that a Supplier doesn't take any parameters and returns a value. In this case, it constructs a new StringBuilder.

The second parameter is a BiConsumer, which takes two parameters and doesn't return anything. It is responsible for adding one more element to the data collection. In this example, it appends the next String to the StringBuilder.

The final parameter is another BiConsumer. It is responsible for taking two data collections and merging them. This is useful when we are processing in parallel. Two smaller collections are formed and then merged into one. This would work with StringBuilder only if we didn't care about the order of the letters. In this case, the accumulator and combiner have similar logic.

Now let's look at an example where the logic is different in the accumulator and combiner:

```
Stream<String> stream = Stream.of("a", "c", "b", "d");
TreeSet<String> set = stream.collect(TreeSet::new,
                                       TreeSet::add, TreeSet::addAll);
System.out.println(set); // [a, b, c, d]
```

The collector has three parts as before. The supplier creates an empty TreeSet. The accumulator adds a single String from the Stream to the TreeSet.

The combiner adds all of the elements of one TreeSet to another in case the operations were done in parallel and need to be merged.

We started with the long signature because that's how you implement your own collector.

In practice, there are many common collectors that come up over and over. Rather than making developers keep reimplementing the same ones, Java provides an interface with common collectors. This approach also makes the code easier to read because it is more expressive. For example, we could rewrite the previous example as follows:

```
Stream<String> stream = Stream.of("a", "c", "b", "d");
TreeSet<String> set =
    stream.collect(Collectors.toCollection(TreeSet::new));
System.out.println(set); // [a, b, c, d]
```

If we didn't need the set to be sorted, we could make the code even shorter:

```
Stream<String> stream = Stream.of("a", "c", "b", "d");
Set<String> set = stream.collect(Collectors.toSet());
System.out.println(set); // [d, c, a, b]
```

You might get different output for this last one since `toSet()` makes no guarantees as to which implementation of Set you'll get. It is likely to be a HashSet, but you shouldn't expect or rely on that.

Intermediate Operations

Unlike a terminal operation, intermediate operations deal with infinite streams simply by returning an infinite stream. Since elements are produced only as needed, this works fine. The assembly line worker doesn't need to worry about how many more elements are coming through and instead can focus on the current element.

`filter()`

The `filter()` method returns a Stream with elements that match a given expression. Here is the method signature:

```
Stream<T> filter(Predicate<? super T> predicate)
```

This operation is easy to remember and very powerful because we can pass any Predicate to it. For example, this filters all elements that contain the letter X:

```
Stream<String> s = Stream.of("P. XI", "J-16", "S 2500 L");
s.filter(x -> x.contains("X")).forEach(System.out::print); // P. XI
```

`distinct()`

The `distinct()` method returns a stream with duplicate values removed. The duplicates do not need to be adjacent to be removed. As you might imagine, Java calls `equals()` to determine whether the objects are the same. The method signature is as follows:

```
Stream<T> distinct()
```

Here's an example:

```
Stream<String> s = Stream.of("A", "B", "C", "B");
s.distinct().forEach(System.out::print); // ABC
```

`limit()` and `skip()`

The `limit()` and `skip()` methods make a Stream smaller. They could make a finite stream smaller, or they could make a finite stream out of an infinite stream.

The method signatures are shown here:

```
Stream<T> limit(int maxSize)
Stream<T> skip(int n)
```

The following code creates an infinite stream of numbers counting from 1.

The `skip()` operation returns an infinite stream starting with the numbers counting from 6, since it skips the first five elements. The `limit()` call takes the first two of those. Now we have a finite stream with two elements:

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
```

```
s.skip(5).limit(2).forEach(System.out::print); // 67
```

map()

The `map()` method creates a one-to-one mapping from the elements in the stream to the elements of the next step in the stream. The method signature is as follows:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

This one looks more complicated than the others you have seen. It uses the lambda expression to figure out the type passed to that function and the one returned. The return type is the stream that gets returned.

As an example, this code converts a list of `String` objects to a list of `Integers` representing their lengths:

```
Stream<String> s = Stream.of("Zhuzhou", "Quick", "Gleshenkov");
s.map(String::length).forEach(System.out::print); // 7510
```

Remember that `String::length` is shorthand for the lambda `x -> x.length()`, which clearly shows it is a function that turns a `String` into an `Integer`.

flatMap()

The `flatMap()` method takes each element in the stream and makes any elements it contains top-level elements in a single stream.

This is helpful when you want to remove empty elements from a stream or you want to combine a stream of lists.

Here is the method signature (for consistency with the other methods) but don't worry if it doesn't make too much sense:

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

This basically says that it returns a `Stream` of the type that the function contains at a lower level. Don't worry about the signature. It's a bit complicated.

What you should understand is how it works. This gets all of the `Aircraft` into the same level along with getting rid of the empty list:

```
List<String> zero = Arrays.asList();
List<String> one = Arrays.asList("ThruXton");
List<String> two = Arrays.asList("Wallis", "Gadfly");
Stream<List<String>> aircraft = Stream.of(zero, one, two);
aircraft.flatMap(l -> l.stream()).forEach(System.out::print);
```

Here's the output:

```
ThruXtonWallisGadfly
```

As you can see, it removed the empty list completely and changed all elements of each list to be at the top level of the stream.

sorted()

The sorted() method returns a stream with the elements sorted. Just like sorting arrays, Java uses natural ordering unless we specify a comparator.

The method signatures are these:

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

Calling the first signature uses the default sort order:

```
Stream<String> s = Stream.of("VTOL:", "STOL:");
s.sorted().forEach(System.out::print); // STOL: VTOL:
```

Remember that we can pass a lambda expression as the comparator.

For example, we can pass a Comparator implementation:

```
Stream<String> s = Stream.of("VTOL:", "CTOL:", "STOL:");
s.sorted(Comparator.reverseOrder())
    .forEach(System.out::print); // VTOL: STOL: CTOL:
```

Here we passed a Comparator to specify that we want to sort in the reverse of natural sort order.

peek()

The peek() method is the final intermediate operation in this section. It is useful for debugging because it performs a stream operation without actually changing the stream.

The method signature is as follows:

```
Stream<T> peek(Consumer<? super T> action)
```

The most common use for peek() is to output the contents of the stream as it goes by. Suppose that some erroneous code counts Aircraft containing the number 4 instead of 1.

We are puzzled why the code doesn't do what we want but with many intermediate operations we are not sure where the error is. By adding a peek() the problem is exposed to view:

```
Stream<String> stream =
    Stream.of("A380", "747", "C-5", "MD-11", "Tu-154M");
long count = stream.filter(s -> s.contains("4"))
    .peek(System.out::println).count(); // 747
System.out.println(count); // 1
```

Constructing a Stream Pipeline

Streams allow you to use chaining and express what you want to accomplish rather than how to do so. Let's say that we wanted to get the first two numbers alphabetically that are four characters long.

In Java 7, we'd have to write something like the following:

```
List<String> list =
    Arrays.asList("uno", "dos", "tres", "cuatro", "cinco", "seis");
List<String> filtered = new ArrayList<>();
for (String name: list) {
    if (name.length() == 4) {
        filtered.add(name);
    }
}
Collections.sort(filtered);
Iterator<String> iter = filtered.iterator();
if (iter.hasNext()) System.out.println(iter.next());
if (iter.hasNext()) System.out.println(iter.next());
```

This works. It takes some reading and thinking to figure out what is going on.

The problem we are trying to solve gets lost in the implementation. It is also very focused on the how rather than on the what. In Java 8, the equivalent code is as follows:

```
List<String> list =
    Arrays.asList("uno", "dos", "tres", "cuatro", "cinco", "seis");
list.stream().filter(n -> n.length() == 4).sorted()
    .limit(2).forEach(System.out::println);
```

Before you say that it is harder to read, we can format it:

```
stream.filter(n -> n.length() == 4)
    .sorted()
    .limit(2)
    .forEach(System.out::println);
```

This code expresses what the code is doing:

- We care about String objects of length 4.
- Then we then want them sorted.
- Then we want to find the first two.
- Then we want to print them out.

It maps better to the problem that we are trying to solve, and it is simpler because we don't have to deal with counters and such.

Once you start using streams in your code, you may find yourself using them in many places. Having shorter, briefer, and clearer code is definitely a real benefit for you and your colleagues.

In this example, you see all three parts of the pipeline.

This diagram shows how each intermediate operation in the pipeline feeds into the next.

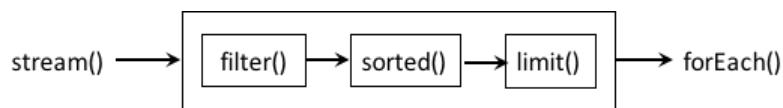


Figure 4-4 Stream Pipeline with Intermediate Operations

Primitive Streams

Until now, we have been using wrapper classes when we needed primitives to go into streams. We did this with the Collections API so it would feel natural.

With streams, there are also equivalents that work with the int, double, and long primitives.

First let's take a look at why this is needed. Suppose that we want to calculate the sum of numbers in a finite stream:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
System.out.println(stream.reduce(0, (s, n) -> s + n));
```

Not bad. It wasn't hard to write a reduction. We started the accumulator with zero. We then added each number to that running total as it came up in the stream.

There is another way of doing that:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
System.out.println(stream.mapToInt(x -> x).sum());
```

This time, we converted our `Stream<Integer>` to an `IntStream` and asked the `IntStream` to calculate the sum for us. The primitive streams know how to perform certain common operations automatically.

So far, this seems like a nice convenience but not terribly important. Now think about how you would compute an average. You need to divide the sum by the number of elements. The problem is that streams allow only one pass. Java recognizes that calculating an average is a common thing to do, and it provides a method to calculate the average on the stream classes for primitives:

```
IntStream intStream = IntStream.of(1, 2, 3);
OptionalDouble avg = intStream.average();
System.out.println(avg.getAsDouble());
```

Not only is it possible to calculate the average, but it is also easy to do so.

Clearly primitive streams are important. We will look at creating and using such streams, including optionals and functional interfaces.

Creating Primitive Streams

There are three types of primitive streams:

- `IntStream`: Used for the primitive types int, short, byte, and char
- `LongStream`: Used for the primitive type long
- `DoubleStream`: Used for the primitive types double and float

Why doesn't each primitive type have its own primitive stream? These three are the most common, so the API designers went with them.

Some of the methods for creating a primitive stream are equivalent to how we created the source for a regular Stream. You can create an empty stream with this:

```
DoubleStream empty = DoubleStream.empty();
```

Another way is to use the of() factory method from a single value or by using the varargs overload:

```
DoubleStream oneValue = DoubleStream.of(3.14);
DoubleStream varargs = DoubleStream.of(1.0, 1.1, 1.2);
oneValue.forEach(System.out::println);
System.out.println();
varargs.forEach(System.out::println);
```

This code outputs the following:

3.14

1.0
1.1
1.2

It works the same way for each type of primitive stream. You can also use the two methods for creating infinite streams, just like we did with Stream:

```
DoubleStream random = DoubleStream.generate(Math::random);
DoubleStream fractions = DoubleStream.iterate(.5, d -> d / 2);
random.limit(3).forEach(System.out::println);
System.out.println();
fractions.limit(3).forEach(System.out::println);
```

Since the streams are infinite, we added a limit intermediate operation so that the output doesn't print values forever. The first stream calls a static method on Math to get a random double.

Since the numbers are random, your output will obviously be different. The second stream keeps creating smaller numbers, dividing the previous value by two each time. The output from when we ran this code was as follows:

0.85643634265842346
0.08118641378906547
0.5112661346311403
0.5
0.25
0.125

The Random class provides a method to get primitives streams of random numbers directly. For example, ints() generates an infinite stream of int primitives.

When dealing with int or long primitives, it is common to count. Suppose that we wanted a stream with the numbers from 1 through 5. We could write this using what you know so far:

```
IntStream count = IntStream.iterate(1, n -> n+1).limit(5);
count.forEach(System.out::println);
```

This code does print out the numbers 1–5, one per line. However, it is a lot of code to do something so simple. Java provides a method that can generate a range of numbers:

```
IntStream range = IntStream.range(1, 6);
range.forEach(System.out::println);
```

This is better. The `range()` method indicates that we want the numbers 1–6, not including the number 6. However, it still could be clearer. We want the numbers 1–5. We should be able to type the number 5, and we can do so as follows:

```
IntStream rangeClosed = IntStream.rangeClosed(1, 5);
rangeClosed.forEach(System.out::println);
```

Even better. This time we expressed that we want a closed range, or an inclusive range. This method better matches how we express a range of numbers in plain English.

The final way to create a primitive stream is by mapping from another stream type.

This table shows that there is a method for mapping between any stream types:

To Create:				
From:	Stream	DoubleStream	IntStream	LongStream
Stream	map	mapToDouble	mapToInt	mapToLong
DoubleStream	mapToObj	map	mapToInt	mapToLong
IntStream	mapToObj	mapToDouble	map	mapToLong
LongStream	mapToObj	mapToDouble	mapToInt	map

Optional and Primitive Streams

Earlier you saw a method to calculate the average of an int array. With primitive streams, you can calculate the average in one line:

```
IntStream stream = IntStream.rangeClosed(1, 10);
OptionalDouble optional = stream.average();
```

The return type is not the `Optional` you have become accustomed to using. It is a new type called `OptionalDouble`. Why do we have a separate type, you might wonder? Why not just use `Optional<Double>`?

The difference is that `OptionalDouble` is for a primitive and `Optional<Double>` is for the `Double` wrapper class.

Working with the primitive optional class looks similar to working with the `Optional` class itself:

```
optional.ifPresent(System.out::println);
System.out.println(optional.getAsDouble());
System.out.println(optional.orElseGet(() -> Double.NaN));
```

The only noticeable difference is that we called `getAsDouble()` rather than `get()`.

This makes it clear that we are working with a primitive.

Another difference is that `orElseGet()` takes a `DoubleSupplier` instead of a `Supplier`.

As with the primitive streams, there are three type-specific classes for primitives.

This table shows the minor differences among the three:

	OptionalDouble	OptionalInt	OptionalLong
Get as primitive	<code>getAsDouble()</code>	<code>getAsInt()</code>	<code>getAsLong()</code>
<code>orElseGet()</code> parameter	<code>DoubleSupplier</code>	<code>IntSupplier</code>	<code>LongSupplier</code>
<code>max()</code> returns	<code>OptionalDouble</code>	<code>OptionalInt</code>	<code>OptionalLong</code>
<code>sum()</code> returns	<code>double</code>	<code>int</code>	<code>long</code>
<code>avg()</code> returns	<code>OptionalDouble</code>	<code>OptionalDouble</code>	<code>OptionalDouble</code>

You may remember from the terminal operations section that a number of stream methods return an optional such as `min()` or `findAny()`.

These each return the corresponding optional type. The primitive stream implementations also add two new methods that you need to know.

The `sum()` method does not return an optional. If you try to add up an empty stream, you simply get zero.

The `avg()` method always returns an `OptionalDouble`, since an average can potentially have fractional data for any type.

Summary Statistics

You've learned enough to be able to get the maximum value from a stream of int primitives. If the stream is empty, we want to throw an exception:

```
private static int max(IntStream ints) {
    OptionalInt optional = ints.max();
    return optional.orElseThrow(NullPointerException::new);
}
```

This should be familiar by now. We got an `OptionalInt` because we have an `IntStream`. If the optional contains a value, we return it. Otherwise, we throw a new `RuntimeException`.

Now we want to change the method to take an `IntStream` and return a range. The range is the minimum value subtracted from the maximum value. However, both `min()` and `max()` are terminal operations, which means that they use up the stream when they are run.

You can't run two terminal operations against the same stream. Luckily, this is a common problem and the primitive streams solve it for us with summary statistics. Statistic is just a term for a number that was calculated from data.

```
private static int range(IntStream ints) {  
    IntSummaryStatistics stats = ints.summaryStatistics();  
    if (stats.getCount() == 0){  
        throw new RuntimeException();  
    }  
    return stats.getMax() - stats.getMin();  
}
```

Here we asked Java to perform many calculations about the stream. This includes the minimum, maximum, average, size, and the number of values in the stream. If the stream were empty, we'd have a count of zero. Otherwise, we can get the minimum and maximum out of the summary.

Functional Interfaces for Primitives

Just as there are special streams and optional classes for primitives, there are also special functional interfaces.

Luckily, most of them are for the double, int, and long types that you saw for streams and optionals. There is one exception, which is BooleanSupplier. We will cover that before introducing the ones for double, int, and long.

boolean Functional Interfaces

BooleanSupplier is a separate type. It has one method to implement:

```
bool ean getAsBool ean()
```

It works just as you've come to expect from functional interfaces, for example:

```
Bool eanSuppl i er b1 = () -> true;
Bool eanSuppl i er b2 = () -> Math. random() > .5;
System. out. printl n(b1. getAsBool ean());
System. out. printl n(b2. getAsBool ean());
```

The first two lines each create a BooleanSupplier, which is the only functional interface for boolean. The next line prints true, since it is the result of b1. The last line may print out true or false, depending on the random value generated.

double, int, and long Functional Interfaces

Most of the functional interfaces are for double, int, and long to match the streams and optionals that we've been using for primitives.

This table summarises the operation of these primitives:

Functional Interfaces	Parameters	Return	Method
DoubleSupplier IntSupplier LongSupplier	0	double int long	getAsDouble getAsInt getAsLong
DoubleConsumer IntConsumer LongConsumer	1	void	accept
DoublePredicate IntPredicate LongPredicate	1	boolean	test
DoubleFunction<R> IntFunction<R> LongFunction<R>	1	R	apply

DoubleUnaryOperator IntUnaryOperator LongUnaryOperator	1	double int long	applyAsDouble applyAsInt applyAsLong
DoubleBinaryOperator IntBinaryOperator LongBinaryOperator	2	double int long	applyAsDouble applyAsInt applyAsLong

There are a few things to notice that are different between This table and the non-primitive version you saw earlier:

- Generics are gone from some of the interfaces, since the type name tells us what primitive type is involved. In other cases, such as IntFunction, only the return type generic is needed.
- The single abstract method is often, but not always, renamed to reflect the primitive type involved.
- There are no primitive equivalents for BiConsumer, BiPredicate, and BiFunction. The API designers stuck to the most common operations. For primitives, the functions with two different type parameters just aren't used very often.

Some interfaces are specific to primitives.

This table lists these.

Functional Interfaces	Parameters	Return	Method
ToDoubleFunction<T> ToIntFunction<T> ToLongFunction<T>	1	double int long	applyAsDouble applyAsInt applyAsLong
ToDoubleBiFunction<T, U> ToIntBiFunction<T, U> ToLongBiFunction<T, U>	2	double int long	applyAsDouble applyAsInt applyAsLong
DoubleToIntFunction DoubleToLongFunction IntToDoubleFunction IntToLongFunction LongToDoubleFunction LongToIntFunction	1	int long double long double int	applyAsInt applyAsLong applyAsDouble applyAsLong applyAsDouble applyAsInt
ObjDoubleConsumer<T> ObjIntConsumer<T> ObjLongConsumer<T>	2	void	accept

Advanced Stream Concepts

There are a few more advanced topics to cover. You'll see the relationship between streams and the underlying data, chaining Optional and grouping collectors.

Linking Streams to the Underlying Data

What do you think this outputs?

```
List<String> departments = new ArrayList<>();
cats.add("HR");
cats.add("IT");
Stream<String> stream = departments.stream();
cats.add("Sales");
System.out.println(stream.count());
```

The correct answer is 3.

The first three lines create a List with two elements.

The next line requests that a stream be created from that List.

Remember that streams are lazily evaluated. This means that the stream isn't actually created where it is requested in the code. An object is created that knows where to look for the data when it is needed.

Then the List gets a new element. When we try to evaluate count(), the stream pipeline actually runs. The stream pipeline runs first, looking at the source and seeing three elements.

Chaining Optionals

By now, you are familiar with the benefits of chaining operations in a stream pipeline. A few of the intermediate operations for streams are available for Optional.

Suppose that you are given an `Optional<Integer>` and asked to print the value, but only if it is a three-digit number. Without functional programming, you could write the following:

```
private static void threeDigit(Optional<Integer> optional) {
    if (optional.isPresent()) { // outer if
        Integer num = optional.get();
        String string = "" + num;
        if (string.length() == 3){ // inner if
            System.out.println(string);
        }
    }
}
```

It works, but it contains nested if statements. That's extra complexity. Let's try this again with functional programming:

```
private static void threeDigits(Optional<Integer> optional) {
    optional.map(n -> "" + n) // part 1
        .filter(s -> s.length() == 3) // part 2
        .ifPresent(System.out::println); // part 3
}
```

This is much shorter and more expressive.

Suppose that we are given an empty Optional. The first approach returns false for the outer if. The second approach sees an empty Optional and has both map() and filter() pass it through. Then ifPresent() sees an empty Optional and doesn't call the Consumer parameter.

The next case is where we are given an Optional.of(4). The first approach returns false for the inner if. The second approach maps the number 4 to the String "4". The filter then returns an empty Optional since the filter doesn't match, and ifPresent() doesn't call the Consumer parameter.

The final case is where we are given an Optional.of(123). The first approach returns true for both if statements. The second approach maps the number 123 to the String "123". The filter then returns the same Optional, and ifPresent() now does call the Consumer parameter.

Now suppose that we wanted to get an Optional<Integer> representing the length of the String contained in another Optional. Easy enough:

```
Optional<Integer> result = optional.map(String::length);
```

What if we had a helper method that did the logic of calculating something for us and it had the signature static Optional<Integer> calculator(String s)? Using map doesn't work:

```
Optional<Integer> result = optional.map(ChainingOptional::calculator);
// DOES NOT COMPILE
```

The problem is that calculator returns Optional<Integer>. The map() method adds another Optional, giving us Optional<Optional<Integer>>. Well, that's no good. The solution is to call flatMap() instead:

```
Optional<Integer> result =
    optional.flatMap(ChainingOptional::calculator);
```

This one works because flatMap removes the unnecessary layer. In other words, it attenuates the result. Chaining calls to flatMap() is useful when you want to transform one Optional type to another.

Collecting Results

We're almost finished with the advanced topics about streams. This last topic builds on what you've learned so far to group the results. Early in the chapter, you saw the collect() terminal operation. There are many predefined collectors, the most important are shown in this table:

Collector	Description	Return value
averagingDouble(ToDoubleFunction f) averagingIntToIntFunction f) averagingLong(ToLongFunction f)	Calculates the average for our three core primitive types	Double
counting()	Counts number of elements	long
groupingBy(Function f) groupingBy(Function f, Collector dc) groupingBy(Function f, Supplier s, Collector dc)	Creates a map grouping by the specified function with the optional type and optional downstream collector	Map<K, List<T>>
joining() joining(CharSequence cs)	Creates a single String using cs as a delimiter between elements if one is specified	String
maxBy(Comparator c) minBy(Comparator c)	Finds the largest/smallest elements	Optional<T>
mapping(Function f, Collector dc)	Adds another level of collectors	Collector
partitioningBy(Predicate p) partitioningBy(Predicate p, Collector dc)	Creates a map grouping by the specified predicate with the optional further downstream collector	Map<Boolean, List<T>>
summarizingDouble(ToDoubleFunction f) summarizingIntToIntFunction f) summarizingLong(ToLongFunction f)	Calculates average, min, max, and so on	DoubleSummaryStatistics IntSummaryStatistics LongSummaryStatistics
summingDouble(ToDoubleFunction f) summingIntToIntFunction f) summingLong(ToLongFunction f)	Calculates the sum for our three core primitive types	Double Integer Long
toList()	Creates an	List

<code>toSet()</code>	arbitrary type of list or set	Set
<code>toCollection(Supplier s)</code>	Creates a Collection of the specified type	Collection
<code>toMap(Function k, Function v)</code> <code>toMap(Function k, Function v, BinaryOperator m)</code> <code>toMap(Function k, Function v, BinaryOperator m, Supplier s)</code>	Creates a map using functions to map the keys, values, an optional merge function, and an optional type	Map

Basic Collectors

Luckily, many of these collectors work in the same way. Let's look at an example:

```
Stream<String> moves = Stream.of("forward", "left", "back");
String result = moves.collect(Collectors.joining(", "));
System.out.println(result); // forward, left, back
```

Notice how the predefined collectors are in the `Collectors` class rather than the `Collector` class. This is a common theme, which you saw with `Collection` vs. `Collections`. We pass the predefined `joining()` collector to the `collect()` method. All elements of the stream are then merged into a `String` with the specified delimiter between each element.

It is very important to pass the `Collector` to the `collect` method. It exists to help collect elements. A `Collector` doesn't do anything on its own.

Let's try another one. What is the average length of the three move instructions?

```
Stream<String> moves = Stream.of("forward", "left", "back");
Double result = moves.collect(Collectors.averagingInt(String::length));
System.out.println(result); // 5.0
```

The pattern is the same. We pass a collector to `collect()` and it performs the average for us. This time, we needed to pass a function to tell the collector what to average. We used a method reference, which returns an `int` upon execution. With primitive streams, the result of an average was always a `double`, regardless of what type is being averaged. For collectors, it is a `Double` since those need an `Object`.

Often, you'll find yourself interacting with code that was written prior to Java 8. This means that it will expect a `Collection` type rather than a `Stream` type. No problem. You can still express yourself using a `Stream` and then convert to a `Collection` at the end, for example:

```
Stream<String> moves = Stream.of("forward", "left", "back");
TreeSet<String> result = moves.filter(s -> s.contains("a"))
```

```
.collect(Collectors.toCollection(TreeSet::new));
System.out.println(result); // [forward, back]
```

This time we have all three parts of the stream pipeline. Stream.of() is the source for the stream. The intermediate operation is filter(). Finally, the terminal operation is collect(), which creates a TreeSet. If we didn't care which implement of Set we got, we could have written Collectors.toSet() instead.

At this point, you should be able to use all of the Collectors in the table above except groupingBy(), mapping(), partitioningBy(), and toMap().

Collecting into Maps

Collector code involving maps can get long. We will build it up slowly. Make sure that you understand each example before going on to the next one. Let's start out with a straightforward example to create a map from a stream:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<String, Integer> map = moves.collect(Collectors
    .toMap(s -> s, String::length));
System.out.println(map); // {forward=7, left=4, back=4}
```

When creating a map, you need to specify two functions. The first function tells the collector how to create the key. In our example, we use the provided String as the key. The second function tells the collector how to create the value. In our example, we use the length of the String as the value.

Returning the same value passed into a lambda is a common operation, so Java provides a method for it. You can rewrite `s -> s` as `Function.identity()`. It is not shorter and may or may not be clearer, so use your judgment on whether to use it.

Now we want to do the reverse and map the length of the move instruction to the name itself. Our first incorrect attempt is shown here:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Integer, String> map = moves.collect(Collectors
    .toMap(String::length, k -> k)); // BAD
```

Running this gives an exception similar to the following:

```
Exception in thread "main" java.lang.IllegalStateException:
        Duplicate key back
at
java.util.stream.Collectors$Lambda$114(Collectors.java:133)
at java.util.stream.Collectors$$Lambda$3/1044036744.apply(Unknown Source)
```

What's wrong? Two of the move instructions are the same length. We didn't tell Java what to do. Should the collector choose the first one it encounters? The last one it encounters? Concatenate the two? Since the collector has no idea what to do, it "solves" the problem by throwing an exception and making it our problem. How thoughtful. Let's suppose that our requirement is to create a comma-separated String with the move instructions. We could write this:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Integer, String> map = moves.collect(Collectors.toMap(
    String::length, k -> k, (s1, s2) -> s1 + "," + s2));
System.out.println(map); // {4=left, back, 7=forward}
System.out.println(map.getClass()); // class java.util.HashMap
```

It so happens that the Map returned is a HashMap. This behaviour is not guaranteed. Suppose that we want to mandate that the code return a TreeMap instead. No problem. We would just add a constructor reference as a parameter:

```
Stream<String> moves = Stream.of("forward", "left", "back");
TreeMap<Integer, String> map = moves.collect(Collectors.toMap(
    String::length, k -> k, (s1, s2) -> s1 + "," + s2, TreeMap::new));
System.out.println(map); // // {4=left, back, 7=forward}
System.out.println(map.getClass()); // class java.util.TreeMap
```

This time we got the type that we specified. With us so far? This code is long but not particularly complicated. We did promise you that the code would be long!

Grouping, Partitioning, and Mapping

Now suppose that we want to get groups of names by their length. We can do that by saying that we want to group by length:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Integer, List<String>> map = moves.collect(
    Collectors.groupingBy(String::length));
System.out.println(map); // {4=[left, back], 7=[forward]}
```

The groupingBy() collector tells collect() that it should group all of the elements of the stream into lists, organizing them by the function provided. This makes the keys in the map the function value and the values the function results.

Suppose that we don't want a List as the value in the map and prefer a Set instead. No problem. There's another method signature that lets us pass a downstream collector. This is a second collector that does something special with the values:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Integer, Set<String>> map = moves.collect(
    Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println(map); // {4=[left, back], 7=[forward]}
```

We can even change the type of Map returned through yet another parameter:

```
Stream<String> moves = Stream.of("forward", "left", "back");
TreeMap<Integer, Set<String>> map = moves.collect(
    Collectors.groupingBy(String::length, TreeMap::new,
        Collectors.toSet()));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

Partitioning is a special case of grouping. With partitioning, there are only two possible groups—true and false. Partitioning is like splitting a list into two parts.

Suppose that we are making a token for each move. We have two sizes of token. One can accommodate names with five or fewer characters. The other is needed for longer names. We can partition the list according to which token we need:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Boolean, List<String>> map = moves.collect(
    Collectors.partitioningBy(s -> s.length() <= 5));
System.out.println(map); // {false=[forward], true=[left, back]}
```

Here we passed a Predicate with the logic for which group each move instruction belongs in. Now suppose that we've figured out how to use a different font, and seven characters can now fit on the smaller token. No worries. We just change the Predicate:

```
Stream<String> moves = Stream.of("forward", "left", "back");
Map<Boolean, List<String>> map = moves.collect(
    Collectors.partitioningBy(s -> s.length() <= 7));
System.out.println(map); // {false=[], true=[forward, left, back]}
```

Notice that there are still two keys in the map—one for each boolean value. It so happens that one of the values is an empty list, but it is still there.

Summary

Lambdas can reference static variables, instance variables, effectively final parameters, and effectively final local variables. A functional interface has a single abstract method.

Here are the key functional interfaces:

- Supplier<T>: Method get() returns T
- Consumer<T>: Method accept(T t) returns void
- BiConsumer<T>: Method accept(T t, U u) returns void
- Predicate<T>: Method test(T t) returns boolean
- BiPredicate<T>: Method test(T t, U u) returns boolean
- Function<T, R>: Method apply(T t) returns R
- BiFunction<T, U, R>: Method apply(T t, U u) returns R
- UnaryOperator<T>: Method apply(T t) returns T
- BinaryOperator<T>: Method apply(T t1, T t2) returns T

An Optional can be empty or store a value. You can check if it contains a value with ifPresent() and get() the value inside.

There are also three methods that take functional interfaces as parameters:

- ifPresent(Consumer c)
- orElseGet(Supplier s)
- orElseThrow(Supplier s)

There are three optional types for primitives: DoubleSupplier, IntSupplier, and LongSupplier. These have the methods getDouble(), getInt(), and getLong(), respectively.

A stream pipeline has three parts. The source is required, and it creates the data in the stream. There can be zero or more intermediate operations, which aren't executed until the terminal operation runs. Examples of intermediate operations include filter(), flatMap(), and sorted(). Examples of terminal operations include allMatch(), count(), and forEach().

There are three primitive streams: DoubleStream, IntStream, and LongStream. In addition to the usual Stream methods, they have range() and rangeClosed(). The call range(1, 10) on IntStream and LongStream creates a stream of the primitives from 1 to 9. By contrast, rangeClosed(1, 10) creates a stream of the primitives from 1 to 10.

The primitive streams have math operations including average(), max(), and sum(). They also have summaryStatistics() to get many statistics in one call. There are also functional interfaces specific to streams. Except for BooleanSupplier, they are all for double, int, and long primitives as well.

You can use a Collector to transform a stream into a traditional collection. You can even group fields to create a complex map in one line. Partitioning works the same way as grouping, except that the keys are always true and false. A partitioned map always has two keys even if the value is empty for the key.

Finally, remember that streams are lazily evaluated. They take lambdas or method references as parameters, which occur later when the method is run.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 5

Dates, Strings and Localisation

Introduction

This chapter looks at Dates and Times in Java and starts with a review of the basic concepts and then moves on to cover more advanced concepts including time zones, daylight savings time, and comparing values and instants.

Strings are not listed in the objectives for the OCP Java Programming II exam but they are involved in so much else in the language and dealing with them efficiently is very important. This chapter covers the important points.

There are also some important classes that will help to make your application work in different languages with localisation.

Finally you will learn how to read and write numbers, dates, and money using different international formats.

Date and Time API Classes

Java 8 introduces a completely new set of classes to work with dates and times. The old classes (Date and Calendar) are still supported but you are strongly recommended to use the new classes as they are more efficient and easier to read. They are also popular with other developers so you will find your colleagues use them by preference.

You need an import to work with the date and time classes. Most of them are in the `java.time` package. To use it, add this import to your program:

```
import java.time.*; // import time classes
```

Creating Dates and Times

People usually talk about dates in their own time zone. When someone says, “I’ll call you at 7pm on Tuesday evening,” we assume that 19:00 means the same thing to both of us.

But if I live in Hong Kong and you live in Cape Town, we need to be more specific. Cape Town is six hours earlier than Hong Kong because they are in different time zones. You would instead say “I’ll call you at 19:00 SAST (South Africa Standard Time) on Tuesday evening.”

When working with dates and times, the first thing to do is to decide how much information you need. The exam gives you four choices:

- `LocalDate` Contains just a date—no time and no time zone. E.g. date of birth – you just need to know the day and it doesn’t make any difference what time you were born.
- `LocalTime` Contains just a time—no date and no time zone. E.g. event time like backups at 2am.
- `LocalDateTime` Contains both a date and time but no time zone. E.g. appointment where the time zone is implied by the location
- `ZonedDateTime` Contains a date, time, and time zone. E.g. conference call between people from multiple global locations.

Oracle recommends avoiding time zones unless you really need them. Try to act as if everyone is in the same time zone when you can.

As you may remember, you obtain date and time instances using a static method:

```
System.out.println(LocalDate.now());  
System.out.println(LocalTime.now());  
System.out.println(LocalDateTime.now());  
System.out.println(ZonedDateTime.now());
```

Each of the four classes has a static method called `now()`, which gives the current date and time.

Your output is going to depend on the date/time when you run it and where you live.

The output looks like the following when run on September 27 at 4:10 pm:

```
2017-09-27
16: 10: 20. 087
2017-09-27T16: 10: 20. 088
2017-09-27T16: 10: 20. 091+01: 00[Europe/London]
```

Now that you know how to create the current date and time, let's look at other specific dates and times.

To begin, let's create just a date with no time. Both of these examples create the same date:

```
Local Date date1 = Local Date. of(2015, Month. JANUARY, 20);
Local Date date2 = Local Date. of(2015, 1, 20);
```

Both pass in the year, month, and date. Although it is good to use the Month constants (to make the code easier to read), you can pass the int number of the month directly. Just use the number of the month the same way you would if you were writing the date in real life.

The method signatures are as follows:

```
public static Local Date of(int year, int month, int dayOfMonth)
public static Local Date of(int year, Month month, int dayOfMonth)
```

Month is an enum. Remember that an enum is not an int and cannot be compared to one, for example:

```
Month month = Month. JANUARY;
boolean b1 = month == 1; // DOES NOT COMPILE
boolean b2 = month == Month. APRIL; // false
```

When creating a time, you can choose how detailed you want to be. You can specify just the hour and minute, or you can include the number of seconds. You can even include nanoseconds if you want to be very precise. (A nanosecond is a billionth of a second, though you probably won't need to be that specific.)

The method signatures are as follows:

```
public static Local Time of(int hour, int minute)
public static Local Time of(int hour, int minute, int second)
public static Local Time of(int hour, int minute, int second, int nanos)
```

You can combine dates and times into one object:

```
Local DateTi me dateTime1 =
    Local DateTi me. of(2015, Month. JANUARY, 20, 6, 15, 30);

Local Date date1 = Local Date. of(2015, Month. JANUARY, 20);
Local Time time1 = Local Ti me. of(6, 15);
Local DateTi me dateTime2 = Local DateTi me. of(date1, time1);
```

The first line of code shows how you can specify all of the information about the LocalDate/Time right in the same line. There are many method signatures allowing you to specify different things. Having that many numbers in a row gets to be hard to read,

though. The other lines of code show how you can create LocalDate and LocalTime objects separately first and then combine them to create a LocalDateTime object.

Now there are a lot of method signatures since there are more combinations. You can see them all on the Javadocs Web pages.

In order to create a ZonedDateTime, we first need to get the desired time zone. We will use Europe/London in our examples:

```
ZoneId zone = ZoneId.of("Europe/London");
ZonedDateTime zdt1 = ZonedDateTime.of(2017, 9, 27, 16, 20, 0, 0, zone);
ZonedDateTime zdt2 = ZonedDateTime.of(date1, time1, zone);
ZonedDateTime zdt3 = ZonedDateTime.of(dateTime1, zone);
```

There is no overload that uses the Month enum, at least not in Java 8.

Finding out your time zone is easy. You can just print it out with:

```
System.out.println(ZoneId.systemDefault());
```

You have to use the factory methods for all these classes as the constructors are all private.

There is a DateTimeException that Java will throw if you put an invalid date (month > 12 etc.)

Manipulating Dates and Times

Adding to a date is easy. The date and time classes are immutable but you just assign the results of these methods to a reference variable so that they are not lost.

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date.plusDays(2));
System.out.println(date.plusWeeks(8));
System.out.println(date.plusMonths(6));
System.out.println(date.plusYears(1));
```

This code is very readable because it does just what it looks like it should do.

Java is smart enough to take into account leap years and adds/subtracts appropriately.

The methods for subtraction are similarly easy to use and understand:

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
System.out.println(date.minusDays(2));
System.out.println(date.minusWeeks(8));
System.out.println(date.minusMonths(6));
System.out.println(date.minusYears(1));
```

Java hides seconds and nanoseconds when not being used.

Since these methods return the same object type, you can chain them:

```
LocalDate date = LocalDate.of(2014, Month.JANUARY, 20)
    .plusYears(2).plusMonths(6).plusDays(2);
System.out.println(date);
```

There are plus and minus methods for:

- Years
- Months
- Weeks
- Days
- Hours
- Minutes
- Seconds
- Nanos

Period Class

Java provides a Period class for dealing with time periods that will be reused.

You can create a Period using various factory methods. The signatures are shown here:

```
static Period of(int years, int months, int days)
static Period ofDays(int days)
static Period ofMonths(int months)
static Period ofWeeks(int weeks)
static Period ofYears(int years)
```

These are pretty self-explanatory.

They can be used to adjust LocalDate, LocalDateTime and ZonedDateTime objects using the plus() and minus() methods of those objects.

These cannot be chained as they are static so you just have to use the of() method if you want a Period containing a combination of years, months and days.

If you try adjusting a LocalTime object with a Period you will get an UnsupportedTemporalTypeException.

Period has an overridden toString() method. The signature and explanation from Javadocs is:

```
public String toString()
```

Outputs this period as a String, such as P6Y3M1D.

The output will be in the ISO-8601 period format. A zero period will be represented as zero days, 'P0D'.

Duration Class

Period is used for a day or more of time. There is also Duration, which is intended for smaller units of time. For Duration, you can specify the number of days, hours, minutes, seconds, or nanoseconds. You could pass 365 days to make a year, but you'd be much better using a Period.

Conveniently, Duration roughly works the same way as Period, except it is used with objects that have time.

Remember that a Period is output beginning with a P. Duration is output beginning with PT, which you can think of as a period of time. A Duration is stored in hours, minutes, and seconds. The number of seconds includes fractional seconds.

We can create a Duration using a number of different granularities:

```
Duration daily = Duration.ofDays(1); // PT24H
Duration hourly = Duration.ofHours(1); //PT1H
Duration minute = Duration.ofMinutes(1); //PT1M
Duration tenSeconds = Duration.ofSeconds(10); //PT10S
Duration milli = Duration.ofMillis(1); //PT0.001S
Duration nano = Duration.ofNanos(1); //PT0.000000001S
```

This is similar to Period. We pass a number to the most appropriate method and this makes the code readable.

Duration doesn't have a constructor that takes multiple units like Period does. If you want something to happen every hour and a half, you would specify 90 minutes.

Duration includes another more generic factory method. It takes a number and a TemporalUnit. The idea is, say, something like "5 seconds." However, TemporalUnit is an interface. At the moment, there is only one implementation named ChronoUnit.

The previous examples could be rewritten as this:

```
Duration daily = Duration.of(1, ChronoUnit.DAYS);
Duration hourly = Duration.of(1, ChronoUnit.HOURS);
Duration minute = Duration.of(1, ChronoUnit.MINUTES);
Duration tenSeconds = Duration.of(10, ChronoUnit.SECONDS); Duration
milli = Duration.of(1, ChronoUnit.MILLISECONDS);
Duration nano = Duration.of(1, ChronoUnit.NANOS);
```

ChronoUnit also includes some convenient units such as ChronoUnit.HALF_DAYS to represent 12 hours.

Duration objects are used with the plus() and minus() methods of objects with a time (e.g. not LocalDate).

Instant Class

The Instant class represents a specific moment in time in the GMT time zone.

Suppose that you want to run a timer:

```
Instant start = Instant.now();
// do something time consuming
Instant finish = Instant.now();
Duration duration = Duration.between(now, later);
System.out.println(duration.toMillis());
```

Depending on what the "something time consuming" was, you might get a different number but if it was about a second then you will get something around 1000.

If you have a ZonedDateTime, you can turn it into an Instant:

```
Local Date date = Local Date. of(2017, 9, 27);
Local Time time = Local Time. of(16, 20, 00);
ZoneId zone = ZoneId. of("Europe/London");
ZonedDateTime zonedDateTime = ZonedDateTime. of(date, time, zone);
Instant instant = zonedDateTime. toInstant();
System.out.println(zonedDateTime);
System.out.println(instant);
```

The last two lines represent the same moment in time. The ZonedDateTime includes a time zone. The Instant gets rid of the time zone and turns it into an Instant of time in GMT.

You cannot convert a LocalDateTime to an Instant. Remember that an Instant is a point in time. A LocalDateTime does not contain a time zone, and it is therefore not universally recognised around the world as the same moment in time.

Using an Instant, you can do arithmetic. Instant allows you to add any unit day or smaller, for example:

```
Instant nextDay = instant. plus(1, ChronoUnit. DAYS);
System.out.println(nextDay); // 2015-05-26T15:55:00Z
Instant nextHour = instant. plus(1, ChronoUnit. HOURS);
System.out.println(nextHour); // 2015-05-25T16:55:00Z
Instant nextWeek = instant. plus(1, ChronoUnit. WEEKS); // Exception
```

It's strange that an Instant displays a year and month while preventing you from doing maths with those fields.

Daylight Savings Time

Some countries like the UK observe daylight savings time and the clocks are adjusted by an hour twice a year to make better use of the sunlight. Not all countries participate, and those that do use different weekends for the change.

In Britain, we move the clocks an hour ahead in March and move them an hour back in October. We officially change our clocks forward at 1:00am and back at 2:00am, which is very early on the Sunday morning.

For example, on March 26, 2017, we move our clocks forward an hour and jump from 1:00 a.m. to 2:00 a.m. This means that there is no 1:30 a.m. that day. If we wanted to know the time an hour later than 12:30, it would be 2:30.

```
Local Date date = Local Date. of(2017, Month. MARCH, 26);
Local Time time = Local Time. of(0, 30);
ZoneId zone = ZoneId. of("Europe/London");
ZonedDateTime dateTi me = ZonedDateTime. of(date, time, zone);
System.out.println(dateTi me); // 2017-03-26T00:30Z[Europe/London]
dateTi me = dateTi me. plusHours(1);
System.out.println(dateTi me); // 2017-03-26T02:30+01:00[Europe/London]
```

Notice that two things change in this example. The time jumps from 0:30 to 2:30. The UTC offset also changes. This shows that the time really did change by one hour from GMT's point of view.

Trying to create a time that doesn't exist just rolls it forward. If you tried to create a time of 1:30am in the previous example, it would actually become 2:30am.

String Class Functionality

As you know, a string is a sequence of characters and the String class provides methods for accessing individual characters or substrings by using zero-based indexes.

Since there are so many String objects in a program, the String class is final and String objects are immutable meaning the value of the string cannot change.

This allows Java to optimize by storing string literals in the string pool. This also means that you can compare string literals with `==`. However, this cannot be relied upon and it is still a good idea to compare with `equals()`, because String objects created via a constructor (`new String("...")`) or a method call will not always match when using comparison with `==`.

Since String is such a fundamental class, Java allows using the `+` operator to combine them, which is called concatenation.

Strings have a number of useful instance methods:

- `charAt()` Returns the character located at the specified index
- `concat()` Appends one string to the end of another (`+` also works)
- `equalsIgnoreCase()` Determines the equality of two strings, ignoring case
- `length()` Returns the number of characters in a string
- `replace()` Replaces occurrences of a character with a new character
- `substring()` Returns a part of a string
- `toLowerCase()` Returns a string, with uppercase characters converted to lowercase
- `toString()` Returns the value of a string
- `toUpperCase()` Returns a string, with lowercase characters converted to uppercase
- `trim()` Removes whitespace from both ends of a string

StringBuilder

Since String is immutable, it is inefficient for when you are updating the value in a loop. StringBuilder is better for that scenario. A StringBuilder is mutable, which means that it can change value and increase in capacity. If multiple threads are updating the same object, you should use StringBuffer rather than StringBuilder.

StringBuilder and StringBuffer have identical methods.

Here are some of the useful ones:

```
public StringBuilder append(String s)
```

As you've seen earlier, this method will update the value of the object that invoked the method, whether or not the returned value is assigned to a variable. This method will take many different arguments, including boolean, char, double, float, int, long, and others.

```
public StringBuilder delete(int start, int end)
```

This method modifies the value of the `StringBuilder` object used to invoke it. The starting index of the substring to be removed is defined by the first argument (which is zero-based), and the ending index of the substring to be removed is defined by the second argument (but it is one-based)! Here is `StringBuilder` in action:

```
StringBui lder sb = new StringBui lder("0123456789");
System.out.println(sb.delete(4, 6)); // output is "01236789"

public StringBui lder insert(int offset, String s)
```

This method updates the value of the `StringBuilder` object that invoked the method call. The String passed in to the second argument is inserted into the `StringBuilder` starting at the offset location represented by the first argument (the offset is zero-based). Again, other types of data can be passed in through the second argument (boolean, char, double, float, int, long, and so on).

```
public StringBui lder reverse()
```

This method updates the value of the `StringBuilder` object that invoked the method call. When invoked, the characters in the `StringBuilder` are reversed.

```
public String toString()
```

This method returns the value of the `StringBuilder` object that invoked the method call as a String.

Internationalisation and Localisation

Internationalisation is the process of designing your program so it can be adapted. This involves placing strings in a property file and using classes like DateFormat so that the right format is used based on user preferences. You do not actually need to support more than one language or country to internationalise the program. Internationalisation just means that you can.

Localisation means actually supporting multiple locales. Oracle defines a locale as “a specific geographical, political, or cultural region.” You can think of a locale as being like a language and country pairing. Localisation includes translating strings to different languages. It also includes outputting dates and numbers in the correct format for that locale. You can go through the localisation process many times in the same application as you add more languages and countries.

In this section, we will look at how to define a locale, work with resources bundles, and format dates and numbers.

Specifying a Locale

While Oracle defines a locale as ‘a specific geographical, political, or cultural region’ and so could be something different, this section only looks at languages and countries.

The Locale class is in the java.util package. The first useful Locale to find is the user’s current locale. Try running the following code on your computer:

```
Locale local = Locale.getDefault();
System.out.println(local);
```

When we run it, it prints en_GB. It might be different for you. This default output tells us that our computers are using English and are in the UK.

Notice the format. First comes the lowercase language code. Then comes an underscore followed by the uppercase country code. The underscore and country code are optional. It is valid for a Locale to be only a language (en).

You can also use a Locale other than the default. There are three main ways of creating a Locale. First, the Locale class provides constants for some of the most commonly used locales:

```
System.out.println(Locale.GERMAN); // de
System.out.println(Locale.GERMANY); // de_DE
```

Notice that the first one is the German language and the second is Germany the country; similar, but not the same. The other two main ways of creating a Locale are to use the constructors. You can pass just a language or both a language and country:

```
System.out.println(new Locale("fr")); // fr
System.out.println(new Locale("hi", "IN")); // hi_IN
```

The first is the language French and the second is Hindi in India.

Java will let you create a Locale with an invalid language or country. However, it will not match the Locale that you want to use and your program will not behave as expected.

There's another way to create a Locale that is more flexible.

The builder design pattern lets you set all of the properties that you care about and then build it at the end. This means that you can specify the properties in any order:

```
Locale loc = new Locale.Builder()
    .setRegion("GB")
    .setLanguage("en")
    .build();
```

When testing a program, you might need to use a Locale other than the default for your computer. You can set a new default right in Java:

```
System.out.println(Locale.getDefault()); // en_US
Locale locale = new Locale("fr");
Locale.setDefault(locale); // change the default
System.out.println(Locale.getDefault()); // fr
```

Don't worry, the Locale changes only for that one Java program. It does not change any settings on your computer. It does not even change future programs. If you run the previous code multiple times, the output will stay the same.

Resource Bundles

A resource bundle contains the local specific objects to be used by a program. It is like a map with keys and values. The resource bundle can be in a property file or in a Java class. A property file is a file in a specific format with key/value pairs.

Up until now, we've kept all of the strings from our program in the classes that use them. Localisation requires externalising them to rather than hard-coding them in your program.

This is typically done using a property file, but it could be a resource bundle class that returns the values programmatically, e.g. from a database.

Creating a Property File Resource Bundle

Luckily, Java doesn't require us to create four different resource bundles. If we don't have a country-specific resource bundle, Java will use a language-specific one. It's a bit more involved than this, which we cover later in the chapter.

Let's say we need English and French property file resource bundles to start with.

First, create two property files:

Translate_en.properties

welcome=welcome

submit=submit

thankyou=thank you

Translate_fr.properties

welcome=bienvenue
 submit=envoyer
 thankyou=merci

Notice that the filenames are the name of our resource bundle followed by an underscore followed by the target locale. Here is a program that uses these resource bundle to print localised information:

```
import java.util.*;
public class WebPage {
    public static void main(String[] args) {
        Locale uk = new Locale("en", "GB");
        Locale france = new Locale("fr", "FR");
        printProperties(uk);
        System.out.println();
        printProperties(france);
    }
    public static void printProperties(Locale locale) {
        ResourceBundle rb = ResourceBundle.getBundle("Translate", locale);
        System.out.println("Page title: " + rb.getString("welcome"));
        System.out.println("Submit button: " + rb.getString("submit"));
        System.out.println("Confirmation: " + rb.getString("thankyou"));
    }
}
```

NOTE

Property File Format

The most common syntax is where a property file contains key/value pairs in the format:

key=value

There are two other less common formats that you can use to express these pairs.

- key:value
- key value

There are other forms of syntax in a property file. The common ones are these:

- Lines beginning with # or ! are comments
- Spaces before or after the separator character are ignored
- Spaces at the beginning of a line are ignored
- Spaces at the end of a line are not ignored
- End a line with a backslash if you want to break the line for readability
- You can use normal Java escape characters like \t and \n

You can loop through the keys and values to list all of the pairs.

The ResourceBundle class provides a method to get a set of all keys:

```
Locale uk = new Locale("en", "GB");
ResourceBundle rb = ResourceBundle.getBundle("Translate", uk);
Set<String> keys = rb.keySet();
keys.stream().map(k -> k + " " + rb.getString(k))
    .forEach(System.out::println);
```

This example goes through all of the keys using a stream.

Java Class Resource Bundles

A property file resource bundle is usually enough to meet the program's needs. It does have a limitation in that only String values are allowed. Java class resource bundles allow any Java type as the value. Keys are always strings though.

To implement a resource bundle in Java, you create a class with the same name that you would use for a property file. Only the extension is different. Since we have a Java object, the file must be a .java file rather than a .properties file. For example, the following class is equivalent to the English property file that you saw in the last section:

```
import java.util.*;
public class Translate_en extends ListResourceBundle {
    protected Object[][] getContents() {
        return new Object[][] {
            { "welcome", "welcome" },
            { "submit", "submit" },
            { "thankyou", "thank you" }
        };
    }
}
```

The ListResourceBundle abstract superclass leaves one method called `getContents()` for subclasses to implement. The rest of the code creates a 2D array with the keys hello and open.

There are two main advantages of using a Java class instead of a property file for a resource bundle:

- You can use a value type that is not a String
- You can create the values of the properties at runtime

Selecting the Resource Bundle

There are two main methods for getting a resource bundle:

```
ResourceBundle.getBundle("name");
ResourceBundle.getBundle("name", Locale);
```

The first one uses the default locale. You are likely to use this one in programs that you write.

Java handles the logic of picking the best available resource bundle for a given key. It tries to find the most specific value. When there is a tie, Java class resource bundles are given preference.

This table shows what Java goes through when asked for resource bundle Translate with the locale new Locale("fr", "FR") when the default locale is British English:

Step	Selects if Present	Logic
1	Translate_fr_FR.java	Requested locale
2	Translate_fr_FR.properties	Requested locale
3	Translate_fr.java	Requested language, no country
4	Translate_fr.properties	Requested language, no country
5	Translate_en_GB.java	Default locale
6	Translate_en_GB.properties	Default locale
7	Translate_en.java	Default language, no country
8	Translate_en.properties	Default language, no country
9	Translate.java	No locale
10	Translate.properties	No locale
11	Throws MissingResourceException	No matching file found

Java can't always get all of the keys from the same resource bundle. It can get them from any parent of the matching resource bundle if they aren't in the matching one. A parent resource bundle in the hierarchy just removes components of the name until it gets to the top.

The parents of Translate_fr_FR.java are:

- Translate_fr.java
- Translate.java

Formatting Numbers

The `java.text` package has classes to that allow formatting numbers (including currency) and dates. You can also use these classes to convert from a string to a number or date by parsing the string.

The following sections cover how to format and parse numbers, currency, and dates.

Regardless of whether you want to format or parse, the first step is the same. You need to create a `NumberFormat`. The class provides factory methods to get the desired formatter.

This table shows the available methods:

Use	Factory Method
General purpose formatting	<code>getInstance()</code> <code>getInstance(locale)</code>
Same as above	<code>getNumberInstance()</code> <code>getNumberInstance(locale)</code>
Formatting currency	<code>getCurrencyInstance()</code> <code>getCurrencyInstance(locale)</code>
Formatting percentages	<code>getPercentagelInstance()</code> <code>getPercentagelInstance(locale)</code>
Rounding decimals for display	<code>getIntegerInstance()</code> <code>getIntegerInstance(locale)</code>

Once you have the `NumberFormat` instance, you can call `format()` to turn a number into a String and `parse()` to turn a String into a number.

NOTE

The format classes are not thread-safe so do not store them in instance or static variables.

Formatting

The `format` method formats the given number based on the locale associated with the `NumberFormat` object. For marketing literature, we want to share the monthly number of visitors to the Web site.

The following shows printing out the same number in three different locales:

```

import java.text.*;
import java.util.*;
public class FormatNumbers {
    public static void main(String[] args) {
        int visitorsPerMonth = 8129;
        NumberFormat uk = NumberFormat.getInstance(Locale.GB);
        System.out.println(uk.format(visitorsPerMonth));
        NumberFormat g = NumberFormat.getInstance(Locale.GERMANY);
        System.out.println(g.format(visitorsPerMonth));
        NumberFormat ca = NumberFormat.getInstance(Locale.CANADA_FRENCH);
        System.out.println(ca.format(visitorsPerMonth));
    }
}

```

output looks like this:

```

8,129
8.129
8 129

```

Formatting currency works the same way:

```

double price = 48;
NumberFormat ukcurr = NumberFormat.getCurrencyInstance();
System.out.println(ukcurr.format(price));

```

When run with the default locale of en_GB, the output is £48.00. Java automatically formats with two decimals and adds the pound sign. This is quite convenient whether you need to localise your program or not.

Parsing

The NumberFormat class defines a parse method for parsing a String into a number using a specific locale. The result of parsing depends on the locale. For example, if the locale is Britain and the number contains commas, the commas are treated as formatting symbols. If the locale is a country or language that uses commas as a decimal separator, the comma is treated as a decimal point. In other words, the value of the resulting number depends on the locale.

The parse methods for the different types of formats throw the checked exception ParseException if they fail to parse.

Let's look at an example. The following code parses an input price with different locales:

```

NumberFormat en = NumberFormat.getInstance(Locale.UK);
NumberFormat fr = NumberFormat.getInstance(Locale.FRANCE);
String s = "40.45";
System.out.println(en.parse(s)); // 40.45
System.out.println(fr.parse(s)); // 40

```

In the UK, a dot is part of a number and the number is parsed how you might expect. France does not use a decimal point to separate numbers (they use the comma).

Java parses it as a formatting character, and it stops looking at the rest of the number.

This shows how important it is to parse using the right locale.

The parse method is also used for parsing currency. For example, we can read in the total sales from the Web site:

```
String amt = "£92,807.99";
NumberFormat cf = NumberFormat.getCurrencyInstance();
double value = (Double) cf.parse(amt);
System.out.println(value); // 92807.99
```

The currency string "£92,807.99" contains a pound sign and a comma. The parse method strips out the characters and converts the value to a number. The return value of parse is a Number object. Number is the parent class of all the java.lang wrapper classes, so the return value can be cast to its appropriate data type. The Number is cast to a Double and then automatically unboxed into a double.

The NumberFormat classes have other features and capabilities which you can find in the Javadocs if you need to do something more unusual with numbers.

Formatting Dates and Times

The date and time classes support many methods to obtain data from them:

```
Local Date date = Local Date. of(2020, Month. JANUARY, 20);
System. out. print l n(date. getDayOfWeek()); // MONDAY
System. out. print l n(date. getMonth()); // JANUARY
System. out. print l n(date. getYear()); // 2020
System. out. print l n(date. getDayOfYear()); // 20
```

We could use this information to display information about the date. However, it would be more work than necessary.

Java provides a class called `DateFormatter` to help us out. Unlike the `LocalDate` class, `DateFormatter` can be used to format any type of date and/or time object. What changes is the format.

`DateFormatter` is in the package `java.time.format`.

```
Local Date dt = Local Date. of(2020, Month. JANUARY, 20);
Local Time tm = Local Time. of(11, 12, 34);
Local DateTi me dtm = Local DateTi me. of(date, time);
System. out. print l n(dt. format(DateTi meFormatter. ISO_LOCAL_DATE));
System. out. print l n(tm. format(DateTi meFormatter. ISO_LOCAL_TIME));
System. out. print l n(dtm. format(DateTi meFormatter. ISO_LOCAL_DATE_TIME));
```

ISO is a standard for dates. The output of the previous code looks like this:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

This is a reasonable way for computers to communicate, but not ideal for us humans! Also you might want to have more control about the look and feel of your program. Luckily, there are some predefined formats that are more useful:

```
DateTi meFormatter shortDateTi me =
        DateTi meFormatter. ofLocal i zedDate(FormatStyl e. SHORT);
System. out. print l n(shortDateTi me. format(dateTi me)); // 1/20/20
System. out. print l n(shortDateTi me. format(date)); // 1/20/20
System. out. print l n(shortDateTi me. format(ti me));
                                // UnsupportedTemporal TypeExcepti on
```

Here we say that we want a localised formatter in the predefined short format. The last line throws an exception because a time cannot be formatted as a date.

The `format()` method is declared on both the formatter objects and the date/time objects, allowing you to reference the objects in either order. The following statements print exactly the same thing as the previous code:

```
DateTi meFormatter shortDateTi me =
        DateTi meFormatter. ofLocal i zedDate(FormatStyl e. SHORT);
```

```
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

If you don't want to use one of the predefined formats, you can create your own. For example, this code spells out the month:

```
DateFormatter f =
        DateFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f)); // January 20, 2020, 11:12
```

- **MMMM** M represents the month. The more Ms you have, the more verbose the Java output. For example, M outputs 1, MM outputs 01, MMM outputs Jan, and MMMM outputs January.
- **dd** d represents the day in the month. As with month, the more ds you have, the more verbose the Java output. dd means to include the leading zero for a single-digit day.
- , Use , if you want to output a comma (this also appears after the year).
- **yyyy** y represents the year. yy outputs a two-digit year and yyyy outputs a four-digit year.
- **hh** h represents the hour. Use hh to include the leading zero if you're outputting a single-digit hour.
- : Use : if you want to output a colon.
- **mm** m represents the minute omitting the leading zero if present. mm is more common and represents the minutes using two digits.

Summary

A LocalDate contains just a date. A LocalTime contains just a time. A LocalDateTime contains both a date and time. A ZonedDateTime adds a time zone.

All four have private constructors and are created using LocalDate.now() or LocalDate.of() (or the equivalents for that class). Instant represents a moment in time.

Dates and times can be manipulated using plusX or minusX methods.

The date and time classes are all immutable, which means that the return value must be used or the operation will be ignored.

The Period class represents a number of days, months, or years to add to or subtract from a LocalDate, LocalDateTime, or ZonedDateTime.

The Duration class represents hours, minutes, and seconds. It is used with LocalTime, LocalDateTime, or ZonedDateTime.

UTC represents the time zone offset from zero. Daylight savings time is observed in the UK and other countries by moving the clocks ahead an hour in the spring and an hour back in the fall. Java changes time and UTC offset to account for this.

You can create a Locale class with a desired language and optional country. The language is a two-letter lowercase code, and the country is a two-letter uppercase code.

For example, en and en_GB are locales for English and UK English, respectively. ResourceBundle allows specifying key/value pairs in a property file or in a Java class. Java goes through candidate resource bundles from the most specific to the most general to find a match. If no matches are found for the requested locale, Java switches to the default locale and then finally the default resource bundle.

Java looks at the equivalent Java class before the property file for each locale. Once a matching resource bundle is found, Java only looks in the hierarchy of that resource bundle to find keys.

NumberFormat uses static methods to retrieve the desired formatter, such as one for currency. DateTimeFormatter is used to output dates and times in the desired format.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 6

Exceptions and Assertions

Introduction

A program can fail for many reasons. Here are just a few of the possibilities for program failure that you will probably encounter on a frequent basis:

- Trying to read a file or directory that doesn't exist
- Trying to access a database over a network, but the network connection is down
- Using an invalid SQL statement in your JDBC code
- Using the wrong specifier when parsing dates

As you can see, some of these are coding mistakes. Others are completely beyond your control. Your program can't help it if the network is temporarily unavailable. What it can do is deal with the situation.

We will cover the key terms used with exceptions, syntax, and rules for working with exceptions. We will also let you know which exception classes you should be familiar with.

Assertions were added to the Java language in version 1.4, and are intended to let you test your assumptions during development, without the expense (in both your time and program overhead) of writing exception handlers for exceptions that you assume will never happen once the program is out of development and fully deployed.

We will look at the basics of how assertions work, including how to enable them, how to use them, and how not to use them.

Handling Exceptions

A `RuntimeException` is the JVM's way of saying, "I give up. I don't know what to do right now. You deal with it." When you write a method, you can either deal with the exception or make it the calling code's problem.

If you have designed your class to follow the Single Responsibility Principle, be properly encapsulated and loosely coupled then you shouldn't need to know what other people (or classes) are going to do with your class/module/unit of code.

That means you won't know what they want to do if there's an exception. There are probably thousands of options. So your code shouldn't even think about it. Just throw an exception. This is the best way to communicate back to the calling class. You could use a special 'magic' return value and this used to be very popular. However it couples the code together as both classes need to know the special meanings of the possible return values.

The so called 'happy path' is when nothing goes wrong. With bad code, there might not be a happy path. For example, your code might have a bug where it always throws a `NullPointerException`. In this case, you have an exception path but not a happy path, because execution can never complete normally.

If the exception only occurs sometimes then you have at least two paths through your code and there are two scenarios that need testing before release. After all, if an exception should be thrown then you'd better test that it actually is thrown.

Exception Types

You may previously have learned about the major categories of exception classes.

The following diagram shows the hierarchy of these classes.

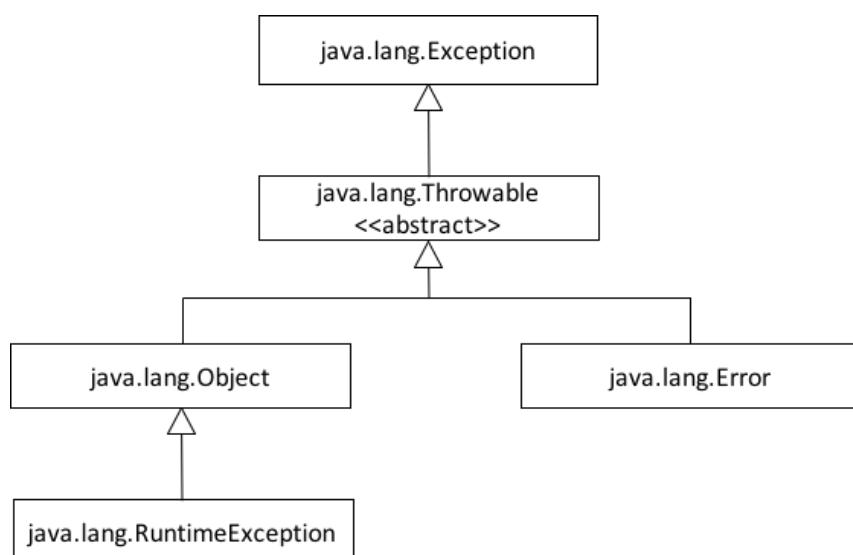


Figure 6-1 Exception Inheritance Hierarchy

Remember that a runtime exception, or unchecked exception, may be caught, but it is not required that it be caught.

After all, if you had to check for `NullPointerException`, every piece of code that you wrote would need to deal with it. A checked exception is any class that extends `Exception` but is not a runtime exception.

Checked exceptions must follow the ‘handle or declare’ rule where they are either caught or thrown to the caller. An error is fatal and should not be caught by the program. While it is legal to catch an error, it is not good practice.

Here are some examples of key exceptions in each category:

First runtime (unchecked) exceptions:

Exception	When thrown
<code>ArithmaticException</code>	Division by zero
<code>ArrayIndexOutOfBoundsException</code>	Use of an illegal or invalid index to access an array
<code>ClassCastException</code>	Casting an object to a subclass of which it is not an instance
<code>NullPointerException</code>	Calling a method on null
<code>java.lang.ArrayStoreException</code>	Assigning the wrong type to an array
<code>java.time.DateTimeException</code>	Using an invalid format string for a date
<code>java.util.MissingResourceException</code>	Accessing a non-existent resource bundle
<code>java.lang.IllegalStateException</code>	Running an invalid operation in collections and concurrency
<code>java.lang.UnsupportedOperationException</code>	

Now, checked exceptions:

Exception	When thrown
<code>IllegalArgumentException</code>	Passing an invalid argument to a method
<code>NumberFormatException</code>	Convert a string to a numeric type using an inappropriate format
<code>java.text.ParseException</code>	Converting a String to a number
<code>java.io.IOException</code>	Performing IO (input/output), this is the superclass of all IO exceptions
<code>java.io.FileNotFoundException</code>	Trying to access a non-existent file
<code>java.io.NotSerializableException</code>	Trying to serialize a class
<code>java.sql.SQLException</code>	Database issues, this is the superclass of all

	SQL exceptions
--	----------------

Try Blocks

The try block consists of a mandatory try clause. It can include one or more catch clauses to handle the exceptions that are thrown. It can also include a finally clause, which runs regardless of whether an exception is thrown. This is all still true for both try statements and try-with-resources statements.

```
try{
    // Possible exception-throwing code
}catch(ExceptionClassName identifier){
    // Exception handling code
    // ... potentially multiple catch blocks for less-specific exceptions
}finally{
    // Code that always needs to run e.g. closing resource streams
}
```

A simple try statement is required to have either or both of the catch and finally clauses. This is true for simple try statements, but is not true for try-with-resources statements

There are two other rules that you need to remember about the catch clauses:

- Java checks the catch blocks in the order in which they appear. It is illegal to declare a subclass exception in a catch block that is lower down in the list than a superclass exception because it will be unreachable code.
- Java will not allow you to declare a catch block for a checked exception type that cannot potentially be thrown by the try clause body. This is again to avoid unreachable code.

Throw and Throws

The exam might test whether you are paying attention to the difference between throw and throws. Remember that throw means an exception is actually being thrown and throws indicate that the method merely has the potential to throw that exception. The following example uses both:

```
public String getDataFromDatabase() throws SQLException {
    throw new UnsupportedOperationException();
}
```

The first line declares that the method might or might not throw a SQLException. Since this is a checked exception, the caller needs to handle or declare it. Then the code throws an UnsupportedOperationException. Since this is a runtime exception, it does not need to be declared as part of the method signature.

This might seem strange but is actually a common pattern. The implementer of this method hasn't written the logic to go to the database yet.

Maybe the database isn't available just now. The method still declares that it throws a SQLException, so any callers handle it right away and aren't surprised later by a change in method signature.

Custom Exceptions

Java provides many exception classes out of the box. Sometimes, you want to write a method with a more specialised type of exception. You can create your own exception class to do this.

When creating your own exception, you need to decide whether it should be a checked or unchecked exception. While you can extend any exception class, it is most common to extend Exception (for checked) or RuntimeException (for unchecked.)

Creating your own exception class is really easy.

```
public class HrException extends Exception {}
public class InvalidEmployeeException extends HrException {}
```

These are both valid classes that can be used with the throws and throw keywords. They do not provide any code so their only benefit is to provide the description contained in their name. This is often enough.

Exception is the superclass of all exceptions and it has the following constructors:

```
Exception()
Exception(String message)
Exception(String message, Throwable cause)
protected Exception(String message, Throwable cause,
                     boolean enableSuppression, boolean writableStackTrace)
Exception(Throwable cause)
```

If you do not provide any code then the 'no args' constructor provided by the compiler is the only one you can use with your custom exception.

If you want to provide a message then you need to provide the 'no args' constructor and one that accepts a String:

```
public class HrException extends Exception {
    public HrException(){}
    public HrException(String message){
        super(message);
    }
}
```

If you want to wrap another exception then also provide a constructor that accepts a Throwable.

Multi-catch Try Blocks

When something goes wrong in a program, it is common to log the error and convert it to a different exception type. In this example, we print the stack trace rather than write to a log. Next, we throw a runtime exception:

```
public static void main(String[] args) {
    try {
        Path path = Paths.get("File.txt");
        String text = new String(Files.readAllBytes(path));
        LocalDate date = LocalDate.parse(text);
        System.out.println(date);
    } catch (DateTimeParseException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    } catch (IOException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}
```

The two catch blocks on lines print a stack trace and then wrap the exception in a `RuntimeException`.

This works but as you know, duplicating code is bad. Think about what happens if we decide that we want to change the code to write to a log file instead of printing the stack trace. We have to be sure to change the code in two places.

Before Java 7, there were two approaches to deal with this problem.

One was to catch `Exception` instead of the specific types. This gets rid of the duplicate code, however, this isn't a good approach because it catches other exceptions too.

The other approach would be to refactor the duplicate code into a helper method.

The Java language designers recognised that this situation is an undesirable trade off. In Java 7, they introduced the ability to catch multiple exceptions in the same catch block, also known as multi-catch. Now we have an elegant solution to the problem:

```
public static void main(String[] args) {
    try {
        Path path = Paths.get("File.txt");
        String text = new String(Files.readAllBytes(path));
        LocalDate date = LocalDate.parse(text);
        System.out.println(date);
    } catch (DateTimeParseException | IOException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}
```

This is much better. There's no duplicate code, the common logic is all in one place, and the logic is exactly where we would expect to find it.

Multi-catch is like a regular catch clause, except two or more exception types are specified separated by a pipe. The pipe is also used as the “or” operator, making it easy to remember that you can use either/or of the exception types. Notice how there is only one variable name in the catch clause. Java is saying that the variable named e can be of either exception type.

Multi-catch is designed to be used for exceptions that aren’t related, and Java prevents you from specifying redundant types in a multi-catch. For example you can’t multi-catch Exception and SQLException as it would always be Exception. The compiler picks this up.

For multi-catch blocks you can no longer reassign the exception variable (usually e) to a new exception. That’s because Java can’t know what data type the variable is until run time. This was a bit of a bad practice anyway so, no loss really.

Try-With-Resources

Multi-catch allows you to write code without duplication. Another problem arises with duplication in finally blocks. It is important to close resources when you are finished with them. A resource could be anything you have to open and close but is commonly a file or database

Imagine that you want to write a simple method to read the first line of one file and write it to another file.

Here’s some pre-Java 7 code to do this:

```
public void oldApproach(Path path1, Path path2) throws IOException {
    BufferedReader in = null;
    BufferedWriter out = null;
    try {
        in = Files.newBufferedReader(path1);
        out = Files.newBufferedWriter(path2);
        out.write(in.readLine());
    } finally {
        if (in != null) in.close();
        if (out != null) out.close();
    }
}
```

That’s twelve lines of code to do something quite simple, and we don’t even deal with catching the exception.

Switching to the try-with-resources syntax introduced in Java 7, it can be rewritten as follows:

```
public void newApproach(Path path1, Path path2) throws IOException {
    try (BufferedReader in = Files.newBufferedReader(path1);
         BufferedWriter out = Files.newBufferedWriter(path2)) {
        out.write(in.readLine());
    }
}
```

The new version has half as many lines. There is no longer code just to close resources. The new try-with-resources statement automatically closes all resources opened in the try clause. This feature is also known as automatic resource management, because Java automatically takes care of the closing.

In the following sections, we will look at the try-with-resources syntax and how to indicate a resource can be automatically closed. We will introduce suppressed exceptions.

You might have noticed that there is no finally block in the try-with-resources code. Previously, you learned that a try statement must have one or more catch blocks or a finally block. This is still true. The finally clause exists implicitly. You just don't have to type it.

Notice that one or more resources can be opened in the try clause. Also, notice that parentheses are used to list those resources and semicolons are used to separate the declarations. This works just like declaring multiple indexes in a for loop.

A try-with-resources block is still allowed to have catch and/or finally blocks. They are run in addition to the implicit one. The implicit finally block runs before any programmer-coded ones.

The resources created in the try clause are only in scope within the try block. This is another way to remember that the implicit finally runs before any catch/finally blocks that you code yourself. The implicit close has run already, and the resource is no longer available.

AutoCloseable

You can't just put any random class in a try-with-resources statement. Java commits to closing automatically any resources opened in the try clause. Here we tell Java to try to close the AppStream class when we are finished with it:

```
public class AppStream {  
    public static void main(String[] args) {  
        try (AppStream a = new AppStream ()) { // DOES NOT COMPILE  
            System.out.println(t);  
        }  
    }  
}
```

Java doesn't allow this. It has no idea how to close an AppStream. Java informs us of this fact with a compiler error.

In order for a class to be created in the try clause, Java requires it to implement an interface called AutoCloseable.

The AutoCloseable interface has only one method to implement:

```
public void close() throws Exception;
```

Your class doesn't have to throw an Exception. An overriding method is allowed to declare more specific exceptions than the parent or even none at all. By declaring

Exception, the AutoCloseable interface is saying that implementers may throw any exceptions they choose.

The following shows what happens when an exception is thrown during the attempt to close the resource:

```
public class AppStream implements AutoCloseable {
    public void close() throws Exception {
        throw new Exception("AppStream unable to close");
    }
    public static void main(String[] args) {
        try (AppStream a = new AppStream()) { // DOES NOT COMPILE
            System.out.println("testing...");
```

This try-with-resources statement throws a checked exception. You know that checked exceptions need to be handled or declared. Tricky isn't it? This is something that you need to watch out for in your code.

If the main() method declared an Exception, this code would compile.

Java strongly recommends that close() not actually throw Exception. It is better to throw a more specific exception. Java also recommends to make the close() method ‘idempotent’.

Idempotent means that the method can be called multiple times without any side effects or undesirable behaviour on subsequent runs. For example, it shouldn't throw an exception the second time or change state or the like. Both these negative practices are allowed. They are merely discouraged.

Closeable

The AutoCloseable interface was introduced in Java 7. Before that, another interface existed called Closeable. It was similar to what the language designers wanted, with the following exceptions:

- Closeable restricts the type of exception thrown to IOException.
- Closeable requires implementations to be idempotent.

The language designers emphasise backward compatibility.

Because changing the existing interface was undesirable, they made a new one called AutoCloseable. This new interface is less strict than Closeable.

As Closeable meets the requirements for AutoCloseable, it started implementing AutoCloseable when the latter was introduced.

Suppressed Exceptions

What happens if the close() method throws an exception? If the AppStream resource doesn't close, the network connection would be left open. Clearly we need to handle such a condition for security reasons if nothing else.

We already know that the resources are closed before any programmer-coded catch blocks are run. This means that we can catch the exception thrown by close() if we wish. Alternatively, we can allow the caller to deal with it. Just like a regular exception, checked exceptions must be handled or declared. Runtime exceptions do not need to be acknowledged.

Now that we have an extra step of closing resources in the try, it is possible for multiple exceptions to get thrown. Each close() method can throw an exception in addition to the try block itself.

```
public class Suppressed {
    public static void main(String[] args) {
        try (One one = new One()) {
            throw new Exception("Try");
        } catch (Exception e) {
            System.out.println(e.getMessage());
            for (Throwable t : e.getSuppressed()) {
                System.out.println("suppressed: " + t);
            }
        }
    }
}

class One implements AutoCloseable {
    public void close() throws IOException {
        throw new IOException("Closing");
    }
}
```

We know that after the exception in the try block gets thrown, the try-with-resources still calls close() and the catch block catches one of the exceptions. Running the code prints:

```
Try
suppressed: java.io.IOException: Closing
```

This tells us the exception we thought we were throwing still gets treated as most important. Java also adds any exceptions thrown by the close() methods to a suppressed array in that main exception. The catch block or caller can deal with any or all of these.

If we remove the line that throws the Exception in main(), the code just prints Closing.

In other words, the exception thrown in close() doesn't always get suppressed. It becomes the main exception if there isn't already an existing exception to add it to.

The following two rules apply to the order in which code runs in a try-with-resources statement:

- Resources are closed after the try clause ends and before any catch/finally clauses.
- Resources are closed in the reverse order from which they were created

Java Assertions

An assertion is a Boolean expression that you place at a point in your code where you expect something to be true. The English definition of the word assert is to state that something is true, which means that you assert that something is true. An assert statement contains this statement along with an optional String message.

An assertion allows for detecting defects in the code. You can turn on assertions for testing and debugging while leaving them off when your program is in production.

Why assert something when you know it is true? It is only true when everything is working properly. If the program has a defect, it might not actually be true. Detecting this earlier in the process lets you know something is wrong.

In the following sections, we cover the syntax for using an assertion, how to turn them on/off, and common uses of assertions.

Assert Statement

The syntax for an assert statement has two forms:

```
assert boolean_expression;
assert boolean_expression: error_message;
```

The boolean expression must evaluate to true or false. It can be inside optional parentheses. The optional error message is a String used as the message for the `AssertionError` that is thrown.

That's right. An assertion throws an `AssertionError` if it is false. Since programs aren't supposed to catch an Error, this means that assertion failures are fatal and end the program.

The three possible outcomes of an assert statement are as follows:

- If assertions are disabled, Java skips the assertion and goes on in the code.
- If assertions are enabled and the boolean expression is true, then our assertion has been validated and nothing happens. The program continues to execute in its normal manner.
- If assertions are enabled and the boolean expression is false, then our assertion is invalid and a `java.lang.AssertionError` is thrown.

Presuming assertions are enabled, an assertion is a shorter/better way of writing the following:

```
if (!boolean_expression) throw new AssertionException();
```

The assert syntax is easier to read. Remember when we said a developer shouldn't be throwing an Error? With the assert syntax, you aren't. Java is throwing the Error.

Suppose that we enable assertions by running the following example with the command

```
java -ea Assertions;
```

```
public class Assertions {  
    public static void main(String[] args) {  
        int numGuests = -5;  
        assert numGuests > 0;  
        System.out.println(numGuests);  
    }  
}
```

We made a typo in the code. We intended for there to be five guests and not minus five. The assertion statement detects this problem. Java throws the `AssertionError` at this point. The next line never runs since an error was thrown.

The program ends with a stack trace similar to this:

```
Exception in thread "main" java.lang.AssertionError at  
asserts.Assertions.main(Assertions.java: 7)
```

If we run the same program using the command line `java Assertions`, we get a different result. The program prints `-5`. Now, in this example, it is pretty obvious what the problem is since the program is only seven lines.

But in a more complicated program, knowing the state of variables at a specific point in the code is more useful.

Enabling Assertions

By default, `assert` statements are ignored by the JVM at runtime. To enable assertions, use the `-enableassertions` flag on the command line:

```
java -enableassertions MyClass
```

You can also use the shortcut `-ea` flag:

```
java -ea MyClass
```

Using the `-enableassertions` or `-ea` flag without any arguments enables assertions in all classes except system classes.

System classes are classes that are part of the Java runtime. You can think of them as the classes that come with Java.

You can also enable assertions for a specific class or package. For example, the following command enables assertions only for classes in the `com.stayahead.demo` package and any subpackages:

```
java -ea: com.stayahead.demo... my.programs.Main
```

The ellipsis (...) means any class in the specified package or subpackages. You can also enable assertions for a specific class:

```
java -ea: com.stayahead.demo.TestAssert my.programs.Main
```

You can disable assertions using the `-disableassertions` (or `-da`) flag for a specific class or package that was previously enabled.

For example, the following command enables assertions for the com.stayahead.demo package but disables assertions for the TestAssert class:

```
j ava -ea: com. stayahead. demo. . . -da: com. stayahead. demo. TestAssert  
my. programs. Mai n
```

Reasons to Use Assertions

You can use assertions for many reasons, including the following.

- **Internal Invariants** You assert that a value is within a certain constraint. `assert x < 0` is an example of an internal invariant.
- **Class Invariants** You assert the validity of an object's state. Class invariants are typically private methods within the class that return a boolean. The upcoming Rectangle class demonstrates a class invariant.
- **Control Flow Invariants** You assert that a line of code you assume is unreachable is never reached. The upcoming TestSeasons class demonstrates a control flow invariant.
- **Preconditions** You assert that certain conditions are met before a method is invoked.
- **Post Conditions** You assert that certain conditions are met after a method executes successfully.

Summary

An exception indicates that something unexpected happened.

Subclasses of `java.lang.Error` are exceptions that a program should not attempt to handle.

Subclasses of `java.lang.RuntimeException` are runtime (unchecked) exceptions.

Subclasses of `java.lang.Exception` that do not subclass `java.lang.RuntimeException` are checked exceptions. Java requires checked exceptions to be handled or declared.

If a try statement has multiple catch blocks, at most one catch block can run. Java looks for an exception that can be caught by each catch block in the order in which they appear, and the first match is run. Then execution continues after the try statement to the finally block if present. If both catch and finally throw an exception, the one from finally gets thrown. Common checked exceptions include `ParseException`, `IOException`, and `SQLException`.

Multi-catch allows catching multiple exception types in the same catch block. The types are separated with a pipe (|). The multiple exception types are not allowed to have a subclass/superclass relationship. The variable in a multi-catch expression is effectively final.

Try-with-resources allows Java to take care of calling the `close()` method. This is called automatic resource management. Objects instantiated in the try clause must implement the `AutoCloseable` interface. This interface has a single method `close()` and can throw any type of `Exception`.

Unlike traditional try statements, try-with-resources does not require a catch or finally block to be present.

If the try clause and one or more of the `close()` methods throw an exception, Java uses suppressed exceptions to keep track of both. Similarly, if multiple `close()` methods throw an exception, the first one is the primary exception and the others are suppressed exceptions. `getSuppressed()` allows these exceptions to be retrieved.

An assertion is a boolean expression placed at a particular point in your code where you think something should always be true. A failed assertion throws an `AssertionError`. Assertions should not change the state of any variables.

The `-ea` and `-enableassertions` flags turn on assertions. The `-da` and `-disableassertions` flags turn them off for specific classes or packages where a whole package has been enabled.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 7

Concurrency

Introduction

Computers often depend on reading and writing data to external resources such as files and databases. Unfortunately, as compared to internal CPU processing, these disk or network operations tend to be extremely slow. So slow, in fact, that if your computer's operating system were to stop and wait for every disk or network operation to finish, your users would never get any work done and their computers would appear to freeze or lock up constantly.

Luckily, all modern operating systems support what is known as multi-threaded processing. The 'multi' bit has become much more realistic now that most desktop PCs and servers are 'multi-core', commonly having four processors capable of executing code simultaneously.

The idea behind multi-threaded processing is to allow an application or group of applications to execute multiple tasks at the same time. This allows tasks waiting for other resources to give way to other processing requests.

Of course you have to design your code well to achieve this or the processors will just be working very fast only to get stuck in some badly written synchronous processing code that limits the system by causing bottlenecks.

Since its early days, Java has supported multi-threading programming using the Thread class.

When Java 5 was released, the Concurrency API was introduced in the java.util.concurrent package.

This included numerous classes for performing complex thread-based tasks. The idea was simple: managing complex thread interactions is quite difficult for even the most skilled developers; therefore a set of reusable features was created. The Concurrency API has grown over the years to include numerous classes and frameworks to assist you in developing complex, multi-threaded applications.

In this chapter, we will examine the concept of threads and provide numerous ways to manage threads using the Concurrency API.

Threads and concurrency tend to be one of the more challenging topics for many programmers to grasp, as problems with threads can be frustrating even for veteran developers to understand. In practice, concurrency issues are among the most difficult problems to diagnose and resolve. But they are also very rewarding in terms of performance of your code as well as the satisfaction of a job well done.

Threads

Let's start this chapter by reviewing common terminology associated with threads.

A thread is the smallest unit of execution that can be scheduled by the operating system.

A process is a group of associated threads that execute in the same, shared environment. It follows, then, that a single-threaded process is one that contains exactly one thread, whereas a multi-threaded process is one that contains one or more threads.

The applications you have written and the example you have seen so far on this course have been single-threaded.

However, you probably don't remember starting any threads yourself, so where did the single thread come from?

The answer is that when the JVM loads the main method in your starting class, it creates a thread to run the code. This is the 'main' thread. If you get an exception, the JVM's message tells you there was an 'Exception in thread "main"'

By saying that the threads execute in a shared environment, we mean that the threads in the same process share the same memory space and can communicate directly with one another.

You will see how static variables can be useful for performing complex, multi-threaded tasks. Remember that static methods and variables are defined on a single class object that all instances share. For example, if one thread updates the value of a static object, then this information is immediately available for other threads within the process to read.

In this chapter, we will talk a lot about tasks and their relationships to threads. A task is a single unit of work performed by a thread.

Throughout this chapter, a task will commonly be implemented as a lambda expression. A thread can complete multiple independent tasks but only one task at a time.

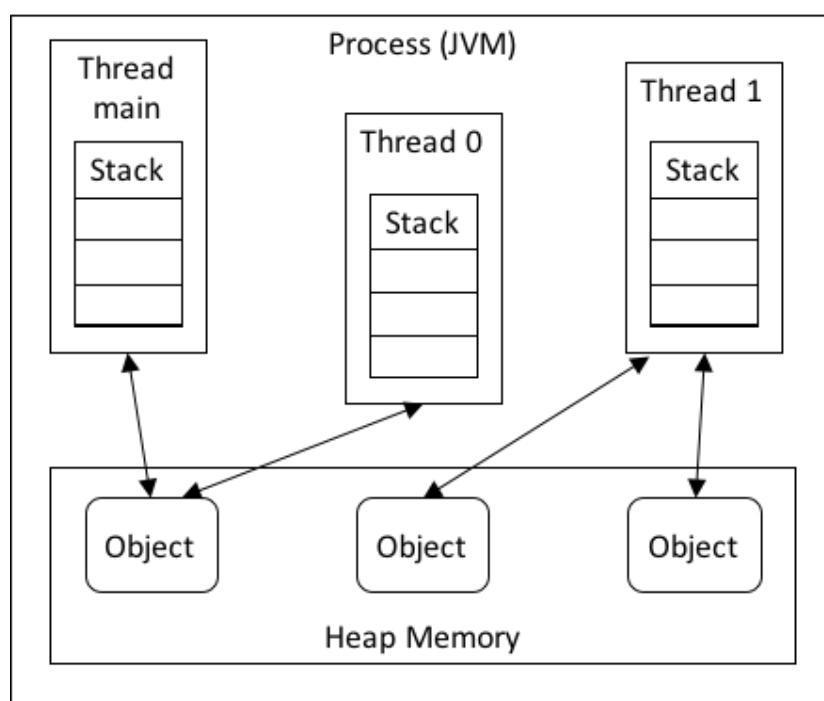


Figure 7-1 Threads in a Process

The process model shows a single process with three threads.

The next diagram shows how they are mapped to an arbitrary number of CPUs available within the system. Keep this diagram in mind as we discuss task scheduling in the next section.

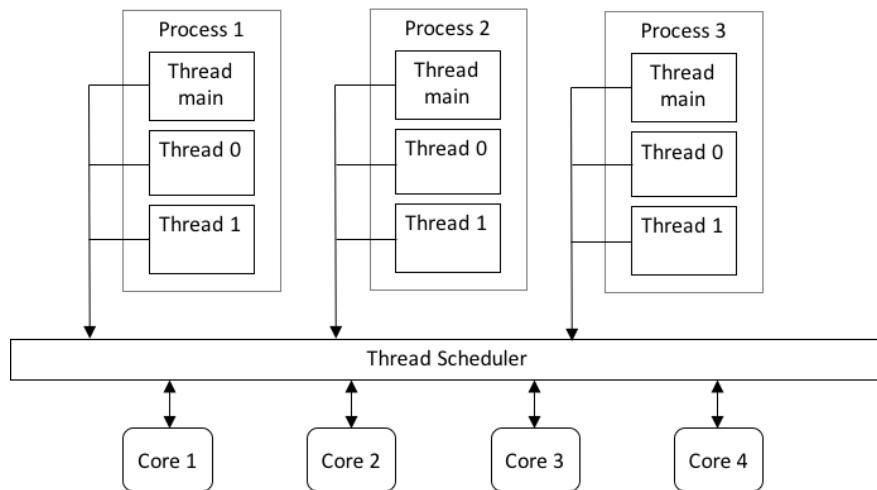


Figure 7-2 Threads and Multi-processing

Thread Types

It might surprise you that all Java applications, including all the ones that we have presented in this book, are all multi-threaded. Even a simple Java application that prints Hello World to the screen is multi-threaded. To help you understand this, we introduce the concepts of system threads and user-defined threads.

A system thread is created by the JVM and runs in the background of the application. For example, the garbage-collection thread is a system thread that is created by the JVM and runs in the background, helping to free memory that is no longer in use. For the most part, the execution of system-defined threads is invisible to the application developer. When a system-defined thread encounters a problem, and cannot recover, such as running out of memory, it generates a Java Error, as opposed to an Exception.

A user-defined thread, on the other hand, is one created by the application developer to accomplish a specific task. All the applications that we have created up to this point have been multi-threaded, but they contained only one user-defined thread, which calls the `main()` method. For simplicity, we commonly refer to threads that contain only a single user-defined thread as a single-threaded application, since we are not often interested in the system threads.

Thread Concurrency

As previously stated, multi-threaded processing allows operating systems to execute threads at the same time. The property of executing multiple threads and processes at the same time is referred to as concurrency. Of course, with a single-core CPU system, only one task is actually executing at a given time. Even in multi-core or multi-CPU systems, there are often far more threads than CPU processors available. How does the system decide what to execute when there are multiple threads available?

Operating systems use a thread scheduler to determine which threads should be currently executing, as shown in Figure 7-2.

For example, a thread scheduler may employ a round-robin schedule in which each available thread receives an equal number of CPU cycles with which to execute, with threads visited in a circular order. If there are 12 available threads, they might each get 100 milliseconds in which to execute, with the process returning to the first thread after the last thread has executed.

When a thread's allotted time is complete but the thread has not finished processing, a context switch occurs. A context switch is the process of storing a thread's current state and later restoring the state of the thread to continue execution. Be aware that there is often a cost associated with a context switch by way of lost time saving and reloading a thread's state.

A thread can interrupt or supersede another thread if it has a higher thread priority than the other thread. A thread priority is a numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executing. In Java, thread priorities are specified as integer values.

The Thread class includes three important static constants that help the programmer request that a thread is given a certain priority.

Thread priority is an integer instance variable with a value between 1 and 10. By default, user-defined threads receive a thread priority value of Thread.NORM_PRIORITY (equal to 5).

If you have a thread that must be executed right away, you can increase this value to 6 or higher or use the Thread.MAX_PRIORITY value, which equals 10. If two threads have the same priority, the thread scheduler will arbitrarily choose the one to process first.

Runnable Interface

java.lang.Runnable, or Runnable for short, is a functional interface that takes no arguments and returns no data. The following is the definition of the Runnable interface:

```
@FunctionalInterface public interface Runnable {  
    void run();  
}
```

The Runnable interface is commonly used to define the work a thread will execute, separate from the main application thread. We will be relying on the Runnable interface throughout this chapter, especially when we discuss applying parallel operations to streams.

The following lambda expressions each implement the Runnable interface:

```
() -> System.out.println("Hello World") () -> {int i=10; i++;}  
() -> {}  
() -> {}
```

Notice that all of these lambda expressions start with a set of empty parentheses, (). Also, note that none of them return a value.

Any lambda that either takes parameters or returns a value is incompatible with Runnable. The following lambdas, while valid for other functional interfaces, are examples that are not compatible with Runnable:

```
(() -> "")  
(() -> 5)  
(() -> {return new Object();})
```

These examples are invalid Runnable expressions because they each return a value.

Creating Threads

The simplest way to execute a thread is by using the `java.lang.Thread` class, or `Thread` for short. Executing a task with `Thread` is a two-step process. First you define the `Thread` with the corresponding task to be done. Then you start the task by using the `Thread.start()` method.

As you will see later in the chapter, Java does not provide any guarantees about the order in which a thread will be processed once it is started. It may be executed immediately or delayed for later execution.

Defining the task, or work, that a `Thread` instance will execute can be done two ways in Java:

- Provide a `Runnable` object or lambda expression to the `Thread` constructor.
- Create a class that extends `Thread` and overrides the `run()` method.

The following are examples of these techniques:

```
public class PrintData implements Runnable {  
    public void run() {  
        for(int i=0; i<3; i++) {  
            System.out.println("Printing record: " + i);  
        }  
    }  
    public static void main(String[] args) {  
        (new Thread(new PrintData())).start();  
    }  
}  
  
public class ReadInventoryThread extends Thread {  
    public void run() {  
        System.out.println("Reading inventory");  
    }  
    public static void main(String[] args) {  
        (new ReadInventoryThread()).start();  
    }  
}
```

The first example creates a `Thread` using a `Runnable` instance, while the second example uses the less common practice of extending the `Thread` class and overriding the `run()` method.

Anytime you create a Thread instance, make sure that you remember to start the task with the Thread.start() method. This starts the task in a separate operating system thread.

In general, you should extend the Thread class only under very specific circumstances, such as when you are creating your own priority-based thread. In most situations, you should implement the Runnable interface rather than extend the Thread class.

We conclude our discussion of the Thread class here.

You are strongly encouraged to try to use the Concurrency API to create and manage Thread objects wherever possible rather than creating and running threads yourself. However, it is important to understand how they work. We will be looking at the Concurrency API in detail shortly.

Polling with Sleep

Often you need a thread to poll for a result to finish. Polling is the process of checking data at some fixed interval. For example, let's say that you have a thread that modifies a shared static counter value and your main() thread is waiting for the thread to increase the value above 100, as shown in the following class:

```
public class CheckResults {
    private static int counter = 0;
    public static void main(String[] args) {
        new Thread(() -> {
            for(int i=0; i<500; i++){
                CheckResults.counter++;
            }
        }).start();
        while(CheckResults.counter<100) {
            System.out.println("Not reached yet");
        }
        System.out.println("Reached!");
    }
}
```

How many times do you think the while() loop in this code will execute and output “Not reached yet”?

It is not possible to predict this with any degree of accuracy. It could output zero, ten, or a million times. If our thread scheduler is particularly ungenerous to our thread, it could operate infinitely!

Using a while() loop to check for data without some kind of delay is considered a very bad coding practice as it ties up CPU resources for no reason.

We can improve this result by using the Thread.sleep() method to implement polling. The Thread.sleep() method requests the current thread of execution rest for a specified number of milliseconds. When used inside the body of the main() method, the thread associated with the main() method will pause, while the separate thread will continue to run.

Compare the previous implementation with the following one that uses Thread.sleep():

```
public class CheckResults {
    private static int counter = 0;
    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
            for(int i=0; i<500; i++) CheckResults.counter++;
        }).start();
        while(CheckResults.counter<100) {
            System.out.println("Not reached yet");
            Thread.sleep(1000); // 1 SECOND
        }
        System.out.println("Reached!");
    }
}
```

In this example, the call to sleep() delays processing for 1,000 milliseconds (1 second) at the end of the loop.

While this may seem like a small amount, we have now prevented a possibly infinite loop from executing and locking up our main program.

Notice that we also modified the signature of the main method to declare that Thread.sleep() throws the checked InterruptedException. Alternatively, we could have wrapped each call to the Thread.sleep() method in a try/catch block.

How many times does the while() loop execute in this revised class? This is still unknown. You could try some experiments to see what the average is.

This does not necessarily solve the problem though. Polling does prevent the CPU from being overwhelmed with a potentially infinite loop, it does not guarantee when the loop will terminate. For example, the separate thread could be losing CPU time to a higher-priority process, resulting in a large number of executions of the while() loop before it finishes.

Another issue to be concerned about is the shared counter variable. What if one thread is reading the counter variable while another thread is writing it? The thread reading the shared variable may end up with an invalid or incorrect value. We will discuss these issues in detail in the upcoming section on synchronisation.

Using ExecutorService

ExecutorService and executors are classes that help to execute threads. They are both pronounced with the stress on the ‘u’ rather than the ‘e’.

With the Concurrency API, Java introduced the ExecutorService, which creates and manages threads for you.

To use it you first obtain an instance of an ExecutorService interface, and then you send the service tasks to be processed.

The framework includes numerous useful features, such as thread pooling and scheduling, which would be cumbersome for you to implement in every project. Therefore, it is recommended that you use this framework anytime you need to create and execute a separate task, even if you need only a single thread.

Single-Thread Executor

Since ExecutorService is an interface, how do you obtain an instance of it? The Concurrency API includes the Executors factory class that can be used to create instances of the ExecutorService object.

As you saw earlier in the course, the factory pattern is a creational pattern in which the underlying implementation details of the object creation are decoupled from the code that uses them. This means you code to the interface rather than the implementation. The factory method returns an appropriate implementation of the interface without the caller needing to know which implementation is returned. It is guaranteed to provide all the interface’s methods.

Here’s a simple example using the newSingleThreadExecutor() method to obtain an ExecutorService instance and the execute() method to perform asynchronous tasks:

```
import java.util.concurrent.*;
public class InfoSource {
    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newSingleThreadExecutor();
            System.out.println("begin");
            service.execute(() -> System.out.println("Printing stock"));
            service.execute(() -> {for(int i=0; i<3; i++)
                System.out.println("Printing record: " + i); });
            service.execute(() -> System.out.println("Printing report"));
            System.out.println("end");
        } finally {
            if(service != null) service.shutdown();
        }
    }
}
```

This is similar to some of the earlier examples but now uses Runnable lambda expressions and an ExecutorService instance.

In this example, we used the newSingleThreadExecutor() method, which is the simplest ExecutorService that we could create.

Unlike our earlier example, in which we had three extra threads for newly created tasks, this example uses only one, which means that the threads will order their results. For example, the following is a possible output for this code snippet:

```
begin
Printing stock
Printing record: 0
end
Printing record: 1
Printing record: 2
Printing report
```

With a single-thread executor, results are guaranteed to be executed in the order in which they are added to the executor service.

However, that the end text is output while our thread executor tasks are still running. This is because the main() method is still an independent thread (main thread) not connected with the ExecutorService, and it can perform tasks while the other thread is running.

Executor Shutdown

Once you have finished using a thread executor, it is important that you call the shutdown() method. A thread executor creates a new Thread for the first task that is executed, so failing to call shutdown() will result in your application never terminating.

This diagram shows the life cycle of an ExecutorService object.

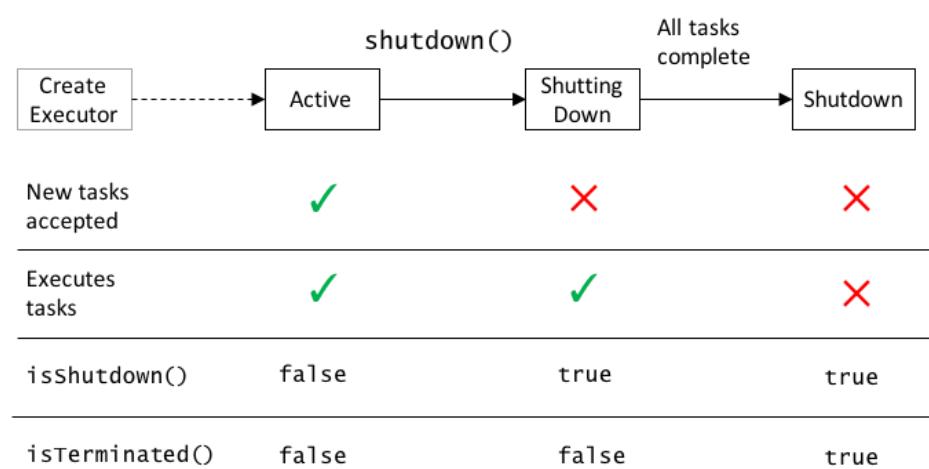


Figure 7-1 Lifecycle of Thread Executor

The shutdown process for a thread executor involves first rejecting any new tasks submitted to the thread executor while continuing to execute any previously submitted tasks.

During this time, calling `isShutdown()` will return true, while `isTerminated()` will return false. If a new task is submitted to the thread executor while it is shutting down, a `RejectedExecutionException` will be thrown.

Once all active tasks have been completed, `isShutdown()` and `isTerminated()` will both return true.

Submitting Tasks

There are multiple ways to submit tasks to an `ExecutorService`. The first method you saw above, `execute()`, is inherited from the `Executor` interface, which the `ExecutorService` interface extends. The `execute()` method takes a `Runnable` lambda expression or instance and completes the task asynchronously.

Because the return type of the method is `void`, it does not tell us anything about the result of the task. It is considered a “fire-and-forget” method, as once it is submitted, the results are not directly available to the calling thread.

Fortunately, the Java designers added other `submit()` methods to the `ExecutorService` interface, which, like `execute()`, can be used to complete tasks asynchronously.

Java provides a `Future<V>` interface to represent the result of an asynchronous computation.

Like `Optional<T>` that you saw earlier, methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

The result can only be retrieved from a `Future<V>` object using the method `get()` when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the `cancel` method.

Additional methods are provided to determine if the task completed normally or was cancelled.

Because `Runnable` always returns `void`, it can only be used to perform tasks that do not have a result. The `submit()` methods accept an object that implements `Callable<V>` that do return a value.

Unlike `execute()`, the `submit()` methods return a `Future` object that can be used to determine if the task is complete. It can also be used to return a generic result object after the task has been completed.

This table shows the five main methods, including execute() and two submit() methods, which you need to know to use ExecutorService objects effectively:

Method	Description
void execute(Runnable command)	Executes a Runnable task at some point in the future at the discretion of the thread scheduler
Future<?> submit(Runnable task)	Executes a Runnable task at some point in the future and returns a Future representing the task
<T> Future<T> submit(Callable<T> task)	Executes a Callable task at some point in the future and returns a Future representing the pending results of the task
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException	Executes the given tasks, synchronously returning the results of all tasks as a Collection of Future objects, in the same order they were in the original collection
<T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException	Executes the given tasks, synchronously returning the result of one of finished tasks, cancelling any unfinished tasks

Submitting Task Collections

The last two methods listed in the table are invokeAll() and invokeAny().

Both of these methods take a Collection object containing a list of tasks to execute. Both of these methods also execute synchronously.

By synchronous, we mean that unlike the other methods used to submit tasks to a thread executor, these methods will wait until the results are available before returning control to the enclosing program.

The invokeAll() method executes all tasks in a provided collection and returns a List of ordered Future objects, with one Future object corresponding to each submitted task, in the order they were in the original collection.

Even though Future.isDone() returns true for each element in the returned List, a task could have completed normally or thrown an exception.

The invokeAny() method executes a collection of tasks and returns the result of one of the tasks that successfully completes execution, cancelling all unfinished tasks. While the first task to finish is often returned, this behaviour is not guaranteed, as any completed task can be returned by this method.

Finally, the invokeAll() method will wait indefinitely until all tasks are complete, while the invokeAny() method will wait indefinitely until at least one task completes. The ExecutorService interface also includes overloaded versions of invokeAll() and invokeAny() that take a timeout value and TimeUnit parameter. We will see how to use these types of parameters in the next section when discussing the Future class.

Retrieving Results

How do we know when a task submitted to an ExecutorService is complete?

As mentioned in the last section, the submit() method returns a java.util.concurrent.Future<V> object, or Future<V> for short, that can be used to determine this result:

```
Future<?> future = service.submit(() -> System.out.println("Test"));
```

This table shows the methods used for managing Future<V> objects:

Method	Description
boolean isDone()	Returns true if the task was completed, threw an exception, or was cancelled
boolean isCancelled()	Returns true if the task was cancelled before it completely normally
boolean cancel()	Attempts to cancel execution of the task
V get()	Retrieves the result of a task, waiting endlessly (blocking) if it is not yet available
V get(long timeout, TimeUnit unit)	Retrieves the result of a task, waiting the specified amount of time. If the result is not ready by the time the timeout is reached, a checked TimeoutException will be thrown

The following is an updated version of our earlier polling example CheckResults class, which uses a Future instance to poll for the results:

```
import java.util.concurrent.*;
public class CheckResults {
    private static int counter = 0;
    public static void main(String[] args) throws InterruptedException,
                                                ExecutionException {
        ExecutorService service = null;
        try {
```

```
service = Executors.newSingleThreadExecutor();
Future<?> result = service.submit(() ->
    { for(int i=0; i<500; i++) CheckResults.counter++; });
result.get(10, TimeUnit.SECONDS);
System.out.println("Reached!");
} catch (TimeoutException e) {
    System.out.println("Not reached in time");
} finally {
    if(service != null)
        service.shutdown();
}
}
```

This example is similar to the earlier polling example, but it does not use the Thread class directly.

It is really the essence of the Concurrency API to do complex things with threads without using the Thread class directly.

This code also waits at most 10 seconds, throwing a `TimeoutException` if the task is not done.

What is the return value of this task? As Future<V> is a generic class, the type V is determined by the return type of the Runnable method. Since the return type of Runnable.run() is void, the get() method always returns null. In the next section, you will see that there is another task class compatible with ExecutorService that supports other return types.

As you saw in the previous example, the `get()` method can take an optional value and enum type `java.util.concurrent.TimeUnit`.

Callable Interface

When the Concurrency API was released in Java 5, the new `java.util.concurrent.Callable` interface was added, or Callable for short, which is similar to `Runnable` except that its `call()` method returns a value and can throw a checked exception.

As you may remember from the definition of Runnable, the `run()` method returns void and cannot throw any checked exceptions.

Along with Runnable, Callable was also made a functional interface in Java 8. The following is the definition of the Callable interface:

```
@FunctionalInterface  
public interface Callable<V> {  
    V call() throws Exception;  
}
```

The Callable interface was introduced as an alternative to the Runnable interface, since it allows more details to be retrieved easily from the task after it is completed.

The ExecutorService includes an overloaded version of the submit() method that takes a Callable object and returns a generic Future<T> object.

Unlike Runnable, in which the get() methods always return null, the get() methods on a Future object return the matching generic type or null.

Let's take a look at an example using Callable:

```
import java.util.concurrent.*;
public class Add {
    public static void main(String[] args) throws InterruptedException,
                                                ExecutionException {
        ExecutorService service = null;
        try {
            service = Executors.newSingleThreadExecutor();
            Future<Integer> result = service.submit(() -> 10 + 32);
            System.out.println(result.get());
        } finally {
            if(service != null){
                service.shutdown();
            }
        }
    }
}
```

We can now retrieve and print the output of the Callable results, 42 in this example.

The results could have also been obtained using Runnable and some shared, possibly static, object, although this solution that relies on Callable is a lot simpler and easier to follow.

Since Callable supports a return type when used with ExecutorService, it is often preferred over Runnable when using the Concurrency API, particularly when splitting a task and recombining the results.

They are more or less interchangeable in situations where the lambda does not throw an exception and there is no return type. If the code will throw an exception, the Runnable interface needs an embedded try/catch block. Here is an example with an ExecutorService called service:

```
service.submit(() -> {Thread.sleep(1000); return null;});
service.submit(() -> {Thread.sleep(1000);});
```

The first line will compile, while the second line will not.

Remember that Thread.sleep() throws a checked InterruptedException. Since the first lambda expression has a return type, the compiler treats this as a Callable expression that supports checked exceptions. The second lambda expression does not return a value; therefore, the compiler treats this as a Runnable expression. Since Runnable methods do not support checked exceptions, the compiler will report an error.

Waiting for Tasks to Finish

After you have submitted a set of tasks to a thread executor, it is often necessary to wait for the results.

As you saw in the previous sections, one solution is to call `get()` on each `Future` object returned by the `submit()` method.

If we don't need the results of the tasks and are finished using the thread executor, there is a simpler approach.

First, we shut down the thread executor using the `shutdown()` method.

Next, we use the `awaitTermination(long timeout, TimeUnit unit)` method available for all thread executors.

This method waits the specified time to complete all tasks, returning sooner if all tasks finish or an `InterruptedException` is detected.

You can see an example of this in the following code:

```
ExecutorService service = null;
try {
    service = Executors.newSingleThreadExecutor();
    // Add tasks to the thread executor
    ...
} finally {
    if(service != null){
        service.shutdown();
    }
}
if(service != null) {
    service.awaitTermination(1, TimeUnit.MILLISECONDS);
    // Check whether all tasks are finished
    if(service.isTerminated())
        System.out.println("All tasks finished");
    else
        System.out.println("At least one task is still running"); }
```

In this example, we submit multiple tasks to the thread executor and then shut it down and wait up to one minute for the results.

Notice the call to `isTerminated()` after the `awaitTermination()` method finishes is to check whether all tasks are actually finished.

Task Scheduling

Sometimes you may need to schedule a task to happen at some future time. We might even need to schedule the task to happen repeatedly, at some set interval. For example, imagine that we want to check the average call-centre waiting time once an hour and increase or decrease the number of call handlers needed. The `ScheduledExecutorService`, which is a subinterface of `ExecutorService`, can be used for just such a task.

Like `ExecutorService`, we obtain an instance of `ScheduledExecutorService` using a factory method in the `Executors` class, as shown in the following snippet:

```
ScheduledExecutorService service =
    Executors.newSingleThreadScheduledExecutor();
```

Note that we could implicitly cast an instance of ScheduledExecutorService to ExecutorService, although doing so would remove access to the scheduled methods that we want to use.

This table shows the various methods available for scheduling tasks:

Method	Description
schedule(Callable<V> callable, long delay, TimeUnit unit)	Creates and executes a Callable task after the given delay
schedule(Runnable command, long delay, TimeUnit unit)	Creates and executes a Runnable task after the given delay
scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)	Creates and executes a Runnable task after the given initial delay, creating a new task every period value that passes
scheduleAtFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Creates and executes a Runnable task after the given initial delay and subsequently with the given delay between the termination of one execution and the commencement of the next

These methods are among the most convenient in the Concurrency API, as they perform relatively complex tasks with a single line of code. The code is therefore easier to understand.

The delay and period parameters rely on the TimeUnit argument to determine the format of the value, such as seconds or milliseconds.

The first two schedule() methods in the table above:

1. take a Callable or Runnable, respectively
2. perform the task after some delay
3. return a ScheduledFuture<V> instance

ScheduledFuture<V> is identical to the Future<V> class, except that it includes a getDelay() method that returns the delay set when the process was created.

The following code uses the schedule() method with Callable and Runnable tasks:

```

ScheduledExecutorService service =
    Executors.newSingleThreadScheduledExecutor();
Runnable task1 = () -> System.out.println("Call average: ");
Callable<Float> task2 = () -> 4.5;
Future<?> result1 = service.schedule(task1, 10, TimeUnit.SECONDS);
Future<?> result2 = service.schedule(task2, 8, TimeUnit.MINUTES);

```

The first task is scheduled 10 seconds in the future, whereas the second task is scheduled 8 minutes in the future.

The last two methods in the table above might seem a little confusing at first glance. Conceptually, they are very similar as they both perform the same task repeatedly, after completing some initial delay. The difference is related to the timing of the process and when the next task starts.

The `scheduleAtFixedRate()` method creates a new task and submits it to the executor every period, regardless of whether or not the previous task finished. The following example executes a Runnable task every minute, following an initial five-minute delay:

```
service.scheduleAtFixedRate(command, 5, 1, TimeUnit.MILLISECONDS);
```

One risk of using this method is the possibility a task could consistently take longer to run than the period between tasks.

What would happen if the task consistently took five minutes to execute? In spite of the fact that the task is still running, the `ScheduledExecutorService` would submit a new task to be started every minute.

If a single-thread executor was used, over time this would result in endless set tasks being scheduled, which would then run back to back assuming that no other tasks were submitted to the `ScheduledExecutorService`.

On the other hand, the `scheduleAtFixedDelay()` method creates a new task only after the previous task has finished. For example, if the first task runs at 12:00 and takes five minutes to finish, with a period of 2 minutes, then the second task will start at 12:07.

```
service.scheduleAtFixedDelay(command, 0, 2, TimeUnit.MILLISECONDS);
```

Thread Pools

All of the examples up until now have been with single-thread executors, which do not provide any concurrency in themselves.

There are three additional factory methods in the `Executors` class that act on a pool of threads, rather than on a single thread.

A thread pool is a group of pre-instantiated reusable threads that are available to perform a set of arbitrary tasks. This table shows the new `Executors` factory methods plus the ones you have already seen:

Method	Return Type	Description
<code>newSingleThreadExecutor()</code>	<code>ExecutorService</code>	Creates a single-threaded executor that uses a single worker thread operating off an unbounded queue. Results are processed sequentially in the order in which they are submitted
<code>newSingleThreadScheduledExecutor()</code>	<code>ScheduledExecutorService</code>	Creates a single-threaded executor that can schedule

		commands to run after a given delay or to execute periodically
newCachedThreadPool()	ExecutorService	Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available
newFixedThreadPool(int nThreads)	ExecutorService	Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue
newScheduledThreadPool(int nThreads)	ScheduledExecutorService	Creates a thread pool that can schedule commands to run after a given delay or to execute periodically

These methods return the same instance types, ExecutorService and ScheduledExecutorService, that you saw earlier in this chapter.

Therefore, all of the previous examples will work with these new pooled-thread executors.

The difference between a single-thread and a pooled-thread executor is what happens when a task is already running.

While a single-thread executor will wait for an available thread to become available before running the next task, a pooled-thread executor can execute the next task concurrently. If the pool runs out of available threads, the task will be queued by the thread executor and wait to be completed.

The newCachedThreadPool() method will create a thread pool of unbounded size, allocating a new thread anytime one is required or all existing threads are busy. This is commonly used for pools that require executing many short-lived asynchronous tasks.

For long-lived processes, usage of this executor is strongly discouraged, as it could grow to encompass a large number of threads over the application life cycle.

The newFixedThreadPool() takes a number of threads and allocates them all upon creation. As long as our number of tasks is less than our number of threads, all tasks will be executed concurrently.

If at any point the number of tasks exceeds the number of threads in the pool, they will wait in similar manner as you saw with a single-thread executor.

Calling newFixedThreadPool() with a value of 1 is equivalent to calling newSingleThreadExecutor().

The newScheduledThreadPool() method is identical to newFixedThreadPool(), except that it returns an instance of ScheduledExecutorService and is therefore compatible with scheduling tasks.

This executor has subtle differences in the way that the scheduleAtFixedRate() performs.

Going back to the previous example in which tasks took five minutes to complete:

```
ScheduledExecutorService service =  
    Executors.newScheduledThreadPool(10);  
service.scheduleAtFixedRate(command, 3, 1, TimeUnit.MILLISECONDS);
```

Whereas with a single-thread executor and a five-minute task execution time, an endless set of tasks would be scheduled over time, with a pooled executor, this can be avoided if the pool size is sufficiently large. With a pool of 10, for example, then as each thread finishes, it is returned to the pool and results in new threads available for the next tasks as they come up.

Synchronising Access to Data

Thread safety is the property of an object that guarantees safe execution by multiple threads at the same time.

Now that we have multiple threads capable of accessing the same objects in memory, we need to make sure to organise our access to this data such that we don't end up with invalid or unexpected results. Since threads run in a shared environment and memory space, how do we prevent two threads from interfering with each other?

Let's say the call centre has a program to count incoming and outgoing calls.

Each call is logged and the total is updated and printed out.

```
import java.util.concurrent.*;
public class CallManager {
    private int callCount = 0;
    private void incrementAndReport() {
        System.out.print((++callCount) + " ");
    }
    public static void main(String[] args) {
        ExecutorService service = null;
        try {
            service = Executors.newFixedThreadPool(20);
            CallManager manager = new CallManager();
            for(int i=0; i<10; i++) {
                service.submit(() -> manager.incrementAndReport());
            }
        } finally {
            if(service != null) service.shutdown();
        }
    }
}
```

Notice that the pre-increment `++` operator is used to update the `callCount` variable.

Remember that the pre-increment operator is shorthand for the following expression in which the newly assigned value is returned:

```
callCount = callCount + 1;
```

A problem occurs when two threads both execute the right side of the expression, reading the “old” value before either thread writes the “new” value of the variable. The two assignments become redundant; they both assign the same new value, with one thread overwriting the results of the other.

The following diagram demonstrates this problem with two threads, assuming that `callCount` has a starting value of 1.

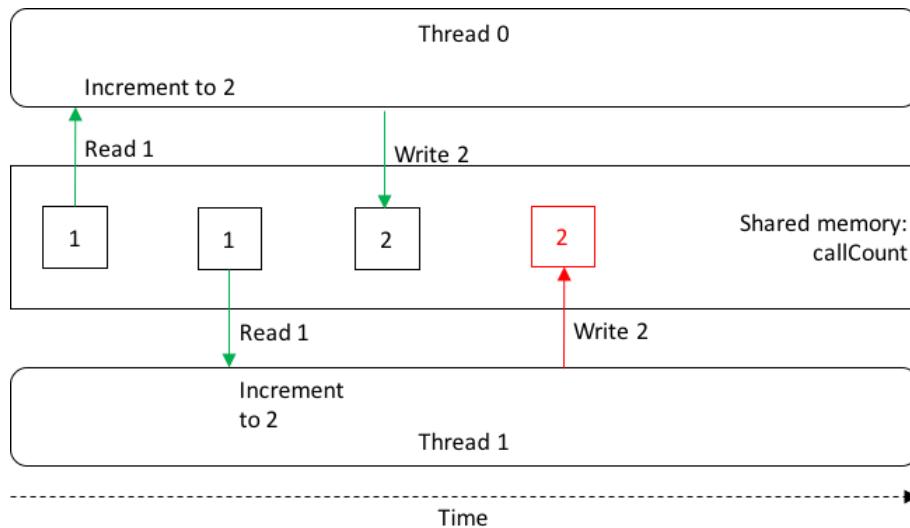


Figure 7-2 Thread Synchronisation

You can see that both threads read and write the same values, causing one of the two `++callCount` operations to be lost.

Therefore, the increment operator `++` is not thread-safe. As you will see later in this chapter, the unexpected result of two tasks executing at the same time is referred to as a race condition.

Returning to our CallManager application, we choose a large thread size of 20 so that all tasks can be run concurrently. Let's say that each lambda expression submitted to the thread executor corresponds to a call centre worker. Each time a call event increments the `callCounter` counter the program reports the results. What would you expect the output of this program to be? Although the output will vary, the following are some samples created by this program:

```
1 2 2 3 4 5 6 7 8 9
2 1 3 4 5 6 7 8 9 10
2 4 5 6 7 8 1 9 10 3
```

In this example, multiple threads are sharing the `callCount` variable. In the first sample, two threads both call `++callCount` at the same time, resulting in one of the increment operations actually being lost, with the last total being 9 instead of 10.

In the other examples, results from earlier threads are output before ones that started later, such as 3 being output after 4 in the third example. We know that we had 10 calls, but the results are incomplete and out of order.

Atomic Classes

With the release of the Concurrency API, Java added a new `java.util.concurrent.atomic` package to help coordinate access to primitive values and object references.

As with most classes in the Concurrency API, these classes are added solely for convenience.

In our first CallManager sample output, the same value, 2, was printed twice, with the highest counter being 9 instead of 10. As we demonstrated in the previous section, the increment operator `++` is not thread-safe. Furthermore, the reason that it is not thread-safe is that the operation is not atomic, carrying out two tasks, read and write, that can be interrupted by other threads.

Atomic is the property of an operation to be carried out as a single unit of execution without any interference by another thread.

A thread-safe atomic version of the increment operator would be one that performed the read and write of the variable as a single operation, not allowing any other threads to access the variable during the operation.

This diagram shows the result of using an atomic variable for `callCount`.

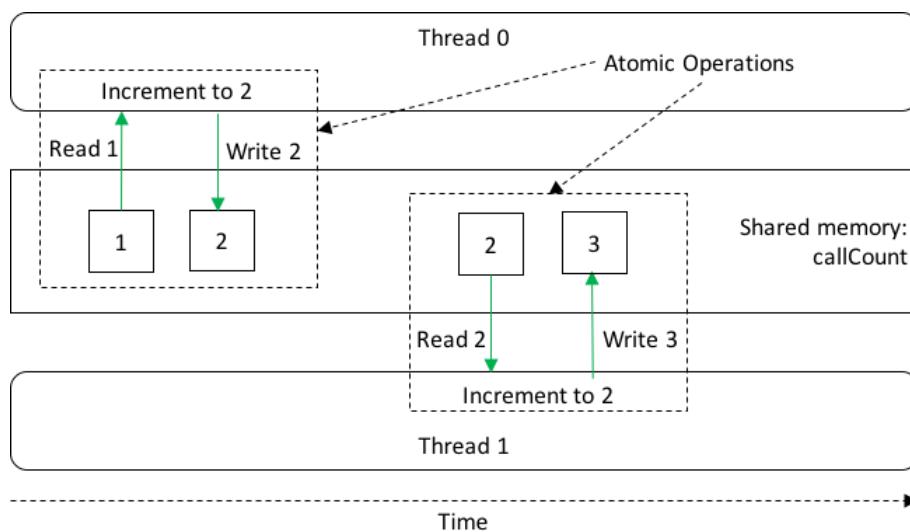


Figure 7-3 Atomic Operations

This resembles the previous diagram, except that reading and writing the data is atomic with regard to the `callCount` variable. Any thread trying to access the `callCount` variable while an atomic operation is in process will have to wait until the atomic operation on the variable is complete.

This exclusivity applies only to the `callCount` variable, with the rest of the memory space unaffected by this operation.

Since accessing primitives and references in Java is common in shared environments, the Concurrency API includes numerous useful classes that are conceptually the same as our primitive classes and arrays but that support atomic operations. These so called `AtomicX` classes are:

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicIntegerArray`
- `AtomicLong`

- AtomicLongArray
- AtomicReference<V>
- AtomicReferenceArray<E>

Each class includes numerous methods that are equivalent to many of the primitive built-in operators that we use on primitives, such as the assignment operator = and the increment operators ++.

The common atomic methods are shown in this table:

Method	Description
get()	Retrieve the current value
set()	Set the given value, equivalent to the assignment = operator
getAndSet()	Atomically sets the new value and returns the old value
incrementAndGet()	For numeric classes, atomic pre-increment operation equivalent to ++value
getAndIncrement()	For numeric classes, atomic post-increment operation equivalent to value++
decrementAndGet()	For numeric classes, atomic pre-decrement operation equivalent to --value
getAndDecrement()	For numeric classes, atomic post-decrement operation equivalent to value--

In the following example, we update our CallManager class with an AtomicInteger:

```
private AtomicInteger callCount = new AtomicInteger(0);
private void incrementAndReport() {
    System.out.print(callCount.incrementAndGet() + " ");
}
```

How does this implementation differ from our previous examples? When we run this modification, we get varying output, such as the following:

```
2 3 1 4 5 6 7 8 9 10
1 4 3 2 5 6 7 8 9 10
1 4 3 5 6 2 7 8 10 9
```

Unlike our previous sample output, the numbers 1 through 10 will always be output.

The results are still not ordered because one thread may be performing an update in parallel with another thread which is reporting the update it has just completed. The key here is that using the atomic classes ensures that the data is consistent between threads and that no values are lost due to concurrent modifications.

Synchronized Blocks

How do we improve the results so that each worker is able to increment and report the results in order? The most common technique is to use a monitor, also called a lock, to synchronise access. A monitor is a structure that supports mutual exclusion or the property that at most one thread is executing a particular segment of code at a given time.

In Java, any Object can be used as a monitor, along with the synchronized keyword, as shown in the following example:

```
CallManager manager = new CallManager();
synchronized(manager) {
    // Work to be completed by one thread at a time
}
```

This example is referred to as a synchronized block.

Each thread that arrives will first check if any threads are in the block. In this manner, a thread “acquires the lock” for the monitor. If the lock is available, a single thread will enter the block, acquiring the lock and preventing all other threads from entering.

While the first thread is executing the block, all threads that arrive will attempt to acquire the same lock and wait for first thread to finish. While waiting their thread state is Thread.State.BLOCKED.

Once a thread finishes executing the block, it will release the lock, allowing one of the waiting threads to proceed.

Looking again at the CallManager example perhaps we can improve the results so that each worker increments and outputs the counter in order.

Let's replace the for() loop with the following implementation:

```
for(int i=0; i<10; i++) {
    synchronized(manager) {
        service.submit(() -> manager.incrementAndReport());
    }
}
```

Does this solution fix the problem? Unfortunately, no it doesn't.

We've synchronized the creation of the threads but not the execution of the threads. In this example, each thread would be created one at a time, but they may all still execute and perform their work at the same time, resulting in the same type of output that you saw earlier.

Diagnosing and resolving threading problems is often one of the most difficult tasks in any programming language.

We now present a corrected version of the CallManager class, which does order the worker threads:

```
import java.util.concurrent.*;
public class SheepManager {
    private int sheepCount = 0;
    private void incrementAndReport() {
```

```

        synchronized(this) {
            System.out.print((++sheepCount)+" ");
        }
    }
public static void main(String[] args) {
    ExecutorService service = null;
    try {
        service = Executors.newFixedThreadPool(20);
        SheepManager manager = new SheepManager();
        for(int i=0; i<10; i++){
            service.submit(() -> manager.incrementAndReport());
        }
    } finally {
        if(service != null){
            service.shutdown();
        }
    }
}
}

```

When this code executes, it will consistently output the following:

1 2 3 4 5 6 7 8 9 10

Although all threads are still created and executed at the same time, they each wait at the synchronized block for the worker to increment and report the result before entering. In this manner, each incoming or outgoing call event waits for the previous one to complete updating and reporting before proceeding.

While it's still random which thread will run the code next, it is guaranteed that there will be at most one working at any one time.

We could have synchronized on any object, so long as it was the same object. For example, the following code snippet would have also worked:

```

private final Object lock = new Object();
private void incrementAndReport() {
    synchronized(lock) {
        System.out.print((++sheepCount)+" ");
    }
}

```

Although there's no need to make the lock variable final, doing so ensures that it is not reassigned after threads start using it.

Synchronized Methods

In the previous example, we established our monitor using synchronized(this) around the body of the method.

Java actually provides a more convenient compiler enhancement for doing so. We can add the synchronized modifier to any instance method to synchronise automatically on the object itself.

For example, the following two method definitions are equivalent:

```
private void incrementAndReport() {  
    synchronized(this) {  
        System.out.print((++sheepCount) + " ");  
    }  
}  
  
private synchronized void incrementAndReport() {  
    System.out.print((++sheepCount) + " ");  
}
```

We can also add the synchronized modifier to static methods.

Guess what object is used as the monitor when we synchronise on a static method? The class object, of course! It couldn't be an instance as there may not be one.

The Cost of Synchronization

We complete this section by noting that synchronisation, while useful, may be costly in practice. While multi-threaded programming is about doing multiple things at the same time, synchronisation is about taking multiple threads and making them perform in a more single-threaded manner.

For example, let's say that we have a highly concurrent class with numerous methods that synchronise on the same object. Let's say that 50 concurrent threads access it. Let's also say that, on average, each thread takes a modest 100 milliseconds to execute.

In this example, if all of the threads try to access the monitor at the same time, how long will it take for them to complete their work, assuming that 50 threads are available in the thread pool?

$$\begin{aligned} 50 \text{ threads} \times 100 \text{ milliseconds} &= 5,000 \text{ milliseconds} \\ &= 5 \text{ seconds} \end{aligned}$$

Even though five seconds may not seem like a lot, it's actually pretty long in computer time. What if 50 new tasks are created before the five seconds are up? This will pile onto the workload, resulting in most threads constantly entering a waiting or blocked state. In the application, this may cause tasks that would normally be quick to finish in a non-synchronised environment to take a significantly long amount of time to complete.

Synchronisation is about protecting data integrity at the cost of performance. In many cases, performance costs are minimal, but in extreme scenarios the application could slow down significantly due to the inclusion of synchronisation.

Being able to identify synchronization problems, including finding ways to improve performance in synchronised multi-threaded environments, is a valuable skill to attain.

Concurrent Collections

Besides managing threads, the Concurrency API includes interfaces and classes that help you coordinate access to collections across multiple tasks.

Concurrent collections are in the Java Collections Framework that you learned about earlier.

In this section, you will see many of the new concurrent classes available to you when using the Concurrency API.

Rationale for using Concurrent Collections

So why do we need the new concurrent collection classes? There are other options; in the previous section you saw that we can use the `synchronized` keyword on any method or block, so this technique could just be used with the existing collection classes?

It is true that you could code synchronized blocks to ensure that your code uses an ‘ordinary’ collection in a thread-safe way. However, you would have to make sure that no one else (another class) gets access to your collection otherwise that other code would have to have synchronized blocks in all the necessary places too.

Just like using `ExecutorService` to manage threads for us, using the concurrent collections is extremely convenient.

It also prevents us from introducing mistakes in our own custom implementation, such as if we forgot to synchronize one of the accessor methods and it doesn’t matter if someone else comes along later and writes code to use the collection.

In fact, the concurrent collections often include performance enhancements that avoid unnecessary synchronisation. Accessing collections from across multiple threads is so common that the writers of Java thought it would be a good idea to have alternate versions of many of the regular collections classes just for multi-threaded access.

Memory Consistency Errors

The purpose of the concurrent collection classes is to solve common memory consistency errors. A memory consistency error occurs when two threads have inconsistent views of what should be the same data. Conceptually, we want writes on one thread to be available to another thread if it accesses the concurrent collection after the write has occurred.

When two threads try to modify the same non-concurrent collection, the JVM may throw a `ConcurrentModificationException` at runtime. In fact, it can happen with a single thread.

Look at the following code:

```
package demo;
import java.util.Map;
import java.util.HashMap;
import java.time.LocalDateTime;
public class Concur{
    public static void main(String[] args){
        Map<LocalDateTime, String> cal IData =
            new HashMap<LocalDateTime, String>();
        cal IData.put(LocalDateTime.now(), "Incoming");
        cal IData.put(LocalDateTime.now(), "Outgoing");
        for(LocalDateTime key: cal IData.keySet()){
            cal IData.remove(key);
        }
    }
}
```

This code throws a `ConcurrentModificationException` at runtime, since the iterator `keySet()` is not properly updated after the first element is removed.

Changing the first line to use a `ConcurrentHashMap` will prevent the code from throwing an exception at runtime.

Although we don't usually modify a loop variable, this example highlights the fact that the `ConcurrentHashMap` is ordering read and write access such that all access to the class is consistent.

In this code snippet, the iterator created by `keySet()` is updated as soon as an object is removed from the Map.

The concurrent classes were created to help avoid common issues in which multiple threads are adding and removing objects from the same collections.

At any given instance, all threads should have the same consistent view of the structure of the collection.

Concurrent Classes

There are numerous collection classes with which you should be familiar for multi-threaded coding. Luckily, you already know how to use most of them, as the methods available are a superset to the non-concurrent collection classes that you learned about earlier in the course.

You should use a concurrent collection class anytime that you are going to have multiple threads modify a collections object outside a synchronized block or method, even if you don't expect a concurrency problem. On the other hand, if all of the threads are accessing an established immutable or read-only collection, a concurrent collection class is not required.

In the same way that we instantiate an `ArrayList` object but pass around a `List` reference, it is considered a good practice to instantiate a concurrent collection but pass it around using a non-concurrent interface whenever possible. This has some similarities with the

factory pattern that you learned about earlier, as the users of these objects may not be aware of the underlying implementation.

In some cases, the callers may need to know that it is a concurrent collection so that a concurrent interface or class is appropriate, but for the majority of circumstances, that distinction is not necessary.

Here is a list of the most commonly used concurrent classes:

- ConcurrentHashMap
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- LinkedBlockingDeque
- LinkedBlockingQueue

Based on your existing knowledge of collections, classes like ConcurrentHashMap, ConcurrentLinkedQueue, and ConcurrentLinkedDeque should be quite easy to learn.

Take a look at the following code samples:

```
Map<String, Integer> map = new ConcurrentHashMap<>();
map.put("zebra", 52);
map.put("el elephant", 10);
System.out.println(map.get("el elephant"));

Queue<Integer> queue = new ConcurrentLinkedQueue<>();
queue.offer(31);
System.out.println(queue.peek());
System.out.println(queue.poll());

Deque<Integer> deque = new ConcurrentLinkedDeque<>();
deque.offer(10);
deque.push(4);
System.out.println(deque.peek());
System.out.println(deque.pop());
```

These samples strongly resemble the collections samples that you saw earlier in the course, with the only difference being the object creation call.

In each of the samples, we assign an interface reference to the newly created object and use it the same way as we would a non-concurrent object.

The ConcurrentHashMap implements the ConcurrentMap interface, also found in the Concurrency API.

You can use either reference type, Map or ConcurrentMap, to access a ConcurrentHashMap object, depending on whether or not you want the caller to know anything about the underlying implementation.

For example, a method signature may require a `ConcurrentMap` reference to ensure that object passed to it is properly supported in a multi-threaded environment.

Blocking Queues

In the list above there are two queue classes that implement blocking interfaces:

- `LinkedBlockingQueue`
- `LinkedBlockingDeque`.

The `BlockingQueue` is just like a regular `Queue`, except that it includes methods that will wait a specific amount of time to complete an operation.

Since `BlockingQueue` inherits all of the methods from `Queue`, let's skip the inherited methods and present the new waiting methods:

- `offer(E e, long timeout, TimeUnit unit)`
Adds item to the queue waiting the specified time, returning false if time elapses before space is available
- `poll(long timeout, TimeUnit unit)`
Retrieves and removes an item from the queue, waiting the specified time, returning null if the time elapses before the item is available

A `LinkedBlockingQueue`, as the name implies, maintains a linked list between elements.

The following sample is using a `LinkedBlockingQueue` to wait for the results of some of the operations.

The `offer` and `poll` methods can each throw a checked `InterruptedException`, as they can be interrupted before they finish waiting for a result; therefore they must be properly caught.

```
try {  
    BlockingQueue<Integer> blockingQueue = new LinkedBlockingQueue<>();  
    blockingQueue.offer(39);  
    blockingQueue.offer(3, 4, TimeUnit.SECONDS);  
    System.out.println(blockingQueue.poll());  
    System.out.println(blockingQueue.poll(10, TimeUnit.MILLISECONDS));  
} catch (InterruptedException e) {  
    // Handle interruption  
}
```

As shown in this example, since `LinkedBlockingQueue` implements both `Queue` and `BlockingQueue`, we can use methods available to both, such as those that don't take any wait arguments.

The `LinkedBlockingDeque` class that maintains a doubly linked list between elements and implements a `BlockingDeque` interface.

The `BlockingDeque` interface extends `Deque` much in the same way that `BlockingQueue` extends `Queue`, providing numerous waiting methods.

Here are the waiting methods defined in BlockingDeque.

- `offerFirst(E e, long timeout, TimeUnit unit)`
Adds an item to the front of the queue, waiting a specified time, returning false if time elapses before space is available
- `offerLast(E e, long timeout, TimeUnit unit)`
Adds an item to the tail of the queue, waiting a specified time, returning false if time elapses before space is available
- `pollFirst(long timeout, TimeUnit unit)`
Retrieves and removes an item from the front of the queue, waiting the specified time, returning null if the time elapses before the item is available
- `pollLast(long timeout, TimeUnit unit)`
Retrieves and removes an item from the tail of the queue, waiting the specified time, returning null if the time elapses before the item is available

Here is a sample of using a LinkedBlockingDeque. As before, the methods each throw a checked InterruptedException, they must be properly caught in the code that uses them.

```
try {
    BlockingDeque<Integer> blockingDeque = new LinkedBlockingDeque<>();
    blockingDeque.offer(91);
    blockingDeque.offerFirst(5, 2, TimeUnit.MILLISECONDS);
    blockingDeque.offerLast(47, 100, TimeUnit.MILLISECONDS);
    blockingDeque.offer(3, 4, TimeUnit.SECONDS);
    System.out.println(blockingDeque.poll());
    System.out.println(blockingDeque.poll(950, TimeUnit.MILLISECONDS));
    System.out.println(blockingDeque.pollFirst(200,
                                              TimeUnit.NANOSECONDS));
    System.out.println(blockingDeque.pollLast(1, TimeUnit.SECONDS));
} catch (InterruptedException e) {
    // Handle interruption
}
```

This example creates a LinkedBlockingDeque and assigns it to a BlockingDeque reference. Since BlockingDeque extends Queue, Deque, and BlockingQueue, all of the previously defined queue methods are available for use.

SkipList Collections

The SkipList classes, ConcurrentSkipListSet and ConcurrentSkipListMap, are concurrent versions of their sorted counterparts, TreeSet and TreeMap, respectively.

They maintain their elements or keys in the natural ordering of their elements. When you see SkipList or SkipSet, just think “sorted” concurrent collections and the rest should follow naturally.

Like other queue examples, it is recommended that you assign these objects to interface references, such as SortedMap or NavigableSet. In this manner, using them is the same as all the code that you worked with in the chapter on collections.

CopyOnWrite Collections

There are two concurrent classes, CopyOnWriteArrayList and CopyOnWriteArraySet, that behave a little differently than the other concurrent examples that you have seen.

These classes copy all of their elements to a new underlying structure anytime an element is added, modified, or removed from the collection. By a modified element, we mean that the reference in the collection is changed. Modifying the actual contents of the collection will not cause a new structure to be allocated.

Although the data is copied to a new underlying structure, our reference to the object does not change. This is particularly useful in multi-threaded environments that need to iterate the collection.

Any iterator established prior to a modification will not see the changes, but instead it will iterate over the original elements prior to the modification.

Let's look at an example:

```
List<Integer> list = new CopyOnWriteArrayList(Arrays.asList(4, 3, 52));
for(Integer item: list) {
    System.out.print(item+" ");
    list.add(9);
}
System.out.println();
System.out.println("Size: "+list.size());
```

When executed as part of a program, this code snippet outputs the following:

```
4 3 52 Size: 6
```

Despite adding elements to the array while iterating over it, only those elements in the collection at the time the for() loop was created were accessed.

Alternatively, if we had used a regular ArrayList object, a ConcurrentModificationException would have been thrown at runtime.

With either class, though, we avoid entering an infinite loop in which elements are constantly added to the array as we iterate over them.

The CopyOnWrite classes can use a lot of memory, since a new collection structure needs be allocated anytime the collection is modified. They are commonly used in multi-threaded environment situations where reads are far more common than writes.

Converting to Synchronised Collections

Besides the concurrent collection classes that we have covered, the Concurrency API also includes methods for obtaining synchronised versions of existing non-concurrent collection objects.

These methods, defined in the Collections class, contain synchronised methods that operate on the inputted collection and return a reference that is the same type as the underlying collection.

These methods are shown here:

- synchronizedCollection(Collection<T> c)
- synchronizedList(List<T> list)
- synchronizedMap(Map<K,V> m)
- synchronizedNavigableMap(NavigableMap<K,V> m)
- synchronizedNavigableSet(NavigableSet<T> s)
- synchronizedSet(Set<T> s)
- synchronizedSortedMap(SortedMap<K,V> m)
- synchronizedSortedSet(SortedSet<T> s)

If you know at the time of creation that your object requires synchronization, then you should use the appropriate concurrent collection class.

On the other hand, if you are given an existing collection that is not a concurrent class and need to access it among multiple threads, you can wrap it using the methods above.

These methods in synchronise access to the data elements, such as the get() and set() methods, they do not synchronise access on any iterators that you may create from the synchronised collection.

Therefore, it is imperative that you use a synchronisation block if you need to iterate over any of the returned collections and be aware that they may throw a ConcurrentModificationException at runtime.

Parallel Streams

Earlier in the course, you learned that the Streams API enables functional programming in Java 8.

One of the most powerful features of the Streams API is that it has built-in concurrency support. The streams with which you have worked on this course so far have been serial streams. A serial stream is a stream in which the results are ordered, with only one entry being processed at a time.

A parallel stream is a stream that is capable of processing results concurrently, using multiple threads.

For example, you can use a parallel stream and the stream `map()` method to operate concurrently on the elements in the stream, vastly improving performance over processing a single element at a time.

Using a parallel stream can change not only the performance of your application but also the expected results. As you shall see, some operations also require special handling to be able to be processed in a parallel manner.

NOTE

By default, the number of threads available in a parallel stream is based on the number of available CPUs in your environment.

To increase the thread count, you need to create your own custom class.

Creating Parallel Streams

The Streams API was designed to make creating parallel streams quite easy. You should become familiar with the two main ways of creating a parallel stream.

parallel()

The first way to create a parallel stream is from an existing stream. You just call `parallel()` on an existing stream to convert it to one that supports multi-threaded processing, as shown in the following code:

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
Stream<Integer> parallelStream = stream.parallel();
```

Be aware that `parallel()` is an intermediate operation that operates on the original stream.

parallelStream()

The second way to create a parallel stream is from a Java collection class. The Collection interface includes a method `parallelStream()` that can be called on any collection and

returns a parallel stream. The following is a revised code snippet that creates the parallel stream directly from the List object:

```
Stream<Integer> parallelStream2 =
    Arrays.asList(1, 2, 3, 4, 5, 6).parallelStream();
```

We will use parallelStream() on Collection objects throughout this section.

Parallel Processing

Creating the parallel stream is the easy part. The interesting part comes in using it. Here's a serial example:

```
Arrays.asList(1, 2, 3, 4, 5, 6)
    .stream()
    .forEach(s -> System.out.print(s + " "));
```

What do you think this code will output when executed as part of a main() method?

This is the output:

```
12345 6
```

As you might expect, the results are ordered and predictable because we are using a serial stream. What happens if we use a parallel stream, though?

```
Arrays.asList(1, 2, 3, 4, 5, 6)
    .parallelStream()
    .forEach(s -> System.out.print(s + " "));
```

With a parallel stream, the forEach() operation is applied across multiple elements of the stream concurrently. The following are each sample outputs of this code snippet:

```
4 1 6 5 2 3
5 2 1 3 6 4
1 2 6 4 5 3
```

As you can see, the results are no longer ordered or predictable. If you compare this to earlier parts of the chapter, the forEach() operation on a parallel stream is equivalent to submitting multiple Runnable lambda expressions to a pooled thread executor.

The Streams API includes an alternate version of the forEach() operation called forEachOrdered(), which forces a parallel stream to process the results in order at the cost of performance. For example:

```
Arrays.asList(1, 2, 3, 4, 5, 6)
    .parallelStream()
    .forEachOrdered(s -> System.out.print(s + " "));
```

Output:

```
1 2 3 4 5 6
```

Performance Improvements

Let's look at another example to see how much using a parallel stream may improve performance in your applications.

Let's say that you have a task that requires processing 4,000 records, with each record taking a modest 10 milliseconds to complete. The following is a sample implementation that uses `Thread.sleep()` to simulate processing the data:

```
import java.util.*;  
public class DataCalculator {  
    public int processRecord(int input) {  
        try {  
            Thread.sleep(10);  
        } catch (InterruptedException e) {  
            // Handle interrupted exception  
        }  
        return input+1;  
    }  
    public void processAllData(List<Integer> data) {  
        data.stream().map(a -> processRecord(a)).count();  
    }  
    public static void main(String[] args) {  
        DataCalculator calculator = new DataCalculator();  
        // Define the data  
        List<Integer> data = new ArrayList<Integer>();  
        for(int i=0; i<4000; i++) {  
            data.add(i); // Process the data  
        }  
        long start = System.currentTimeMillis();  
        calculator.processAllData(data);  
        double time = (System.currentTimeMillis()-start)/1000.0;  
        // Report results  
        System.out.println("\nTasks completed in: "+time+" seconds");  
    }  
}
```

Given that there are 4,000 records, and each record takes 10 milliseconds to process, by using a serial `stream()`, the results will take approximately 40 seconds to complete this task. Each task is completed one at a time:

```
Tasks completed in: 40.044 seconds
```

If we use a parallel stream, though, the results can be processed concurrently:

```
public void processAllData(List<Integer> data) {  
    data.parallelStream().map(a -> processRecord(a)).count();  
}
```

Depending on the number of CPUs available in your environment, the following is a possible output of the code using a parallel stream:

```
Tasks completed in: 10.542 seconds
```

You see that using a parallel stream can have a four-fold improvement in the results. Even better, the results scale with the number of processors. Scaling is the property that, as we add more resources such as CPUs, the results gradually improve.

Does that mean that all of your streams should be parallel? Not quite. Parallel streams tend to achieve the most improvement when the number of elements in the stream is significantly large. For small streams, the improvement is often limited, as there are some overhead costs to allocating and setting up the parallel processing.

Independent Operations

Parallel streams can improve performance because they rely on the property that many stream operations can be executed independently.

An independent operation is one where the results of an operation on one element of a stream do not require or impact the results of another element of the stream.

For example, in the previous example, each call to `processRecord()` can be executed separately, without impacting any other invocation of the method.

As another example, consider the following lambda expression supplied to the `map()` method, which maps the stream contents to uppercase strings:

```
Arrays.asList("desktop", "server", "monitor")
    .parallelStream()
    .map(s -> s.toUpperCase())
    .forEach(System.out::println);
```

In this example, mapping desktop to DESKTOP can be done independently of mapping monitor to MONITOR. In other words, multiple elements of the stream can be processed at the same time and the results will not change.

Many common streams including `map()`, `forEach()`, and `filter()` can be processed independently, although order is never guaranteed. Consider the following modified version of our previous stream code:

```
Arrays.asList("desktop", "server", "monitor")
    .parallelStream()
    .map(s -> {System.out.println(s); return s.toUpperCase(); })
    .forEach(System.out::println);
```

This example includes an embedded `print` statement in the lambda passed to the `map()` method. While the return values of the `map()` operation are the same, the order in which they are processed can result in very different output.

We might even print terminal results before the intermediate operations have finished, as shown in the following generated output:

```
desktop DESKTOP server monitor SERVER MONITOR
```

When using streams, you should avoid any lambda expressions that can produce side effects.

Stateful Operations

Side effects can also appear in parallel streams if your lambda expressions are stateful. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline.

On the other hand, a stateless lambda expression is one whose result does not depend on any state that might change during the execution of a pipeline.

Let's take a look at an example to see why stateful lambda expressions should be avoided in parallel streams:

```
List<Integer> data = Collections.synchronizedList(new ArrayList<>());
Arrays.asList(1, 2, 3, 4, 5, 6).parallelStream()
    .map(i -> {data.add(i); return i; })
    // AVOID STATEFUL LAMBDA EXPRESSIONS!
    .forEachOrdered(i -> System.out.print(i + " "));
System.out.println();
for(Integer e: data) {
    System.out.print(e + " ");
}
```

The following is a sample generation of this code snippet using a parallel stream:

```
1 2 3 4 5 6
2 4 3 5 6 1
```

The `forEachOrdered()` method displays the numbers in the stream sequentially, whereas the order of the elements in the data list is completely random.

You can see that a stateful lambda expression, which modifies the data list in parallel, produces unpredictable results at runtime.

Note that this would not have been noticeable with a serial stream, where the results would have been the following:

```
1 2 3 4 5 6
1 2 3 4 5 6
```

It is strongly recommended that you avoid stateful operations when using parallel streams, so as to remove any potential data side effects. In fact, they should generally be avoided in serial streams wherever possible, since they prevent your streams from taking advantage of parallelisation.

Processing Parallel Reductions

Besides possibly improving performance and modifying the order of operations, using parallel streams can impact how you write your application.

Reduction operations on parallel streams are referred to as parallel reductions. The results for parallel reductions can be different from what you expect when working with serial streams.

Since order is not guaranteed with parallel streams, methods such as `findAny()` on parallel streams may result in unexpected behaviour. Let's take a look at the results of `findAny()` applied to a serial stream:

```
System.out.print(Arrays.asList(1, 2, 3, 4, 5, 6).stream().findAny().get());
```

This code consistently outputs the first value in the serial stream, 1.

With a parallel stream, the JVM can create any number of threads to process the stream.

When you call `findAny()` on a parallel stream, the JVM selects the first thread to finish the task and retrieves its data:

```
System.out.print(Arrays.asList(1, 2, 3, 4, 5, 6).parallelStream()
    .findAny()
    .get());
```

The result is that the output could be 4, 1, or really any value in the stream. You can see that with parallel streams, the results of `findAny()` are no longer predictable.

Any stream operation that is based on order, including `findFirst()`, `limit()`, or `skip()`, may actually perform more slowly in a parallel environment. This is a result of a parallel processing task being forced to coordinate all of its threads in a synchronised-like fashion.

On the plus side, the results of ordered operations on a parallel stream will be consistent with a serial stream. For example, calling `skip(5).limit(2).findFirst()` will return the same result on ordered serial and parallel streams.

Combining Results with reduce()

As you learned earlier, the stream operation `reduce()` combines a stream into a single object. Recall that first parameter to the `reduce()` method is called the identity, the second parameter is called the accumulator, and the third parameter is called the combiner.

We can concatenate a string using the `reduce()` method to produce `wolf`, as shown in the following example:

```
System.out.println(Arrays.asList('b', 'o', 'o', 'k').stream()
    .reduce("", (c, s1) -> c + s1, (s2, s3) -> s2 + s3));
```

On parallel streams, the `reduce()` method works by applying the reduction to pairs of elements within the stream to create intermediate values and then combining those intermediate values to produce a final result. Whereas with a serial stream, `wolf` was built one character at a time, in a parallel stream, the intermediate strings `wo` and `lf` could have been created and then combined.

With parallel streams, though, we now have to be concerned about order. What if the elements of a string are combined in the wrong order to produce `okob` or `koob`? The Streams API prevents this problem, while still allowing streams to be processed in parallel, as long as the arguments to the `reduce()` operation adhere to certain principles.

reduce() Method Arguments

The identity must be defined such that for all elements in the stream `u`, `combiner.apply(identity, u)` is equal to `u`.

The accumulator operator `op` must be associative and stateless such that `(a op b) op c` is equal to `a op (b op c)`.

The combiner operator must also be associative and stateless and compatible with the identity, such that for all `u` and `t` `combiner.apply(u, accumulator.apply(identity, t))` is equal to `accumulator.apply(u, t)`.

If you follow these principles when building your `reduce()` arguments, then the operations can be performed using a parallel stream and the results will be ordered as they would be with a serial stream. Note that these principles still apply to the identity and accumulator when using the one- or two-argument version of `reduce()` on parallel streams.

Although the one- and two-argument versions of `reduce()` do support parallel processing, it is recommended that you use the three-argument version of `reduce()` when working with parallel streams. Providing an explicit combiner method allows the JVM to partition the operations in the stream more efficiently.

Combing Results

Like `reduce()`, the Streams API includes a three-argument version of `collect()` that takes accumulator and combiner operators, along with a supplier operator instead of an identity.

Also like `reduce()`, the accumulator and combiner operations must be associative and stateless, with the combiner operation compatible with the accumulator operator, as previously discussed. In this manner, the three-argument version of `collect()` can be performed as a parallel reduction, as shown in the following example:

```
Stream<String> stream = Stream.of("b", "o", "o", "k").parallel();
SortedSet<String> set = stream.collect(ConcurrentSkipListSet::new,
Set::add, Set::addAll);
System.out.println(set); // [b, k, o, o]
```

Recall that elements in a `ConcurrentSkipListSet` are sorted according to their natural ordering.

You should use a concurrent collection to combine the results, ensuring that the results of concurrent threads do not cause a `ConcurrentModificationException`.

Managing Concurrent Processes

The Concurrency API includes classes that can be used to coordinate tasks among a group of related threads. These classes are designed for use in specific scenarios, similar to many of the design patterns that you saw earlier. In this section, we present two classes with which you should be familiar for the exam, CyclicBarrier and ForkJoinPool.

CyclicBarrier

A CyclicBarrier object is a synchronisation aid that allows a set of threads to all wait for each other to reach a common barrier point.

CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

Here is some sample code:

```
class Solver {
    final int N;
    final float[][] data;
    final CyclicBarrier barrier;

    class Worker implements Runnable {
        int myRow;
        Worker(int row) { myRow = row; }
        public void run() {
            while (!done()) {
                processRow(myRow);

                try {
                    barrier.await();
                } catch (InterruptedException ex) {
                    return;
                } catch (BrokenBarrierException ex) {
                    return;
                }
            }
        }
    }

    public Solver(float[][] matrix) {
        data = matrix;
        N = matrix.length;
        Runnable barrierAction =
            new Runnable() { public void run() { mergeRows(...); } };
        barrier = new CyclicBarrier(N, barrierAction);
    }
}
```

```
List<Thread> threads = new ArrayList<Thread>(N);
for (int i = 0; i < N; i++) {
    Thread thread = new Thread(new Worker(i));
    threads.add(thread);
    thread.start();
}

// wait until done
for (Thread thread : threads)
    thread.join();
}
```

Here, each worker thread processes a row of the matrix then waits at the barrier until all rows have been processed.

When all rows are processed the supplied Runnable barrier action is executed and merges the rows. If the merger determines that a solution has been found then done() will return true and each worker will terminate.

The CyclicBarrier class allows the design of complex, multi-threaded tasks, while all threads stop and wait at logical barriers. This solution is superior to a single-threaded solution, as the individual tasks can be completed in parallel by all worker threads.

There is a slight loss in performance to be expected from using a CyclicBarrier. For example, one worker thread may be slower so the others will have to wait for it to finish. However, this is really the point of using a CyclicBarrier.

After a CyclicBarrier is broken, all threads are released and the number of threads waiting on the CyclicBarrier goes back to zero.

At this point, the CyclicBarrier may be used again for a new set of waiting threads. For example, if our CyclicBarrier limit is 5 and we have 15 threads that call await(), then the CyclicBarrier will be activated a total of three times.

Fork-Join Framework

The Fork-Join Framework provides a highly specialised ExecutorService.

The other ExecutorService instances you've seen so far are centred on the concept of submitting multiple tasks to an ExecutorService. By doing this, you provide an easy avenue for an ExecutorService to take advantage of all the CPUs in a system by using multiple threads to complete tasks. Sometimes, you don't have multiple tasks; instead, you have one really big task.

There are many large tasks or problems you might need to solve in your application. For example, you might need to initialise the elements of a large array with values. You might think that initialising an array doesn't sound like a large complex task in need of a framework. The key is that it needs to be a large task.

What if you need to fill up a 100,000,000-element array with randomly generated values? The Fork-Join Framework makes it easier to tackle big tasks like this, while making use of all of the CPUs in a system.

The no-args ForkJoinPool constructor creates an instance that will use the `Runtime.availableProcessors()` method to determine the level of parallelism. The level of parallelism determines the number of threads that will be used by the ForkJoinPool.

There is also a `ForkJoinPool(int parallelism)` constructor that allows you to override the number of threads that will be used.

Recursion

The fork/join framework relies on the concept of recursion to solve complex tasks. Recursion is the process by which a task calls itself to solve a problem. A recursive solution is constructed with a base case and a recursive case:

- Base case
A non-recursive method that is used to terminate the recursive path
- Recursive case
A recursive method that may call itself one or multiple times to solve a problem

For example, a method that computes the factorial of a number can be expressed as a recursive function.

In mathematics, a factorial is what you get when you multiply a number by all of the integers below it. The factorial of 5 is equal to $5 * 4 * 3 * 2 * 1 = 120$.

The following is a recursive factorial function in Java:

```
public static int factorial (int n) {  
    if(n<=1){  
        return 1;  
    }else{  
        return n * factorial (n-1);  
    }  
}
```

```
}
```

In this example, you see that 1 is the base case, and any integer value greater than 1 triggers the recursive case.

One challenge in implementing a recursive solution is always to make sure that the recursive process arrives at a base case. For example, if the base case is never reached, the solution will continue infinitely and the program will hang.

In Java, this will result in a StackOverflowError anytime the application recurses too deeply.

Let's say we have a List of long integers that we need to check and say which are prime numbers. Here is a method that checks a number to see if it is prime and returns a boolean result:

```
public static boolean isPrime(long potentialPrime) {
    if(potentialPrime <= 1){
        return false;
    }else if(potentialPrime <= 3){
        return true;
    }else if(potentialPrime % 2 == 0 || potentialPrime % 3 == 0){
        return false;
    }
    long divisor = 5;
    while(divisor * divisor <= potentialPrime) {
        if (((potentialPrime % divisor == 0) ||
            (potentialPrime % (divisor + 2) == 0))){
            return false;
        }
        divisor = divisor + 6;
    }
    return true;
}
```

Let's use an ArrayList of Long values called candidates.

```
ArrayList<Long> candidates = new ArrayList<>();
```

Each number could take a long time or a short time to check. We need to divide and conquer otherwise we could be here for a long time if we try to do it in one pass.

Applying the fork/join framework requires us to perform three steps:

1. Create a ForkJoinTask.
2. Create the ForkJoinPool
3. Start the ForkJoinTask

The first step is the most complex, as it requires defining the recursive process.

Fortunately, the second and third steps are easy and can each be completed with a single line of code.

Fork/Join Task

With the Fork-Join Framework, a `java.util.concurrent.ForkJoinTask` instance (actually one of its subclasses – `RecursiveAction` or `RecursiveTask`) is created to represent the task that should be accomplished. This is different from other executor services that primarily used either `Runnable` or `Callable`. A `ForkJoinTask` has many methods (most of which you will never use), but the following methods are important: `compute()`, `fork()`, and `join()`.

A `ForkJoinTask` subclass is where you will perform most of the work involved in completing a Fork-Join task. The basic structure of any `ForkJoinTask` is shown in this pseudocode example:

```
class ForkJoinSomeTask {  
    compute() {  
        if(isTaskSectionSmall()) { // is it a manageable amount of work?  
            PerformTaskSection(); // do the task  
        } else { // task too big, split it  
            ForkJoinSomeTask leftHalf = getLeftHalfOfTask();  
            leftHalf.fork(); // queue left half of task  
            ForkJoinSomeTask rightHalf = getRightHalfOfTask();  
            rightHalf.compute(); // work on right half of task  
            leftHalf.join(); // wait for queued task to be complete  
        }  
    }  
}
```

You need to know how to implement the fork/join solution by extending one of two classes, `RecursiveAction` and `RecursiveTask`, both of which implement the `ForkJoinTask` interface.

The first class, `RecursiveAction`, is an abstract class that requires us to implement the `compute()` method, which returns `void`, to perform the bulk of the work.

The second class, `RecursiveTask`, is an abstract generic class that requires us to implement the `compute()` method, which returns the generic type, to perform the bulk of the work.

`RecursiveAction` and `RecursiveTask` are analogous to the `Runnable` and `Callable` interfaces, respectively, which you saw at the start of the section on threads.

With the Fork-Join Framework, each thread in the `ForkJoinPool` has a queue of the tasks it is working on; this is unlike most `ExecutorService` implementations that have a single shared task queue.

The `fork()` method places a `ForkJoinTask` in the current thread's task queue. A normal thread does not have a queue of tasks, only the specialised threads in a `ForkJoinPool` do. This means that you can't call `fork()` unless you are within a `ForkJoinTask` that is being executed by a `ForkJoinPool`.

Initially, only a single thread in a `ForkJoinPool` will be busy when you submit a task. That thread will begin to subdivide the tasks into smaller tasks. Each time a task is subdivided into two subtasks, you `fork` (or `queue`) the first task and `compute` the second task. In the

event you need to subdivide a task into more than two subtasks, each time you split a task, you would fork every new subtask except one (which would be computed).

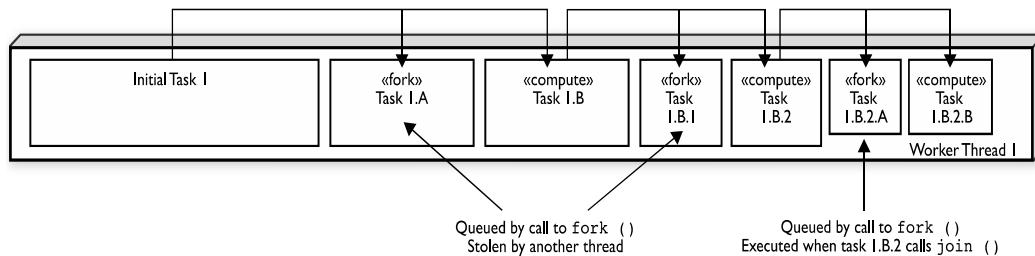


Figure 7-4 Worker Thread Task Queue

Work Stealing

Notice how the call to `fork()` is placed before the call to `compute()` or `join()`.

A key feature of the Fork-Join Framework is work stealing. Work stealing is how the other threads in a `ForkJoinPool` will obtain tasks. When initially submitting a Fork-Join task for execution, a single thread from a `ForkJoinPool` begins executing (and subdividing) that task.

Each call to `fork()` places a new task in the calling thread's task queue. The order in which the tasks are queued is important. The tasks that have been queued the longest represent larger amounts of work.

In the `ForkJoinSomeTask` example, the task that represents 100 percent of the work would begin executing, and its first queued (forked) task would represent 50 percent of the task, the next 25 percent, then 12.5 percent, and so on. Of course, this can vary, depending on how many times the task will be subdivided and whether we are splitting the task into halves or quarters or some other division, but in this example, we are splitting each task into two parts: queuing one part and executing the second part.

The non-busy threads in a `ForkJoinPool` will attempt to steal the oldest (and therefore largest) task from any Fork-Join thread with queued tasks.

Given a `ForkJoinPool` with four threads, one possible sequence of events could be that the initial thread queues tasks that represent 50 percent and 25 percent of the work, which are then stolen by two different threads. The thread that stole the 50 percent task then subdivides that task and places a 25 percent task on its queue, which is then stolen by a fourth thread, resulting in four threads that each process 25 percent of the work.

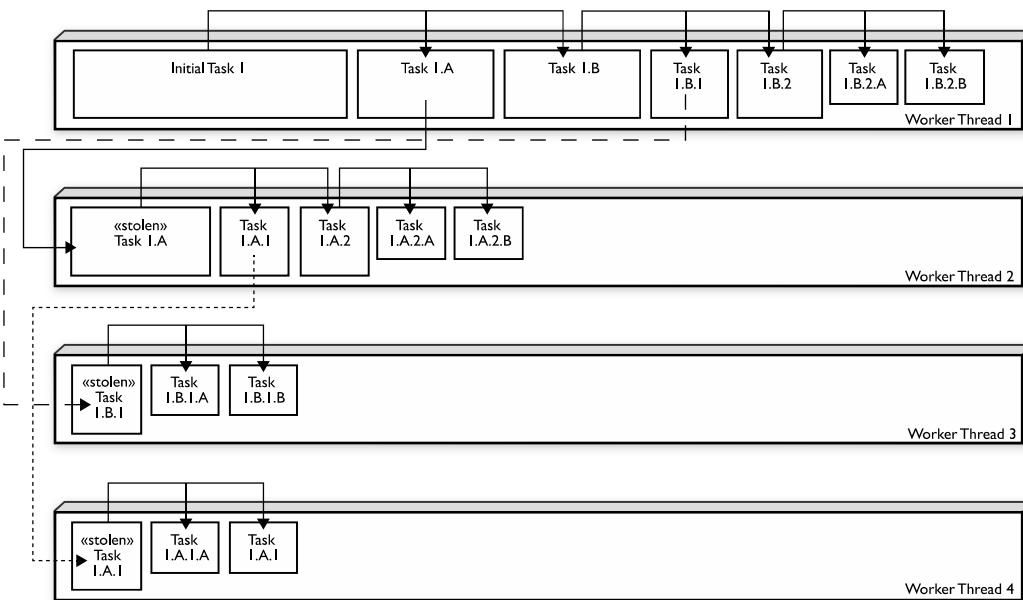


Figure 7-5 Fork/Join Framework Work Stealing

Join

When you call `join()` on the (left) task, it should be one of the last steps in the `compute()` method, after calling `fork()` and `compute()`. Calling `join()` says "I can't proceed unless this (left) task is done." Several possible things can happen when you call `join()`. The task you call `join()` on might:

- already be done. Remember you are calling `join()` on a task that already had `fork()` called. The task might have been stolen and completed by another thread. In this case, calling `join()` just verifies the task is complete and you can continue on.
- be in the middle of being processed. Another thread could have stolen the task, and you'll have to wait until the joined task is done before continuing.
- still be in the queue (not stolen). In this case, the thread calling `join()` will execute the joined task.

RecursiveAction

`ForkJoinTask` is an abstract base class that outlines most of the methods, such as `fork()` and `join()`, in a Fork-Join task. If you need to create a `ForkJoinTask` that does not return a result, then you should subclass `RecursiveAction`. `RecursiveAction` extends `ForkJoinTask` and has a single abstract `compute` method that you must implement:

```
protected abstract void compute();
```

An example of a task that does not need to return a result would be any task that initialises an existing data structure.

The following example will initialise an array to contain random values. Notice that there is only a single array throughout the entire process. When subdividing an array, you should avoid creating new objects when possible.

```

public class RandomInitRecursion extends RecursiveAction {
    private static final int THRESHOLD = 10000;
    private int[] data;
    private int start;
    private int end;
    public RandomInitRecursion(int[] data, int start, int end) {
        this.data = data;
        this.start = start; // where does our section begin at?
        this.end = end; // how large is this section?
    }
    @Override
    protected void compute() {
        if (end - start <= THRESHOLD) {
            // is it a manageable amount of work?
            // do the task
            for (int i = start; i < end; i++) {
                data[i] = ThreadLocalRandom.current().nextInt();
            }
        } else { // task too big, split it
            int halfWay = ((end - start) / 2) + start;
            RandomInitRecursion a1 =
                new RandomInitRecursion(data, start, halfWay);
            a1.fork();
            // queue left half of task
            RandomInitRecursion a2 =
                new RandomInitRecursion(data, halfWay, end);
            a2.compute(); // work on right half of task
            a1.join(); // wait for queued task to be complete
        }
    }
}

```

Sometimes, you will see one of the invokeAll methods from the ForkJoinTask class used in place of the fork/compute/join method combination.

The invokeAll methods are convenience methods that can save some typing. Using them will also help you avoid bugs! The first task passed to invokeAll will be executed (compute is called), and all additional tasks will be forked and joined. In the preceding example, you could eliminate the three fork/compute/join lines and replace them with a single line:

```
i invokeAll(a2, a1);
```

To begin the application, we create a large array and initialise it using Fork-Join:

```

public static void main(String[] args) {
    int[] data = new int[10_000_000];
    ForkJoinPool fjPool = new ForkJoinPool();
    RandomInitRecursion action =
        new RandomInitRecursion(data, 0, data.length);

```

```

    f j Pool . i nvoke(activ e);
}

```

Notice that we do not expect any return values when calling invoke.

A RecursiveAction returns nothing.

RecursiveTask

If you need to create a ForkJoinTask that does return a result, then you should subclass RecursiveTask.

RecursiveTask extends ForkJoinTask and has a single abstract compute method that you must implement:

```
protected abstract V compute(); // V is a generic type
```

The following example will find the position in an array with the greatest value; if duplicate values are found, the first occurrence is returned. Notice that there is only a single array throughout the entire process.

```

public class FindMaxPositionRecursiveTask extends
    RecursiveTask<Integer> {
    private static final int THRESHOLD = 10000;
    private int[] data;
    private int start;
    private int end;
    public FindMaxPositionRecursiveTask(int[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }
    @Override
    protected Integer compute() {
        // return type Integer matches the <generic> type
        if (end - start <= THRESHOLD) {
            // is it a manageable amount of work?
            int position = 0; // if all values are equal, return position 0
            for (int i = start; i < end; i++) {
                if (data[i] > data[position]) {
                    position = i;
                }
            }
            return position;
        } else { // task too big, split it
            int halfWay = ((end - start) / 2) + start;
            FindMaxPositionRecursiveTask t1 =
                new FindMaxPositionRecursiveTask(data, start, halfWay);
            t1.fork(); // queue left half of task
            FindMaxPositionRecursiveTask t2 =
                new FindMaxPositionRecursiveTask(data, halfWay, end);
            int position2 = t2.compute(); // work on right half of task
            int position1 = t1.join(); // wait for queued task to be complete
        }
    }
}

```

```

        // out of the position in two subsection which is greater?
        if (data[position1] > data[position2]) {
            return position1;
        } else if (data[position1] < data[position2]) {
            return position2;
        } else {
            return position1 < position2 ? position1 : position2;
        }
    }
}
}

```

To begin the application, we reuse the RecursiveAction example to create a large array and initialise it using Fork-Join.

After initialising the array with random values, we reuse the ForkJoinPool with our RecursiveTask to find the position with the greatest value:

```

public static void main(String[] args) {
    int[] data = new int[10_000_000];
    ForkJoinPool fjPool = new ForkJoinPool();
    RandomInitRecursiveAction action =
        new RandomInitRecursiveAction(data, 0, data.length);
    fjPool.invoke(action);
    // new code begins here
    FindMaxPositionRecursiveTask task =
        new FindMaxPositionRecursiveTask(data, 0, data.length);
    Integer position = fjPool.invoke(task);
    System.out.println("Position: " + position + ", value: " +
        data[position]);
}

```

Notice that a value is returned by the call to invoke when using a RecursiveTask.

If your application will repeatedly submit tasks to a ForkJoinPool, then you should reuse a single ForkJoinPool instance and avoid the overhead involved in creating a new instance.

Threading Problems

A threading problem can occur in multi-threaded applications when two or more threads interact in an unexpected and undesirable way. For example, two threads may block each other from accessing a particular segment of code.

The Concurrency API was created to help eliminate potential threading issues common to all developers. As you have seen, the Concurrency API creates threads and manages complex thread interactions for you, often in just a few lines of code.

Although the Concurrency API reduces the potential for threading issues, it does not eliminate it. In practice, finding and identifying threading issues within an application is often one of the most difficult tasks a developer can undertake.

Liveness

As you have seen in this chapter, many thread operations can be performed independently, but some require coordination. For example, synchronising on a method requires all threads that call the method to wait for other threads to finish before continuing. You also saw earlier in the chapter that threads in a CyclicBarrier will each wait for the barrier limit to be reached before continuing.

What happens to the application while all of these threads are waiting? In many cases, the waiting is momentary and the user has very little idea that any delay has occurred. In other cases, though, the waiting may be extremely long, perhaps infinite.

Liveness is the ability of an application to be able to execute in a timely manner. Liveness problems, then, are those in which the application becomes unresponsive or stuck. There are three types of liveness issues with which you should be familiar: deadlock, starvation, and livelock.

Thread Deadlock

Perhaps the worst thing that can happen to a Java program is deadlock.

Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

This can happen, for example, when thread A hits synchronised code, acquires a lock B, and then enters another method (still within the synchronised code it has the lock on) that's also synchronised.

But thread A can't get the lock to enter this synchronised code – block C – because another thread D has the lock already.

So thread A goes off to the waiting-for-the-C-lock pool, hoping that thread D will hurry up and release the lock (by completing the synchronised method). But thread A will wait a very long time indeed, because while thread D picked up lock C, it then entered a method synchronised on lock B.

Obviously, thread D can't get the lock B because thread A has it. And thread A won't release it until thread D releases lock C. But thread D won't release lock C until after it can get lock B and continue. And there they sit.

How do you fix a deadlock once it has occurred? The answer is that you can't in most situations. On the other hand, there are numerous strategies to help prevent deadlocks from ever happening in the first place. One common strategy to avoid deadlocks is for all threads to perform their resource requests in the same order. This strategy would have prevented the deadlock described above.

Starvation

Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. The thread is still active, but it is unable to complete its work as a result of other threads constantly taking the resource that they trying to access.

Livelock

Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Livelock is a special case of resource starvation in which two or more threads actively try to acquire a set of locks, are unable to do so, and restart part of the process.

Race Conditions

A race condition is an undesirable result that occurs when two tasks, which should be completed sequentially, are completed at the same time. We encountered two examples of race conditions earlier in the chapter when we introduced synchronisation and parallel streams.

Imagine two users are signing up for an account on a website. Both of them want to use the same username (which is not currently in use) and they each send requests to create the account at the same time.

Possible outcomes for this race condition are:

- Both users are able to create accounts with the same username
- Both users are unable to create an account with the same username but returning an error message to both users
- One user is able to create the account with their chosen username, while the other user receives an error message

Which of these results is most desirable when designing our web server?

The first possibility, in which both users are able to create an account with the same username, could cause serious problems and break numerous invariants in the system. Assuming that the username is required to log into the website, how do they both log in with the same username and different passwords? In this case, the website cannot tell

them apart. This is the worst possible outcome to this race condition, as it causes significant and potentially unrecoverable data problems.

What about the second scenario? If both users are unable to create the account, both will receive error messages and be told to try again. In this scenario, the data is protected since no two accounts with the same username exist in the system. The users are free to try again with the same username, since no one has been granted access to it. Although this might seem like a form of livelock, there is a subtle difference. When the users try to create their account again, the chances of them hitting a race condition tend to diminish. For example, if one user submits their request a few seconds before the other, they might avoid another race condition entirely by the system informing the second user that the account name is already in use.

The third scenario, in which one user obtains the account while the other does not, is often considered the best solution to this type of race condition. Like the second situation, we preserve data integrity, but unlike the second situation, at least one user is able to move forward on the first request, avoiding additional race condition scenarios. Also unlike the previous scenario, we can provide the user who didn't win the race with a clearer error message because we are now sure that the account username is no longer available in the system.

you should understand that race conditions lead to invalid data if they are not properly handled. Even the solution where both participants fail to proceed is preferable to one in which invalid data is permitted to enter the system.

Race conditions tend to appear in highly concurrent applications. As a software system grows and more users are added, they tend to appear more frequently. One solution is to use a monitor to synchronise on the relevant overlapping task. In the previous example, the relevant task is the method that determines whether an account username is in use and reserves it in the system if it is available. We could also use singletons, described earlier to coordinate access to shared resources.

Summary

This chapter introduced you to threads and showed you how to process tasks in parallel using the Concurrency API.

You should know how to create threads indirectly using an ExecutorService and a fork/join recursive framework. The work that a thread performs can be defined in an instance of Runnable or Callable. As of Java 8, these tasks can now be expressed as lambda expressions.

We presented techniques for organising tasks among multiple threads using atomic classes, synchronisation blocks, synchronised methods, and the CyclicBarrier class.

The Concurrency API includes numerous collections classes that support multi-threaded processing. You should also be familiar with the CopyOnWriteArrayList class, which creates a new underlying structure anytime the list is modified.

We then introduced the notion of parallel streams and showed you how using them can improve performance in your application. Parallel streams can also cause unexpected results, since the results are no longer ordered. We also reviewed parallel reductions and showed how they differed from reductions on serial streams.

We concluded this chapter by discussing potential threading issues with which you should be familiar for the exam including deadlock, starvation, livelock, and race conditions. In professional software development, finding and resolving such problems is often quite challenging.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 8

10

Introduction

I/O (input/output) has been around since the very first version of Java.

You could read and write files along with some other common operations. Then with Java 1.4, Java added more I/O functionality and cleverly named it NIO. That stands for "new I/O."

The APIs prior to Java 7 still had a few limitations when you had to write applications that focused heavily on files and file manipulation. Trying to write a little routine listing all the files created in the past day within a directory tree would give you some headaches. There was no support for navigating directory trees, and just reading attributes of a file was also quite hard. From Java 7, this whole routine takes less than 15 lines of code!

But what could Oracle name yet another I/O API? The name "new I/O" was taken, and "new new I/O" would just sound silly.

Since the Java 7 functionality was added to package names that begin with `java.nio`, the new name was chosen as NIO.2. For the purposes of this chapter and the exam, NIO is shorthand for NIO.2.

This chapter focuses on using the `java.io` API to interact with files and streams.

We start by describing how files and directories are organised within a file system and show how to access them with the `java.io.File` class. We then show how to read and write file data with the stream classes. Finally, we conclude this chapter by discussing ways of reading user input at runtime using the `Console` class.

In the next chapter, "NIO.2," we will revisit the discussion of files and show how Java now provides more powerful techniques for managing them.

Files and Directories

We begin this chapter by describing what a file is and what a directory is within a file system. We then present the `java.io.File` class and demonstrate how to use it to read and write file information.

File System Terminology

Before we start working with files and directories, we present the terminology that we will be using throughout this chapter.

A file is record within a file system that stores user and system data. Files are organised into directories. A directory is a record within a file system that contains files as well as other directories.

For simplicity, we often refer to a directory reference as a file record throughout this chapter, since it is stored in the file system with a unique name and with attributes similar to a file. For example, a file and directory can both be renamed with the same operation.

Finally, the root directory is the topmost directory in the file system, from which all files and directories inherit.

In Windows, it is denoted with a drive name such as `c:\`, while on Linux/Unix/Mac it is denoted with a single forward slash `/`.

In order to interact with files, we need to connect to the file system. The file system is responsible for reading and writing data within a computer. The file system is a component of the operating system. The Windows file system provides slightly different functionality from Unix-based ones, specifically around file attributes. There is a lot of overlap though and much of Java IO is portable between Windows and UNIX.

Java has numerous methods, which automatically connect to the local file system for you, allowing you to perform the same operations across multiple file systems.

A path is a String representation of a file or directory within a file system. Each file system defines its own path separator character that is used between directory entries. In most file systems, the value to the left of a separator is the parent of the value to the right of the separator.

For example, the path value `/user/home/File.txt` means that the file `File.txt` is inside the `home` directory, which is inside the `user` directory. Paths can be absolute (from the root) or relative (from another directory).

The File Class

The first class that we will discuss is one of the most commonly used in the `java.io` API, the `java.io.File` class, or `File` class for short. The `File` class is used to read information about existing files and directories, list the contents of a directory, and create/delete files and directories.

An instance of a File class represents the pathname of a particular file or directory on the file system. The File class cannot read or write data within a file, although it can be passed as a reference to many stream classes to read or write data.

Creating a File Object

A File object often is initialised with String containing either an absolute or relative path to the file or directory within the file system. The absolute path of a file or directory is the full path from the root directory to the file or directory, including all subdirectories that contain the file or directory. Alternatively, the relative path of a file or directory is the path from the current working directory to file or directory.

For example, the following is an absolute path to the Data.txt file:

```
/home/data/Data.txt
```

The following is a relative path to the same file, assuming the user's current directory was set to /home:

```
data/Data.txt
```

Different operating systems vary in their format of path names. For example, Unix-based systems use the forward slash / for paths, whereas Windows-based systems use the backslash \ character. That said, many programming languages and file systems support both types of slashes when writing path statements.

For convenience, Java offers two options to retrieve the local separator character: a system property and a static variable defined in the File class. Both of the following examples will output the separator character:

```
System.out.println(System.getProperty("file.separator"));
System.out.println(java.io.File.separator);
```

The following code creates a File object and determines if the path it references exists within the file system:

```
import java.io.File;
public class FileSample {
    public static void main(String[] args) {
        File file = new File("/home/data/Data.txt");
        System.out.println(file.exists());
    }
}
```

This example uses the absolute path to a file and outputs true or false, depending on whether the file exists. The most common File constructor we will use throughout this chapter takes a single String as an argument representing the relative or absolute path.

There are other constructors, such as the one that joins an existing File path with a relative child path, as shown in the following example:

```
File parent = new File("/home");
File child = new File(parent, "data/Data.txt");
```

In this example, we create a path that is equivalent to our previous example, using a combination of a child path and a parent path.

If the parent object happened to be null, then it would be skipped and the method would revert to our single String constructor.

File Objects

The File class contains numerous useful methods for interacting with files and directories within the file system.

The most commonly used ones are listed in the following table.

There are quite a few methods here but many of them are self-explanatory.

Method	Description
exists()	Returns true if the file or directory exists.
getName()	Returns the name of the file or directory denoted by this path.
getAbsolutePath()	Returns the absolute pathname string of this path.
isDirectory()	Returns true if the file denoted by this path is a directory.
isFile()	Returns true if the file denoted by this path is a file.
length()	Returns the number of bytes in the file. For performance reasons, the file system may allocate more bytes on disk than the file actually uses.
lastModified()	Returns the number of milliseconds since the epoch when the file was last modified.
delete()	Deletes the file or directory. If this pathname denotes a directory, then the directory must be empty in order to be deleted.
renameTo(File)	Renames the file denoted by this path.
mkdir()	Creates the directory named by this path.
mkdirs()	Creates the directory named by this path including any non-existent parent directories.
getParent()	Returns the abstract pathname of this abstract pathname's parent or null if this pathname does not name a parent directory.
listFiles()	Returns a File[] array denoting the files in the directory.

The following sample code outputs information about a file or directory including whether it exists, details about the path and what files are contained within it if it is a directory:

```

import java.io.File;
public class ReadFileInformation {
    public static void main(String[] args) {
        File file = new File(args[0]);
        System.out.println("File Exists: " + file.exists());
        if(file.exists()) {
            System.out.println("Absolute Path: " + file.getAbsolutePath());
            System.out.println("Is Directory: " + file.isDirectory());
            System.out.println("Parent Path: " + file.getParent());
            if(file.isFile()) {
                System.out.println("File size: " + file.length());
                System.out.println("File LastModified: " +
                                   file.lastModified());
            } else {
                System.out.println("Files in directory:");
                for(File subfile: file.listFiles()) {
                    System.out.println("\t" + subfile.getName());
                }
            }
        }
    }
}

```

If the path provided pointed to a valid file, it would output something similar to the following:

```

File Exists: true
Absolute Path: C:\data\Data.txt
Parent Path: C:\data
Is Directory: false
File size: 12382
File LastModified: 1420070400000

```

In this example, you can see that the output of an I/O-based program is completely dependent on the directories and files available at runtime in the underlying file system.

Note that we used a Windows-based path in the previous sample, which requires a double backslash in the String literal for the path separator. You may remember that the backslash \ is a reserved character within a String literal and must be escaped with another backslash to be used within a String.

IO Streams

In this section, we present the concept of streams in Java and show how they are used for input/output (I/O) processing. I/O refers to the nature of how data is accessed, either by reading the data from a resource (input), or writing the data to a resource (output). This section will focus on the common ways that data is input and output using files accessed by streams.

Note that the I/O streams that we discuss in this chapter are data streams and completely unrelated to the new Stream API that you saw in an earlier chapter. The naming of the new Stream API can be a little confusing when discussing I/O streams.

Stream Fundamentals

The contents of a file may be accessed or written via a stream, which is a list of data elements presented sequentially. Streams should be conceptually thought of as a long, nearly never-ending “stream of water” with data presented one bucket at a time.

It may be helpful to visualise a stream as being so large that all of the data contained in it could not possibly fit into memory.

For example, a 1 terabyte file could not be stored entirely in memory by most computer systems. However, the file can still be read and written by a program with very little memory because the stream allows the application to focus on only a small portion of the overall stream at any given time.

In this sense, there is some similarity between the processing of IO streams and the use of the Stream API.

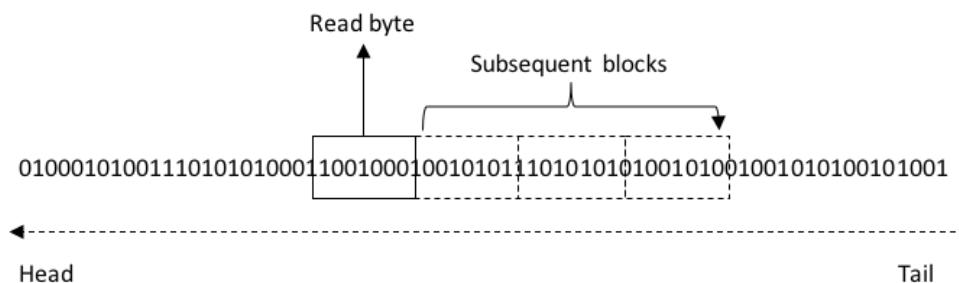


Figure 10-1 Reading Bytes from a File Stream

Each type of stream segments data into blocks in a specific way.

For example, some stream classes read or write data as individual byte values. Other stream classes read or write individual characters or strings of characters. On top of that, some stream classes read or write groups of bytes or characters at a time, specifically those with the word *Buffered* in their name.

Although streams are commonly used with file I/O, they are more generally used to handle reading/writing of a sequential data source. For example, you might construct a

Java application that submits data to a website using an input stream and reads the result via an output stream.

You have been using streams without knowing it since your first “Hello World” program!

Java provides three built-in streams, System.in, System.err, and System.out, the last of which we have been using to output data to the screen throughout this book. We will discuss these three streams later in this chapter when we discuss user input.

Stream Names

The java.io API provides numerous classes for creating, accessing, and manipulating streams. There are so many that it can be overwhelming.

This section reviews the major differences between each stream class and show you how to distinguish between them.

Even if you do come across a particular stream on the exam that you do not recognise, often the name of the stream gives you enough information to understand exactly what it does.

Byte Streams vs. Character Streams

The java.io API defines two sets of classes for reading and writing streams:

- those with Stream in their name
- those with Reader or Writer in their name

For example, the java.io API defines both a FileInputStream class as well as a FileReader class, both of which define a stream that reads a file. The difference between the two classes is based on how the stream is read or written.

Differences between Streams and Readers/Writers

1. The stream classes are used for inputting and outputting all types of binary or byte data.
2. The reader and writer classes are used for inputting and outputting only character and String data.

It is important to remember that even though readers/writers do not contain the word Stream in their class name, they are still in fact streams. The use of Reader/Writer in the name is just to distinguish them from byte streams. Throughout the chapter, we will often refer to Reader/Writer classes as streams, since conceptually they are streams.

The java.io API is structured such that all the stream classes have the word InputStream or OutputStream in their name, while all Reader/Writer classes have either Reader or Writer in their name.

Input and Output

Most Input stream classes have a corresponding Output class and vice versa. For example, the FileOutputStream class writes data that can be read by a FileInputStream. If

you understand the features of a particular Input or Output stream class, you should naturally know what its complementary class does.

It follows, then, that most Reader classes have a corresponding Writer class. For example, the `FileWriter` class writes data that can be read by a `FileReader`.

There are exceptions to this rule. For the exam, you should know that `PrintWriter` has no accompanying `PrintReader` class. Likewise, the `PrintStream` class has no corresponding `InputStream` class. We will discuss these classes later this chapter.

Low- and High-Level Streams

Another way that you can familiarise yourself with the `java.io` API is by segmenting streams into low-level and high-level streams.

A low-level stream connects directly with the source of the data, such as a file, an array, or a `String`. Low-level streams process the raw data or resource and are accessed in a direct and unfiltered manner. For example, a `FileInputStream` is a class that reads file data one byte at a time.

Alternatively, a high-level stream is built on top of another stream using wrapping or decorating. The Decorator design pattern is based on this principle. An instance of the class to be decorated is passed to the constructor of the decorator class and stored as an instance variable. Method calls supported by the original class are exposed through the interface of the decorator and calls to them are delegated to the original class using the instance variable. The decorator class can add as many methods to enhance functionality as are needed. For example a `BufferedReader` has a `readLine()` method that uses a `FileReader`'s `read()` method to read characters until the end of the line.

Here is some sample code:

```
try (BufferedReader bufferedReader = new BufferedReader(
        new FileReader("Data.txt"))) {
    System.out.println(bufferedReader.readLine());
}
```

In this example, `FileReader` is the low-level stream reader, whereas `BufferedReader` is the high-level stream that takes a `FileReader` as input. Many operations on the high-level stream pass through as operations to the underlying low-level stream, such as `read()` or `close()`. This is an example of delegation.

Other operations override or add new functionality to the low-level stream methods. The high-level stream adds new methods, such as `readLine()`, as well as performance enhancements for reading and filtering the low-level data.

Stream Base Classes

The `java.io` library defines four abstract classes that are the parents of all stream classes defined within the API:

- `InputStream`

- OutputStream
- Reader
- Writer

For convenience, the authors of the Java API include the name of the abstract parent class as the suffix of the child class. For example, ObjectInputStream ends with InputStream, meaning it has InputStream as an inherited parent class.

Although most stream classes in java.io follow this pattern, PrintStream, which is an OutputStream, does not.

The constructors of high-level streams often take a reference to the abstract class. For example, BufferedWriter takes a Writer object as input, which allows it to take any subclass of Writer.

The advantage of using a reference to the abstract parent class in the class constructor should be apparent in the previous high-level stream example. With high level-streams, a class may be wrapped multiple times. Furthermore, developers may define their own stream subclass that performs custom filtering.

By using the abstract parent class as input, the high-level stream classes can be used much more often without concern for the particular under-lying stream subclass.

Java I/O Class Names

The function of most stream classes can be understood by decoding the name of the class. The list of rules is shown here

- A class with the word InputStream or OutputStream in its name is used for reading or writing binary data, respectively
- A class with the word Reader or Writer in its name is used for reading or writing character or string data, respectively
- Most, but not all, input classes have a corresponding output class
- A low-level stream connects directly with the source of the data
- A high-level stream is built on top of another stream using wrapping (decoration)
- A class with Buffered in its name reads or writes data in groups of bytes or characters and often improves performance in sequential file systems
- When wrapping a stream you can mix and match only types that inherit from the same abstract parent stream

This table describes the most important java.io streams:

Class	Description
InputStream	The abstract superclass of all InputStream classes
OutputStream	The abstract superclass of all OutputStream classes
Reader	The abstract superclass of all Reader classes
Writer	The abstract superclass of all Writer classes

FileInputStream	Reads file data as bytes
FileOutputStream	Writes file data as bytes
FileReader	Reads file data as characters
FileWriter	Writes file data as characters
BufferedReader	Decorates character Reader in a buffered manner (e.g. reading lines), which improves efficiency and performance
BufferedWriter	Decorates character Writer in a buffered manner (e.g. writing lines), which improves efficiency and performance
ObjectInputStream	Decorates an InputStream to de-serialise primitive Java data types and objects
ObjectOutputStream	Decorates an OutputStream to serialise primitive Java data types and objects
InputStreamReader	Reads character data from an existing InputStream
OutputStreamWriter	Writes character data to an existing OutputStream
PrintStream	Writes formatted representations of Java objects to a binary output stream
PrintWriter	Writes formatted representations of Java objects to a text-based output stream

Common Stream Operations

Before looking into specific stream classes, let's review some common processes when working with streams.

Closing the Stream

Since streams are considered resources, it is imperative that they be closed after they are used lest they lead to resource leaks. As you saw earlier in the course, you can accomplish this by calling the `close()` method in a `finally` block or using the try-with-resource syntax.

In a file system, failing to close a file properly could leave it locked by the operating system such that no other processes could read/write to it until the program is terminated.

Throughout this section, we will close stream resources using the try-with-resource syntax, since this is the preferred way of closing resources in Java.

Flushing the Stream

When data is written to an `OutputStream`, the underlying operating system does not necessarily guarantee that the data will make it to the file immediately. In many operating

systems, the data may be cached in memory, with a write occurring only after a temporary cache is filled or after some amount of time has passed.

If the data is cached in memory and the application terminates unexpectedly, the data would be lost, because it was never written to the file system. To address this, Java provides a `flush()` method, which requests that all accumulated data be written immediately to disk.

The `flush()` method helps reduce the amount of data lost if the application terminates unexpectedly. It is not without cost, though. Each time it is used, it may cause a noticeable delay in the application, especially for large files. Unless the data that you are writing is extremely critical, the `flush()` method should only be used intermittently. For example, it should not necessarily be called after every write but after every dozen writes or so, depending on your requirements. For reasonably small files, you may need to call `flush()` only once.

You do not need to call the `flush()` method explicitly when you have finished writing to a file, since the `close()` method will automatically do this. In some cases, calling the `flush()` method intermittently while writing a large file, rather than performing a single large flush when the file is closed, may appear to improve performance by stretching the disk access over the course of the write process.

Marking the Stream

The `InputStream` and `Reader` classes include `mark(int)` and `reset()` methods to move the stream back to an earlier position.

Before calling either of these methods, you should call the `markSupported()` method, which returns true only if `mark()` is supported. Not all `java.io` input stream classes support this operation, and trying to call `mark(int)` or `reset()` on a class that does not support these operations will throw an exception at runtime.

Once you've verified that the stream can support these operations, you can call `mark(int)` with a read-ahead limit value. You can then read as many bytes as you want up to the limit value. If at any point you want to go back to the earlier position where you last called `mark()`, then you just call `reset()` and the stream will revert to an earlier state.

This operation is not actually putting the data back into the stream but storing the data that was already read into memory for you to read again. Therefore, you should not call the `mark()` operation with too large a value as this could take up a lot of memory.

Assume that we have an `InputStream` instance whose next values are 1234. Consider the following code snippet:

```
InputStream stream = ...  
System.out.print((char)stream.read());  
if(stream.markSupported()) {  
    stream.mark(100);  
    System.out.print((char)stream.read());  
    System.out.print((char)stream.read());  
    stream.reset();  
}
```

```
System.out.print((char)stream.read());
System.out.print((char)stream.read());
System.out.print((char)stream.read());
```

The code snippet will output the following if the mark() operation is supported:

123234

It first outputs 1 before the if/then statement. Assuming that the stream supports the mark() operation, it will enter the if/then statement and read two characters, 23. It then calls the reset() operation, moving our stream back to the state that it was in after the A was read, therefore 23 are read again, followed by 4.

If the mark() operation is not supported, it will just output:

1234

Finally, if you call reset() after you have passed your mark() read limit, an exception may be thrown at runtime since the marked position may become invalidated. The exception may not be thrown as some implementations may use a buffer to allow extra data to be read before the mark is invalidated.

Skipping over Data

The InputStream and Reader classes also include a skip(long) method, which as you might expect skips over a certain number of bytes.

It returns a long value, which indicates the number of bytes that were actually skipped. If the return value is zero or negative, such as if the end of the stream was reached, no bytes were skipped.

Assume that we have an InputStream instance whose next values are COMPUTER. Consider the following code snippet:

```
InputStream stream = ...
System.out.print((char)stream.read());
stream.skip(2)
stream.read();
System.out.print((char)stream.read());
System.out.print((char)stream.read());
```

The code will read one character, C, skip two characters, OM, and then read three more characters, PUT, only the last two of which are printed to the user, which results in the following output.

CUT

You may notice in this example that calling the skip() operation is equivalent to calling read() and discarding the output.

For skipping a handful of bytes, there is virtually no difference. On the other hand, for skipping a large number of bytes, skip() will often be faster, because it will use arrays to read the data.

Using Data Streams

Now that we've reviewed the types of streams and their properties, it's time to look at some stream code.

Some of the techniques for accessing streams may seem a bit new to you, but as you will see they are very similar among different stream classes.

FileInputStream and FileOutputStream

The first stream classes that we are going to discuss in detail are the most basic file stream classes, `FileInputStream` and `FileOutputStream`. They are used to read bytes from a file or write bytes to a file, respectively. These classes include constructors that take a `File` object or `String`, representing a path to the file.

The data in a `FileInputStream` object is commonly accessed by successive calls to the `read()` method until a value of `-1` is returned, indicating that the end of the stream (end of file) has been reached.

Of course, you can also choose to stop reading the stream early just by exiting the loop, such as if some search String is found.

The `FileInputStream` class also contains overloaded versions of the `read()` method, which take a pointer to a byte array where the data is written. The method returns an integer value indicating how many bytes can be read into the byte array. It is also used by `Buffered` classes to improve performance, as you shall see in the next section.

A `FileOutputStream` object is accessed by writing successive bytes using the `write(int)` method. The `FileOutputStream` also contains overloaded versions of the `write()` method that allow a byte array to be passed and can be used by `Buffered` classes.

The following code uses `FileInputStream` and `FileOutputStream` to copy a file:

```
import java.io.*;
public class CopyFileSample {
    public static void copy(File source, File destination)
        throws IOException {
        try (InputStream in = new FileInputStream(source);
             OutputStream out = new FileOutputStream(destination)) {
            int b;
            while((b = in.read()) != -1) {
                out.write(b);
            }
        }
    }
    public static void main(String[] args) throws IOException {
        File source = new File(args[0]);
        File destination = new File(args[1]);
        copy(source, destination);
    }
}
```

The main() method creates two File objects, one for the source file to copy from and one for the destination file to copy to. If the destination file already exists, it will be overwritten by this code.

The file names and paths are input as strings by the user and if the input file doesn't exist at the path indicated, a FileNotFoundException will be thrown.

The copy() method creates instances of FileInputStream and FileOutputStream, and it proceeds to read the FileInputStream one byte at a time, copying the value to the FileOutputStream as it's read. As soon as the in.read() returns a -1 value, the loop ends. Finally, both streams are closed using the try-with-resource syntax shown earlier.

This code sample is inefficient as it only reads a byte at a time. This would be a problem for large files and so a buffered reader would be preferred as it buffers to a byte array.

BufferedInputStream and BufferedOutputStream

We can enhance our implementation with only a few minor code changes by wrapping the FileInputStream and FileOutputStream classes that you saw in the previous example with the BufferedInputStream and BufferedOutputStream classes, respectively.

Instead of reading the data one byte at a time, we use the underlying read(byte[]) method of BufferedInputStream, which returns the number of bytes read into the provided byte array. The number of bytes read is important for two reasons. First, if the value returned is 0, then we know that we have reached the end of the file and can stop reading from the BufferedInputStream. Second, the last read of the file will likely only partially fill the byte array, since it is unlikely for the file size to be an exact multiple of our buffer array size.

For example, if the buffer size is 1,024 bytes and the file size is 1,054 bytes, then the last read will be only 30 bytes. The length value tells us how many of the bytes in the array were actually read from the file. The remaining bytes of the array will be filled with leftover data from the previous read that should be discarded.

The data is written into the BufferedOutputStream using the write(byte[],int,int) method, which takes as input a byte array, an offset, and a length value, respectively. The offset value is the number of values to skip before writing characters, and it is often set to 0. The length value is the number of characters from the byte array to write.

Here's a modified form of the copy() method, which uses byte arrays and the Buffered stream classes:

```
public static void copy(File source, File destination) throws
    IOException {
    try {
        InputStream in =
            new BufferedInputStream(new FileInputStream(source));
        OutputStream out = new BufferedOutputStream(
            new FileOutputStream(destination));
        byte[] buffer = new byte[1024];
        int lengthRead;
        while ((lengthRead = in.read(buffer)) > 0) {
            out.write(buffer, 0, lengthRead);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
        out.flush();  
    }  
}
```

This code that uses byte arrays is very much like the earlier non-buffered code. However, the performance improvement for using both the Buffered classes and byte arrays is an order of magnitude faster in practice.

We also added a `flush()` command in the loop, as previously discussed, to ensure that the written data actually makes it to disk before the next buffer of data is read.

FileReader and FileWriter

The `FileReader` and `FileWriter` classes, along with their associated buffer classes, are among the most convenient classes in the `java.io` API, in part because reading and writing text data are among the most common ways that developers interact with files.

Like the `FileInputStream` and `FileOutputStream` classes, the `FileReader` and `FileWriter` classes contain `read()` and `write()` methods, respectively.

These methods read/write char values instead of byte values; although similar to what you saw with streams, the API actually uses an int value to hold the data so that -1 can be returned if the end of the file is detected. The FileReader and FileWriter classes contain other methods that you saw in the stream classes, including close() and flush(), the usage of which is the same.

The Writer class, which FileWriter inherits from, offers a write(String) method that allows a String object to be written directly to the stream. Using FileReader also allows you to pair it with BufferedReader in order to use the very convenient readLine() method, which you will see in the next example.

BufferedReader and BufferedWriter

Let's take a look at a sample program that makes use of both the `BufferedReader` and `BufferedWriter` classes using the associated `readLine()` and `write(String)` methods. It reads a text file, outputs each line to screen, and writes a copy of the file to disk. Since these classes are buffered, you can expect better performance than if you read/wrote each character one at a time.

```
import java.io.*;
import java.util.*;
public class CopyTextFileSample {
    public static List<String> readFile(File source) throws IOException {
        List<String> data = new ArrayList<String>();
        try (BufferedReader reader = new BufferedReader(new
                FileReader(source))) {
            String s;
            while((s = reader.readLine()) != null) {
                data.add(s);
            }
        }
    }
}
```

```

        }
        return data;
    }
    public static void writefile(List<String> data, File destination)
                                throws IOException {
        try (BufferedWriter writer = new BufferedWriter(
                new FileWriter(destination))) {
            for(String s: data) {
                writer.write(s);
                writer.newLine();
            }
        }
    }
    public static void main(String[] args) throws IOException {
        File source = new File(args[0]);
        File destination = new File(args[1]);
        List<String> data = readfile(source);
        for(String record: data) {
            System.out.println(record);
        }
        writefile(data, destination);
    }
}

```

Although both this example and the previous InputStream/OutputStream solution can successfully copy the file, only the Reader/Writer solution gives us structured access to the text data.

In order to accomplish the same feat with the InputStream/OutputStream classes, the application would have to detect the end of each line, which could be a lot of extra work.

For example, if we are using a BufferedInputStream, multiple end-of-line characters could appear in the buffer array, meaning that we would have to go searching for them and then reconstruct the strings contained within the buffer array manually.

We would also have to write code to detect and process the character encoding. The character encoding determines how characters are encoded and stored in bytes and later read back or decoded as characters.

Although this may sound simple, Java supports a wide variety of character encodings, ranging from ones that may use one byte for Latin characters, UTF-8 and ASCII for example, to using two or more bytes per character, such as UTF-16.

PrintStream and PrintWriter

The PrintStream and PrintWriter classes are high-level stream classes that write formatted representation of Java objects to a text-based output stream.

As you may have worked out from the name, the PrintStream class operates on OutputStream instances and writes data as bytes, whereas the PrintWriter class operates on Writer instances and writes data as characters.

For convenience, both of these classes include constructors that can open and write

to files directly. Furthermore, the `PrintWriter` class even has a constructor that takes an `OutputStream` as input, allowing you to wrap a `PrintWriter` class around an `OutputStream`.

These classes are primarily convenience classes in that you could write the low-level primitive or object directly to a stream without a `PrintStream` or `PrintWriter` class, although using one is helpful in a wide variety of situations.

In fact, the primary method class we have been using to output information to screen throughout this book uses a `PrintStream` object; `System.out` and `System.err` are all `PrintStream` objects.

Because `PrintStream` inherits `OutputStream` and `PrintWriter` inherits from `Writer`, both support the underlying `write()` method while providing a slew of print-based methods. For the exam, you should be familiar with the `print()`, `println()`, `format()`, and `printf()` methods.

Unlike the underlying `write()` method, which throws a checked `IOException` that must be caught in your application, these print-based methods do not throw any checked exceptions. If they did, you would have been required to catch a checked exception anytime you called `System.out.println()` in your code. Both classes provide a method, `checkError()`, that can be used to detect the presence of a problem after attempting to write data to the stream.

For the rest of this section, we will use `PrintWriter` in our examples, as writing String data as characters instead of byte values is recommended. Keep in mind that the same examples could be easily rewritten with a `PrintStream` object.

print()

The most basic of the print-based methods is `print()`, which is overloaded with all Java primitives as well as `String` and `Object`. In general, these methods perform `String.valueOf()` on the argument and call the underlying stream's `write()` method, although they also handle character encoding automatically.

println()

The next methods available in the `PrintStream` and `PrintWriter` classes are the `println()` methods, which are virtually identical to the `print()` methods, except that they insert a line break after the `String` value is written. The classes also include a version of `println()` that takes no arguments, which terminates the current line by writing a line separator.

format() and printf()

Like the `String.format()` methods discussed in Chapter 5, the `format()` method in `PrintStream` and `PrintWriter` takes a `String`, an optional locale, and a set of arguments, and it writes a formatted `String` to the stream based on the input. In other words, it is a convenience method for formatting directly to the stream.

ObjectInputStream and ObjectOutputStream

Throughout this course, we have been managing our data model using classes, so it makes sense that we would want to write these objects to a stream (e.g. a file). The

process of converting an in-memory object to a stored data format is referred to as serialisation, with the reciprocal process of converting stored data into an object, which is known as de-serialisation. In this section, we will show you how Java provides built-in mechanisms for serialising and deserialising streams of objects directly to and from streams, respectively.

The Serializable Interface

In order to serialise objects using the `java.io` API, the class they belong to must implement the `java.io.Serializable` interface.

The `Serializable` interface is a tagging or marker interface, which means that it does not have any methods associated with it. Any class can implement the `Serializable` interface since there are no required methods to implement.

The purpose of implementing the `Serializable` interface is to inform any process attempting to serialise the object that you have taken the proper steps to make the object serialisable, which involves making sure that the classes of all instance variables within the object are also marked `Serializable`.

Many of the built-in Java classes that you have worked with throughout this book, including the `String` class, are marked `Serializable`. This means that many of the simple classes that you have built throughout this course can be marked `Serializable` without any additional work.

A process attempting to serialise an object will throw a `NotSerializableException` if the class or one of its contained classes does not properly implement the `Serializable` interface.

Serialising and Deserialising Objects

The `java.io` API provides two stream classes for object serialisation and de-serialisation called `ObjectInputStream` and `ObjectOutputStream`.

The `ObjectOutputStream` class includes a method to serialise the object to the stream called `void writeObject(Object)`. If the provided object is not `Serializable`, or it contains an embedded reference to a class that is not `Serializable` or not marked `transient`, a `NotSerializableException` will be thrown at runtime.

For the reciprocal process, the `ObjectInputStream` class includes a de-serialisation method that returns an object called `readObject()`. Notice that the return type of this method is the generic type `java.lang.Object`, indicating that the object will have to be cast explicitly at runtime to be used.

Here is a sample program that reads and writes `Payment` data objects:

```
package finance;
import java.io.*;
import java.util.*;
public class SerialisePayments{
    public static List<Payment> getPayments(File dataFile)
```

```

        throws IOException, ClassNotFoundException {
List<Payment> payments = new ArrayList<Payment>();
try (ObjectInputStream in = new ObjectInputStream(
        new BufferedInputStream(
        new FileInputStream(dataFile)))) {
    while(true) {
        Object obj = in.readObject();
        if(obj instanceof Payment){
            payments.add((Payment)obj);
        }
    }
} catch (EOFException e) { // File end reached
    // No action taken
}
return payments;
}
public static void createPaymentsFile(List<Payment> payments,
        File dataFile) throws IOException {
try (ObjectOutputStream out = new ObjectOutputStream(
        new BufferedOutputStream(
        new FileOutputStream(dataFile)))) {
    for(Payment payment : payments){
        out.writeObject(payment);
    }
}
}
public static void main(String[] args) throws IOException,
        ClassNotFoundException {
List<Payment> payments = new ArrayList<Payment>();
payments.add(new Payment("Office Supplies", 672.00F));
payments.add(new Payment("Pippa's Pizza", 897.01F));
File dataFile = new File("payment.data");
createPaymentsFile(payments, dataFile);
System.out.println(getPayments(dataFile));
}
}
}

```

The Payment class looks like this:

```

package finance;
import java.io.Serializable;
public class Payment implements Serializable{
    private String name;
    private float amount;
    public Payment(String name, float amount){
        this.name = name;
        this.amount = amount;
    }
    @Override
    public String toString(){
        return "[" + name + ", " + amount + "]";
    }
}

```

When working with serialisation of objects there may be fields that you do not wish to include in the process. These can be marked with the modifier keyword transient.

Transient variables are defined in Java as: “not part of the persistent state of an object.” They could be variables that change frequently over time or are recalculated as part of some other process. There is no point in storing and retrieving them through serialisation and de-serialisation. Therefore they are not written to streams and when the object is recreated, transient variables are given their default value (zero, null, false etc.).

User IO

The `java.io` API includes numerous classes for interacting with the user. For example, you might want to write an application that asks a user to log in and reads their login details. In this section, we present the final `java.io` class of this course, the `java.io.Console` class, or `Console` class for short.

The `Console` class was introduced in Java 6 as a more evolved form of the `System.in` and `System.out` stream classes. It is now the recommended technique for interacting with and displaying information to the user in a text-based environment.

Pre-Console

Before we delve into the `Console` class, let's review the old way of obtaining text input from the user.

Similar to how `System.out` returns a `PrintStream` and is used to output text data to the user, `System.in` returns an `InputStream` and is used to retrieve text input from the user. It can be chained to a `BufferedReader` to allow input that terminates with the Enter key. Before we can apply the `BufferedReader`, though, we need to wrap the `System.in` object using the `InputStreamReader` class, which allows us to build a `Reader` object out of an existing `InputStream` instance. The result is shown in the following application:

```
import java.io.*;
public class SystemInSample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        String userInput = reader.readLine();
        System.out.println("You entered the following: " + userInput);
    }
}
```

This application fetches a single line of text from the user and then outputs it to the user before terminating.

Notice that we did not close the stream, as closing `System.in` would prevent our application from accepting user input for the remainder of the application execution.

Console

The `System.in` and `System.out` objects have been available since the earliest versions of Java.

In Java 6, the `java.io.Console` class was introduced with far more features and abilities than the original techniques. After all, `System.in` and `System.out` are just raw streams, whereas `Console` is a class with multiple convenience methods, one that is capable of containing additional methods in the future.

To begin, the `Console` class is a singleton, which as you may remember means that there is only one version of the object available in the JVM. It is created automatically for you by the JVM and accessed by calling the `System.console()` method.

Be aware that this method will return `null` in environments where text interactions are not supported such as embedded systems and some servers.

Next, let's look at our previous sample code rewritten using the `Console` class:

```
import java.io.Console;
public class ConsoleSample {
    public static void main(String[] args) {
        Console console = System.console();
        if(console != null) {
            String userinput = console.readLine();
            console.writer().println("You entered the following: "
                + userinput);
        }
    }
}
```

The sample code first retrieves an instance of the `Console` singleton and determines if the `Console` is available by checking it for a `null` value.

If the `Console` is available, it then retrieves a line of input from the user using the `readLine()` method, and it outputs the result using the `Console`'s built-in `PrintWriter` object, accessed via the `writer()` method.

As you can see, the sample with `System.in` and `System.out` is very similar to the `Console` example by design. We will now review the various methods available in the `Console` class.

reader() and writer()

The `Console` class provides access to an instance of `Reader` and `PrintWriter` using the methods `reader()` and `writer()`, respectively.

Access to these classes is analogous to calling `System.in` and `System.out` directly, although they use the `Reader/Writer` classes instead of the `InputStream/OutputStream` classes, which are more appropriate for working with character and `String` data. In this manner, they handle the underlying character encoding automatically.

These `reader()` and `writer()` methods are the most general ones in the `Console` class, and they are used by developers who need raw access to the user input and output stream or who may be in the process of migrating away from `System.in`.

format() and printf()

For outputting data to the user, you can use the `PrintWriter` `writer()` object or use the convenience `format(String, Object...)` method directly.

The `format()` method takes a `String` format and list of arguments, and it behaves in the same manner as `String.format()` described earlier in the course. For convenience to C

developers, there is also a `printf()` method in the `Console` class, which is identical in every way but name to the `format()` method, and it can be used in any place `format()` is used.

Note that the `Console` class defines only one `format()` method, and it does not define a `format()` method that takes a locale variable. In this manner, it uses the default system locale to establish the formatter. Of course, you could always use a custom locale by retrieving the `Writer` object and passing your own locale instance.

flush()

The `flush()` method forces any buffered output to be written immediately.

It is recommended that you call the `flush()` method prior to calling any `readLine()` or `readPassword()` methods in order to ensure that no data is pending during the read. Failure to do so could result in a user prompt for input with no preceding text, as the text prior to the prompt may still be in a buffer.

readLine()

The basic `readLine()` method retrieves a single line of text from the user, and the user presses the Enter key to terminate it.

The `Console` class also supports an overloaded version of the `readLine()` method with the following signature:

```
readLine(String format, Object... args)
```

This displays a formatted prompt to the user prior to accepting text.

readPassword()

The `readPassword()` method is similar to the `readLine()` method, except that echoing is disabled. By disabling echoing, the user does not see the text they are typing, meaning that their password is secure if someone happens to be looking at their screen.

Also like the `readLine()` method, the `Console` class offers an overloaded version of the `readPassword()` method with the following signature:

```
readPassword(String format, Object... args)
```

This is used for displaying a formatted prompt to the user prior to accepting text. Unlike the `readLine()` method, though, the `readPassword()` method returns an array of characters instead of a `String`.

Summary

The bulk of this chapter focused on teaching you how to use Java to interact with files.

First you were introduced to the concept of files and directories, and then how to reference them using path Strings.

You saw the `java.io.File` class and how to use it to read basic file information.

Next we looked at `java.io` streams and how to read and write file contents, and we described their various attributes, including low-level vs. high-level, byte vs. character, input vs. output, and so on. The description of the stream is designed to help you remember the function of the stream by using its name as a context clue.

We visited many of the byte and character stream classes that you will need to know for the exam in increasing order of complexity. A common practice is to start with a low-level resource or file stream and wrap it in a buffered stream to improve performance. You can also apply a high-level stream to manipulate the data, such as a data stream.

We described what it means to be serialisable in Java, and we showed you how to use the object stream classes to persist objects directly to and from disk.

We concluded the chapter by showing you how to read input data from the user, using both the legacy `System.in` method and the newer `Console` class. The `Console` class has many advanced features, such as support for passwords and built-in support for String formatting.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

C H A P T E R 9

NIO.2

Introduction

In the previous chapter we presented the `java.io` API and discussed how to use it to interact with files. In this chapter, we will be focussing on the `java.nio` version 2 API, or NIO.2 for short.

NIO.2 is an acronym that stands for the second version of the Non-blocking Input/Output API, and it is sometimes referred to as the “New I/O.”

In this chapter, you will see how the NIO.2 API enables us to do a lot more with files and directories than the original `java.io` API such as reading and modifying the attributes of a file.

We will conclude the chapter by introducing new NIO.2 methods that were added in Java 8, which rely on streams to perform complex operations with only a single line of code.

Java file I/O has undergone a number of revisions over the years.

The first version of fileI/O available in Java was the `java.io` API. As covered in the previous chapter , the `java.io` API uses byte streams to interact with file data. Although the `java.io` API has grown over the years, the underlying byte stream concept has not changed significantly since it was introduced.

Java introduced a replacement for `java.io` streams in Java 1.4 called Non-blocking I/O, or NIO for short. The NIO API introduced the concepts of buffers and channels in place of `java.io` streams. The basic idea is that you load the data from a file channel into a temporary buffer that, unlike byte streams, can be read forward and backward without blocking on the underlying resource.

Java 7 introduced the NIO.2 API. While this was intended to replace `java.io` streams, it is actually a replacement for the `java.io.File` class

People sometimes refer to NIO.2 as just NIO, although for clarity and to distinguish it from the first version of NIO, it will be referred to as NIO.2 throughout this chapter.

NIO.2 is supposed to provide a more intuitive, more feature-rich API for working with files. It also provides a number of performance improvements over the existing `java.io.File` class.

Path Interface

The `java.nio.file.Path` interface, or `Path` for short, is the primary entry point for working with the NIO.2 API.

A `Path` object represents a hierarchical path on the storage system to a file or directory. In this manner, `Path` is a direct replacement for the legacy `java.io.File` class, and conceptually it contains many of the same properties. For example, both `File` and `Path` objects may refer to a file or a directory.

Both also may refer to an absolute path or relative path within the file system. As we did in Chapter 8 and continue to do in this chapter, for simplicity's sake, we often refer to a directory reference as a file record since it is stored in the file system with similar properties.

Unlike the `File` class, the `Path` interface contains support for symbolic links. A symbolic link is a special file within an operating system that serves as a reference or pointer to another file or directory.

In general, symbolic links are transparent to the user, as the operating system takes care of resolving the reference to the actual file. NIO.2 includes full support for creating, detecting, and navigating symbolic links within the file system.

Path Factory and Helper Classes

NIO.2 makes good use of the factory pattern as discussed earlier in the course.

Remember that a factory class is usually implemented using static methods to create instances of another class (implementers of `Path` in this case).

For example, you can create an instance of a `Path` interface using a static method available in the `Paths` factory class. Note the 's' at the end of the `Paths` class to distinguish it from the `Path` interface.

NIO.2 also includes helper classes such as `java.nio.file.Files`, whose primary purpose is to operate on instances of `Path` objects.

Helper or utility classes are similar to factory classes in that they are often composed primarily of static methods that operate on a particular class. They differ in that helper classes are focused on manipulating or creating new objects from existing instances, whereas factory classes are focused primarily on object creation.

You should become comfortable with this paradigm, if you are not already, as most interactions with NIO.2 will require at least two classes: an interface and a factory or helper class.

Here is a diagram indicating the relationship between the NIO.2 classes and interfaces:

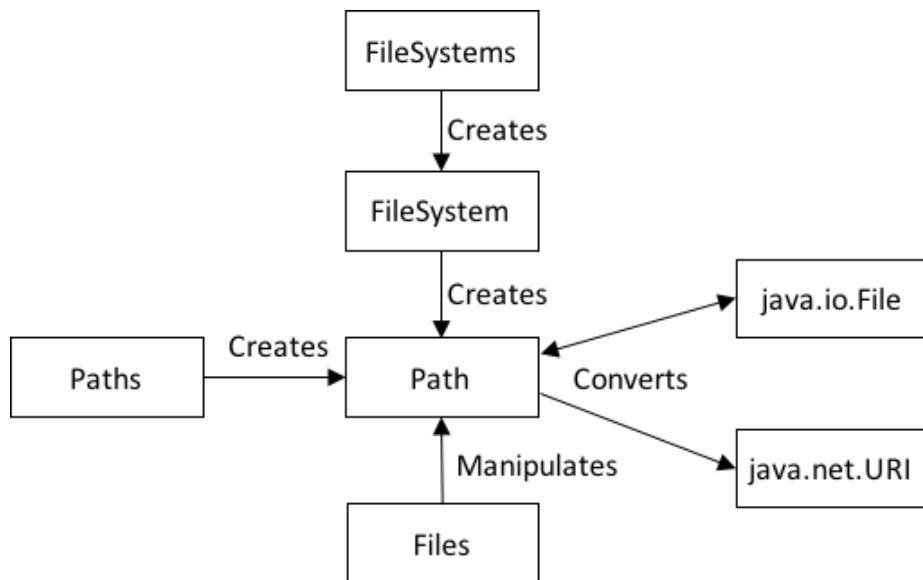


Figure 9-1 NIO.2 Classes and Interfaces

Creating Path Objects

Since Path is an interface, you need a factory class to create instances of one. NIO.2 provides a number of classes and methods that you can use to create Path objects, which you will see in this section.

Using Paths

The simplest and most straightforward way to obtain a Path object is using the `java.nio.files.Paths` factory class, or `Paths` for short. To obtain a reference to a file or directory, use the static method `Paths.getPath(String)` method:

```

Path path1 = Paths.get("demo/picture.jpg");
Path path2 = Paths.get("c:\\home\\Nov\\schedule.csv");
Path path3 = Paths.get("/home/Nov");
  
```

The first example creates a Path reference to a relative file in the current working directory.

The second example creates a Path reference to an absolute file in a Windows-based system.

The third example creates a Path reference to an absolute directory in a Linux or Mac system.

You can also create a Path using the `Paths` class using a vararg of type `String`:

```
Paths.get(String, String...)
```

This creates a Path from a list of `String` values in which the operating system-dependent `path.separator` is automatically inserted between elements.

Another way to construct a Path using the `Paths` class is with a `URI` value.

A uniform resource identifier (URI) is a string of characters that identify a resource. A URL is a subtype of URI which also includes URN (uniform resource name).

It begins with a schema that indicates the resource type, followed by a path value. Examples of schema values include file://, http://, https://, and ftp://.

The java.net.URI class is used to create and manage URI values.

This example accesses the local file system using a URI:

```
Path path = Paths.get(new URI("file:///c:/home/Nov/schedule.csv"));
```

FileSystem Object

The Path.getPath() method used throughout the previous examples is actually shorthand for the class java.nio.file.FileSystem method getPath().

The FileSystem class has a protected constructor, so we use the plural FileSystems factory class to obtain an instance of FileSystem, as shown in the following example code:

```
Path path1 = FileSystems.getDefault().getPath("demo/Text.txt");
Path path2 = FileSystems.getDefault()
    .getPath("c:", "home", "Nov", "schedule.csv");
Path path3 = FileSystems.getDefault().getPath("/home/Nov");
```

This is a rewrite of the previous examples, with the same results.

While most of the time we want access to a Path object that is within the local file system, the FileSystems factory class does give us the ability to connect to a remote file system, as shown in the following sample code:

```
FileSystem fileSystem =
    FileSystems.getFileSystem(new URI("http://www.stayahead.com"));
Path path = fileSystem.getPath("Test.txt");
```

This code is useful when we need to construct Path objects frequently for a remote file system. The power of NIO.2 here is that it lets us rely on the default file system for files and directories as before, while giving us the ability to build more-complex applications that reference external file systems.

Converting File objects

When Path was added in Java 7, the legacy java.io.File class was updated with a new method, toPath(), that operates on an instance File variable:

```
File file = new File("demo/Test.txt");
Path path = file.toPath();
```

For backward compatibility, the Path interface also contains a method toFile() to return a File instance:

```
Path path = Paths.get("File.txt");
File file = path.toFile();
```

As you can see, the Java API is quite flexible, and it allows easy conversion between legacy code using the File class and newer code using Path.

Although Java supports both methods for working with files, it is generally recommended that you rely on the Path API in your applications going forward as it is more feature rich and has built-in support for various file systems and symbolic links.

Paths and Files Classes

Now you know how to obtain an instance of the Path object let's look at what you can do with it.

NIO.2 provides a variety of methods and classes that operate on Path objects. This section covers the key methods that you should know to use NIO.2 effectively.

Path Objects

The Path interface includes numerous methods for using Path objects. You have already seen two of them, toFile() and toUri(), used to convert Path objects to other types of resources.

Many of the methods in the Path interface transform the path value in some way and return a new Path object, allowing the methods to be chained. We demonstrate chaining in the following example, the details of which we'll discuss in this section of the chapter.

```
Paths.get("/home/.. /demo").getParent().normalize().toAbsolutePath();
```

toString(), getNameCount(), and getName()

Path contains three methods to retrieve basic information about the path representative.

The first method, `toString()`, returns a String representation of the entire path. In fact, it is the only method in the Path interface to return a String. Most of the other methods that we will discuss in this section return a new Path object.

The second and third methods, `getNameCount()` and `getName(int)`, are often used in conjunction to retrieve the number of elements in the path and a reference to each element, respectively.

For greater compatibility with other NIO.2 methods, the `getName(int)` method returns the component of the Path as a new Path object rather than a String.

The following sample code uses these methods to retrieve path data:

```
Path path = Paths.get("/demo/code/List.txt");
System.out.println("The Path Name is: " + path);
for(int i=0; i<path.getNameCount(); i++) {
    System.out.println(" Element " + i + " is: " + path.getName(i));
}
```

As you might remember from our discussion of PrintStream/PrintWriter in Chapter 8, printing an object automatically invokes the object's `toString()` method. The output of this code snippet is the following:

```
The Path Name is: /demo/code/List.txt
Element 0 is: demo
Element 1 is: code
Element 2 is: List.txt
```

Notice that the root element / is not included in the list of names. If the Path object represents the root element itself, then the number of names in the Path object returned by getNameCount() will be 0.

What if we ran the preceding code using the relative path demo/code/List.txt?

The output would be as follows:

```
The Path Name is: demo/code/List.txt
Element 0 is: demo
Element 1 is: code
Element 2 is: List.txt
```

Notice that the individual names are the same as before. The getName(int) method is zero-indexed, with the file system root excluded from the path components.

getFileName(), getParent(), and getRoot()

The Path interface contains numerous methods for retrieving specific sub elements of a Path object, returned as Path objects themselves.

The first method, getFileName(), returns a Path instance representing the filename, which is the farthest element from the root. Like most methods in the Path interface, getFileName() returns a new Path instance rather than a String.

The next method, getParent(), returns a Path instance representing the parent path or null if there is no such parent. If the instance of the Path object is relative, this method will stop at the top-level element defined in the Path object. In other words, it will not traverse outside the working directory to the file system root.

The last method, getRoot(), returns the root element for the Path object or null if the Path object is relative.

Here is some code that traverses absolute and relative Path objects to show how each handles the root differently:

```
package demo;
import java.nio.file.*;
public class PathFilepathTest {
    public static void printPathInformation(Path path) {
        System.out.println("Filename is: " + path.getFileName());
        System.out.println("Root is: " + path.getRoot());
        Path currentParent = path;
        while((currentParent = currentParent.getParent()) != null) {
            System.out.println(" Current parent is: " + currentParent);
        }
    }
    public static void main(String[] args) {
        for(String fileName : args){
            printPathInformation(Paths.get(fileName));
            System.out.println();
        }
    }
}
```

The while loop in the `printPathInformation()` method continues until `getParent()` returns null.

When run with the following input:

```
j ava demo.PathF i l ePathTest /Users/Admi n/Documents/Work
```

It produces the following output:

```
Fi lename i s: Work
Root i s: /
Current parent i s: /Users/Admi n/Documents
Current parent i s: /Users/Admi n
Current parent i s: /Users
Current parent i s: /
```

isAbsolute() and toAbsolutePath()

The Path interface contains two methods for assisting with relative and absolute paths.

The first method, `isAbsolute()`, returns true if the path the object references is absolute and false if the path the object references is relative. As discussed earlier in this chapter, whether a path is absolute or relative is often file system dependent.

The second method, `toAbsolutePath()`, converts a relative Path object to an absolute Path object by joining it to the current working directory. If the Path object is already absolute, then the method just returns a copy of it.

The following code snippet shows usage of both of these methods:

```
Path path1 = Paths.get("C:\\\\Work\\\\schedul e.csv");
System.out.println("Path1 i s Absol ute? " + path1. i sAbsol ute());
System.out.println("Absol ute Path1: " + path1. toAbsol utePath());
Path path2 = Paths.get("Work/notes. txt");
System.out.println("Path2 i s Absol ute? " + path2. i sAbsol ute());
System.out.println("Absol ute Path2 " + path2. toAbsol utePath());
```

The output for the code snippet is shown below. Since the precise output is file system dependent, assume the first example is being run on a Windows- based system, whereas the second example is run in a Linux or Mac system with the current working directory of /home.

```
Path1 i s Absol ute? true
Absol ute Path1: C:\\Work\\schedul e.csv
Path2 i s Absol ute? fal se
Absol ute Path2 /home/Work/notes. txt
```

Keep in mind that if the Path object already represents an absolute path, then the output is a new Path object with the same value.

subpath()

The method `subpath(int, int)` returns a relative subpath of the `Path` object, referenced by an inclusive start index and an exclusive end index. It is useful for constructing a new relative path from a particular parent path element to another parent path element, as shown in the following example:

```
Path path = Paths.get("/home/Work/schedule.csv");
System.out.println("Path is: " + path);
System.out.println("Subpath from 0 to 3 is: " + path.subpath(0, 3));
System.out.println("Subpath from 1 to 3 is: " + path.subpath(1, 3));
System.out.println("Subpath from 1 to 2 is: " + path.subpath(1, 2));
```

You might notice that the `subpath()` and `getName(int)` methods are similar in that they both return a `Path` object that represents a component of an existing `Path`.

The difference is that the `subpath()` method may include multiple path components, whereas the `getName(int)` method only includes one.

The output of this code snippet is the following:

```
Path is: /home/Work/schedule.csv
Subpath from 0 to 3 is: home/Work/schedule.csv
Subpath from 1 to 3 is: Work/schedule.csv
Subpath from 1 to 2 is: Work
```

This code demonstrates that the `subpath(int, int)` method does not include the root of the file. Notice that the 0-indexed element is `home` in this example and not the root directory; therefore, the maximum index that can be used is 3.

The following two examples both throw `java.lang.IllegalArgumentException` at runtime:

```
System.out.println("Subpath from 0 to 4 is: " + path.subpath(0, 4));
System.out.println("Subpath from 1 to 1 is: " + path.subpath(1, 1));
```

The first one uses an invalid index (4) and the second uses the same index twice resulting in an invalid empty path.

Two symbols are often used with paths:

- `..` = parent directory
- `.` = current directory

relativize()

The `Path` interface provides a method `relativize(Path)` for constructing the relative path from one `Path` object to another.

Consider the following relative and absolute path examples using the `relativize()` method.

```
Path path1 = Paths.get("Test.txt");
Path path2 = Paths.get("notes.txt");
System.out.println(path1.relativize(path2));
System.out.println(path2.relativize(path1));
```

The code snippet produces the following output when executed:

```
..\\Test.txt ..\\notes.txt
```

If both path values are relative, then the `relativize()` method computes the paths as if they are in the same current working directory. Notice that `..\\` is included at the start of the first set of examples. Since our path value points to a file, we need to move to the parent directory that contains the file.

If both path values are absolute however, then the method computes the relative path from one absolute location to another, regardless of the current working directory.

The following example demonstrates this:

```
Path path3 = Paths.get("E:\\\\Work");
Path path4 = Paths.get("E:\\\\Work\\\\Nov");
System.out.println(path3.relativize(path4));
System.out.println(path4.relativize(path3));
```

This code snippet produces the following output when executed:

```
..\\Work\\Nov
...\\..\\Work
```

resolve()

The Path interface includes a `resolve(Path)` method for creating a new Path by joining an existing path to the current path.

In other words, the object on which the `resolve()` method is invoked becomes the basis of the new Path object, with the input argument being appended onto the Path. Let's see what happens if we apply `resolve()` to an absolute path and a relative path:

```
final Path path1 = Paths.get("/Work/..\\temp");
final Path path2 = Paths.get("j unk");
System.out.println(path1.resolve(path2));
```

The code snippet generates the following output:

```
/Work/..\\temp/j unk
```

Like the `relativize()` method, the `resolve()` method does not clean up path symbols, such as the parent directory `..` symbol. For that, you'll need to use the `normalize()` method (see below).

In this example, the input argument to the `resolve()` method was a relative path, but what if had been an absolute path?

```
final Path path1 = Paths.get("/Users/Admin");
final Path path2 = Paths.get("/Applications/Java");
System.out.println(path1.resolve(path2));
```

Since the input parameter `path2` is an absolute path, the output would be:

```
/Applications/Java
```

If an absolute path is provided as input to the method, such as `path1.resolve(path2)`, then `path1` would be ignored and a copy of `path2` would be returned as it doesn't make much sense to join them together.

normalize()

As you saw with the `relativize()` method, file systems can construct relative paths using `..` and `.` values. There are times, however, when relative paths are combined such that there are redundancies in the path value.

Java provides the `normalize(Path)` method to eliminate the redundancies in the path.

For example, let's take some output that results in the path value `..\user\home` and try to reconstitute the original absolute path using the `resolve()` method:

```
Path path3 = Paths.get("E:\\\\data");
Path path4 = Paths.get("E:\\\\user\\\\home");
Path relativePath = path3.relativize(path4);
System.out.println(path3.resolve(relativePath));
```

The result of this sample code would be the following output:

```
E:\\data\\..\\user\\home
```

You can see that this path value contains a redundancy and it does not match our original value, `E:\\user\\home`. We can resolve this redundancy by applying the `normalize()` method as shown here:

```
System.out.println(path3.resolve(relativePath).normalize());
```

This modified last line of code nicely produces our original path value:

```
E:\\user\\home
```

Like `relativize()`, the `normalize()` method does not check whether the file exists.

toRealPath()

The `toRealPath(Path)` method takes a `Path` object that may or may not point to an existing file within the file system, and it returns a reference to a real path within the file system.

It is similar to the `toAbsolutePath()` method in that it can convert a relative path to an absolute path, except that it also verifies that the file referenced by the path exists, and thus it throws a checked `IOException` at runtime if the file cannot be located. It also supports the `NOFOLLOW_LINKS` option.

The `toRealPath()` method performs additional steps, such as removing redundant path elements. In other words, it implicitly calls `normalize()` on the resulting absolute path.

Let's say that we have a file system in which we have a symbolic link from `Tasks.source` to `Tasks.txt`, as described in the following relationship:

`/Work/Task.source → /Users/Admin/Tasks.txt`

Assuming that our current working directory is /Users/Guest, then consider the following code:

```
try {
    System.out.println(Paths.get("/Work/Task.source").toRealPath());
    System.out.println(Paths.get(".././Tasks.txt").toRealPath());
} catch (IOException e) {
    // Handle file I/O exception...
}
```

Notice that we need to catch (or declare) IOException, since unlike the toAbsolutePath() method, the toRealPath() method interacts with the file system to check if the path is valid.

Given the symbolic link and current working directory as described, then the output would be the following:

```
/Users/Admin/Tasks.txt  
/Users/Admin/Tasks.txt
```

In these examples, the absolute and relative paths both resolve to the same absolute file, as the symbolic link points to a real file within the file system.

You can also use the toRealPath() method to gain access to the current working directory, such as shown here:

```
System.out.println(Paths.get(".").toRealPath());
```

Files Class

You have seen how to get access to a Path object, and we can find plenty of information about it, but what can we do with the file it references?

Many of the same operations available in `java.io.File` are available to `java.nio.file.Path` via a helper class called `java.nio.file.Files`, or `Files` for short. Unlike the methods in the `Path` and `Paths` class, most of the options within the `Files` class will throw an exception if the file to which the `Path` refers does not exist.

`exists()`

The `Files.exists(Path)` method takes a `Path` object and returns true if, and only if, it references a file that exists in the file system.

Here's some sample code:

```
Files.exists(Paths.get("/Work/Logo.jpeg"));
Files.exists(Paths.get("/Work"));
```

The first example checks whether a file exists, while the second example checks whether a directory exists. Of course, this method could not throw an exception if the file does not exist, as doing so would prevent this method from ever returning false at runtime.

`isSameFile()`

The `Files.isSameFile(Path, Path)` method is useful for determining if two `Path` objects relate to the same file within the file system.

It takes two `Path` objects as input and follows symbolic links. Despite the name, the method also determines if two `Path` objects refer to the same directory.

The `isSameFile()` method first checks if the `Path` objects are equal in terms of `equal()`, and if so, it automatically returns true without checking to see if either file exists. If the `Path` object `equals()` comparison returns false, then it locates each file to which the path refers in the file system and determines if they are the same, throwing a checked `IOException` if either file does not exist.

`createDirectory()` and `createDirectories()`

To create directories in the legacy `java.io` API, we called `mkdir()` or `mkdirs()` on a `File` object.

In NIO.2, we can use the `Files.createDirectory(Path)` method to create a directory. There is also a plural form of the method called `createDirectories()`, which like `mkdirs()` creates the target directory along with any nonexistent parent directories leading up to the target directory in the path.

The directory-creation methods can throw the checked IOException, such as when the directory cannot be created or already exists. For example, `createDirectory()`, will throw an exception if the parent directory in which the new directory resides does not exist.

Both of these methods also accept an optional list of `FileAttribute<?>` values to set on the newly created directory or directories. We will discuss file attributes in the next section.

Here's a code snippet that shows how to create directories using NIO.2:

```
try {
    Files.createDirectory(Paths.get("/Temp/sort"));
    Files.createDirectories(Paths.get("/Temp/sort/alpha"));
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first example creates a new directory, `sort`, in the directory `/Temp`, assuming `/Temp` exists; or else an exception is thrown.

Contrast this with the second example that creates the directory `green` along with any of the following parent directories if they do not already exist, such as `/Temp`, `/Temp/sort`, or `/Temp/sort/alpha`.

copy()

Unlike the legacy `java.io.File` class, the NIO.2 `Files` class provides a set of overloaded `copy()` methods for copying files and directories within the file system.

The primary one you will use is `Files.copy(Path, Path)`, which copies a file or directory from one location to another. The `copy()` method throws the checked `IOException`, such as when the file or directory does not exist or cannot be read.

Directory copies are shallow rather than deep, meaning that files and subdirectories within the directory are not copied. To copy the contents of a directory, you would need to create a function to traverse the directory and copy each file and subdirectory individually:

```
try {
    Files.copy(Paths.get("/Work"), Paths.get("/Temp"));
    Files.copy(Paths.get("/Work/WorkList.txt"),
               Paths.get("/Temp/WorkList.txt"));
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first `copy()` performs a shallow copy of the `Work` directory, creating a new `Temp` directory, but it does not copy any of the contents of the original directory. The second `copy()` copies the `WorkList.txt` file from the directory `Work` to the directory `Temp`.

By default, copying files and directories will traverse symbolic links, although it will not overwrite a file or directory if it already exists, nor will it copy file attributes.

These behaviours can be altered by providing the additional options NOFOLLOW_LINKS, REPLACE_EXISTING, and COPY_ATTRIBUTES.

move()

The Files.move(Path,Path) method moves or renames a file or directory within the file system. Like the copy() method, the move() method also throws the checked IOException in the event that the file or directory could not be found or moved.

The following is some sample code that uses the move() method:

```
try {
    Files.move(Paths.get("c:\\Temp"), Paths.get("c:\\NewTemp"));
    Files.move(Paths.get("c:\\Users\\names.txt"),
               Paths.get("c:\\NewTemp\\usernames.txt"));
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first example renames the Temp directory to NewTemp, keeping all of the original contents from the source directory. The second example moves the names.txt file from the directory Users to the directory NewTemp, and it renames it to usernames.txt.

By default, the move() method will follow links, throw an exception if the file already exists, and not perform an atomic move.

These behaviours can be changed by providing the optional values NOFOLLOW_LINKS, REPLACE_EXISTING, or ATOMIC_MOVE, respectively, to the method. If the file system does not support atomic moves, an AtomicMoveNotSupportedException will be thrown at runtime.

delete() and deleteIfExists()

The Files.delete(Path) method deletes a file or empty directory within the file system.

The delete() method throws the checked IOException under a variety of circumstances. For example, if the path represents a non-empty directory, the operation will throw the runtime DirectoryNotEmptyException. If the target of the path is a symbol link, then the symbolic link will be deleted, not the target of the link.

The deleteIfExists(Path) method is identical to the delete(Path) method, except that it will not throw an exception if the file or directory does not exist, but instead it will return a boolean value of false. It will still throw an exception if the file or directory does exist but fails, such as in the case of the directory not being empty.

readAllLines()

The Files.readAllLines() method reads all of the lines of a text file and returns the results as an ordered List of String values.

NIO.2 includes an overloaded version that takes an optional Charset value.

The following sample code reads the lines of the file and outputs them to the user:

```
Path path = Paths.get("/Data/Data.txt");
try {
    final List<String> lines = Files.readAllLines(path);
    for(String line: lines) {
        System.out.println(line);
    }
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The code snippet reads all of the lines of the file and then iterates over them. As you might expect, the method may throw an IOException if the file cannot be read for any reason.

The entire file is read when `readAllLines()` is called, with the resulting String array storing all of the contents of the file in memory at once.

Therefore, if the file is large, you may encounter an `OutOfMemoryError` trying to load all of it into memory. Later on in the chapter, we will revisit this method and present a new stream-based NIO.2 method that is far more appropriate for handling large files.

File Attributes

In the previous section, we reviewed methods that could create, modify, read, or delete a file or directory.

The Files class also provides numerous methods for accessing file and directory metadata, referred to as file attributes. Metadata is data about data so file metadata is data about the file or directory record within the file system as distinct from the contents of the file.

A file or directory may be hidden or marked with a permission that prevents the current user from reading it. These are both file attributes and the Files class provides methods for determining this information from within your Java application.

The one thing to keep in mind while reading file metadata in Java is that some methods are operating system dependent. For example, some operating systems may not have a notion of user-level permissions, in which case users can read only files that they have permission to read.

Basic File Attributes

We begin the discussion of file attributes by presenting the basic methods, defined directly within the Files class, for reading file attributes.

These methods are usable within any file system although they may have limited meaning in some file systems. In the next section, we will present a more generalised approach using attribute views and show that they not only improve performance but also allow us to access file system-dependent attributes.

isDirectory(), isRegularFile(), and isSymbolicLink()

The Files class includes three methods for determining if a path refers to a directory, a regular file, or a symbolic link.

The methods to accomplish this are named Files.isDirectory(Path), Files.isRegularFile(Path), and Files.isSymbolicLink(Path), respectively.

Java defines a regular file as one that contains content, as opposed to a symbolic link, directory, resource, or other non-regular file that may be present in some operating systems. If the symbolic link points to a real file or directory, Java will perform the check on the target of the symbolic link. In other words, it is possible for isRegularFile() to return true for a symbolic link, as long as the link resolves to a regular file.

Let's take a look at some sample code:

```
Files.isDirectory(Paths.get("/Data/Project/Plan.xml"));
Files.isRegularFile(Paths.get("/Data/Projects.txt"));
Files.isSymbolicLink(Paths.get("/Data/Project"));
```

The first example returns true if Plan.xml is a directory or a symbolic link to a directory and false otherwise.

Note that directories can have extensions in most file systems, so it is possible for Plan.xml to be the name of a directory. The second example returns true if Projects.txt points to a regular file or alternatively a symbolic link that points to a regular file. The third example returns true if /Data/project is a symbolic link, regardless of whether the file or directory it points to exists.

None of these methods throws an exception if the file or directory does not exist.

isHidden()

The Files class includes the Files.isHidden(Path) method to determine whether a file or directory is hidden within the file system. In Linux or Mac systems, this is often denoted by file or directory entries that begin with a period character (.), while in Windows-based systems this requires the hidden attribute to be set.

The isHidden() method throws the checked IOException, as there may be an I/O error reading the underlying file information.

Here's some sample code:

```
try {
    System.out.println(Files.isHidden(Paths.get("/Data/Index.xml")));
} catch (IOException e) {
    // Handle file I/O exception...
}
```

If the Index.xml file is available and hidden within the file system, this method will return true.

isReadable() and isExecutable()

The Files class includes two methods for reading file accessibility: Files.isReadable(Path) and Files.isExecutable(Path).

This is important in file systems where the filename can be viewed within a directory, but the user may not have permission to read the contents of the file or execute it.

Here's some sample code:

```
System.out.println(Files.isReadable(Paths.get("/Data/Index.xml")));
System.out.println(Files.isExecutable(Paths.get("/Data/Index.xml")));
```

The first example returns true if the Index.xml file exists and its contents are readable, based on the permission rules of the underlying file system. The second example returns true if the Index.xml file is marked executable within the file system. Note that the file extension does not necessarily determine whether a file is executable. For example, an image file that ends in .xml could be marked executable within a Linux-based system.

Like the `isDirectory()`, `isRegularFile()`, and `isSymbolicLink()` methods, the `isReadable()` and `isExecutable()` methods do not throw exceptions if the file does not exist but instead return false.

size()

The `Files.size(Path)` method is used to determine the size of the file in bytes.

The size returned by this method represents the conceptual size of the data, and this may differ from the actual size on the persistence storage device due to file system compression and organisation. The `size()` method throws the checked `IOException` if the file does not exist or if the process is unable to read the file information.

The following is a sample call to the `size()` method:

Since NIO.2 was designed as a replacement for the `java.io` API, it includes many of the same methods in one form or another.

getLastModifiedTime() and setLastModifiedTime()

Most operating systems support tracking a last-modified date/time value with each file.

Some applications use this to determine when the file should be read again. For example, there might be a program that performs an operation anytime the file data changes. It is a lot faster to check the last modified attribute than to reload the entire contents of the file.

The `Files` class provides the method `Files.getLastModifiedTime(Path)`, which returns a `FileTime` object to accomplish this. The `FileTime` class is a simple container class that stores the date/time information about when a file was accessed, modified, or created. For convenience, it has a `toMillis()` method that returns the epoch time.

The `Files` class also provides a mechanism for updating the last-modified date/time of a file using the `Files.setLastModifiedTime(Path, FileTime)` method. The `FileTime` class also has a static `fromMillis()` method that converts from the epoch time to a `FileTime` object.

Both of these methods have the ability to throw a checked `IOException` when the file is accessed or modified.

Here's some sample code:

```
try {
    final Path path = Paths.get("/Data/Index.xml");
    System.out.println(Files.getLastModifiedTime(path).toMillis());
    Files.setLastModifiedTime(path,
        FileTime.fromMillis(System.currentTimeMillis()));
    System.out.println(Files.getLastModifiedTime(path).toMillis());
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first part of the code reads and outputs the last-modified time value of the `Index.xml` file. The next line sets a last-modified date/time using the current time value. Finally, it outputs the newly set last-modified date/time value.

getOwner() and setOwner()

Many UNIX/Linux style file systems also support the notion of user-owned files and directories.

The Files.getOwner(Path) method returns an instance of UserPrincipal that represents the owner of the file within the file system.

As you may have already guessed, there is also a method to set the owner, called Files.setOwner(Path,UserPrincipal).

Note that the operating system may intervene when you try to modify the owner of a file and block the operation. For example, a process running under one user may not be allowed to take ownership of a file owned by another user. Both the getOwner() and setOwner() methods can throw the checked exception IOException in case of any issues accessing or modifying the file.

In order to set a file owner to an arbitrary user, NIO.2 provides a UserPrincipalLookupService helper class for finding a UserPrincipal record for a particular user within a file system. In order to use the helper class, you first need to obtain an instance of a FileSystem object, either by using the FileSystems.getDefault() method or by calling getFileSystem() on the Path object with which you are working, as shown in the following two examples:

```
UserPrincipal owner = FileSystems.getDefault()
    .getUserPrincipalLookupService()
    .lookupPrincipalByName("Admin");

Path path = ...
UserPrincipal owner = path.getFileSystem()
    .getUserPrincipalLookupService()
    .lookupPrincipalByName("Admin");
```

Here are some examples of the getOwner() and setOwner() methods, including an example of how to use the UserPrincipalLookupService:

```
try {
    // Read owner of file
    Path path = Paths.get("/Data/Index.xml");
    System.out.println(Files.getOwner(path).getName());
    // Change owner of file
    UserPrincipal owner = path.getFileSystem()
        .getUserPrincipalLookupService()
        .lookupPrincipalByName("Admin");
    Files.setOwner(path, owner);
    // Output the updated owner information
    System.out.println(Files.getOwner(path).getName());
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first few lines read the owner of the file and outputs the name of the user.

The second set of lines retrieves a user named Admin within the related file system and uses it to set a new owner for the file. Finally, we read the file owner name again to verify that it has been updated.

File Attribute Views

Up until now, we have been accessing individual file attributes with single method calls. While this is functionally correct, there are often costs associated with accessing the file that make it far more efficient to retrieve all file metadata attributes in a single call.

Furthermore, some attributes are file system specific and cannot be easily generalised for all file systems.

NIO.2 addresses both of these concerns by allowing you to construct views for various file systems in a single method call.

A view is a group of related attributes for a particular file system type. A file may support multiple views, allowing you to retrieve and update various sets of information about it.

If you need to read multiple attributes of a file or directory at a time, the performance advantage of using a view may be substantial. Although more attributes are read than in a single method call, there are fewer round-trips between Java and the operating system, whereas reading the same attributes with the previously described single method calls would require many such trips.

In practice, the number of trips between Java and the operating system is more important in determining performance than the number of attributes read.

Introducing Views

To request a view, you need to provide both a path to the file or a directory whose information you want to read, as well as a class object, which tells the method which type of view you would like returned.

The Files API includes two sets of methods of analogous classes for accessing view information. The first method, `Files.readAttributes()`, returns a read-only view of the file attributes. The second method, `Files.getFileAttributeView()`, returns the underlying attribute view, and it provides a direct resource for modifying file information.

Both of these methods can throw a checked `IOException`, such as when the view class type is unsupported. For example, trying to read Windows-based attributes within a Linux file system may throw an `UnsupportedOperationException`.

This table lists the commonly used attributes and view classes:

Attributes Class	View Class	Description
BasicFileAttributes	BasicFileAttributeView	Basic set of attributes supported by all file systems
DosFileAttributes	DosFileAttributeView	Attributes supported by DOS/

		Windows-based systems
PosixFileAttributes	PosixFileAttributeView	Attributes supported by POSIX systems, such as UNIX, Linux, Mac etc.

The DOS and POSIX classes are useful for reading and modifying operating system specific properties where appropriate. They also both inherit from their respective attribute and view classes.

For example, PosixFileAttributes inherits from BasicFileAttributes, just as DosFileAttributeView inherits from BasicFileAttributeView, meaning that all of the operations available on the parent class are available in the respective subclasses.

Reading Attributes

NIO.2 provides a Files.readAttributes(Path,Class<A>) method, which returns read-only versions of a fileview. The second parameter uses generics such that the return type of the method will be an instance of the provided class.

BasicFileAttributes

All attributes classes extend from BasicFileAttributes; therefore it contains attributes common to all supported file systems. It includes many of the file attributes that you previously saw as single-line method calls in the Files class.

Here is some sample code that retrieves BasicFileAttributes on a file and outputs various metadata about the file:

```

import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
public class BasicFileAttributesSample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("/Projects/index.xml");
        BasicFileAttributes data = Files.readAttributes(path,
                BasicFileAttributes.class);
        System.out.println("Is path a directory? " + data.isDirectory());
        System.out.println("Is path a regular file? " +
                data.isRegularFile());
        System.out.println("Is path a symbolic link? " +
                data.isSymbolicLink());
        System.out.println("Path not a file, directory, nor symbolic link?
"
                + data.isOther());
        System.out.println("Size (in bytes): " + data.size());
        System.out.println("Creation date/time: " + data.creationTime());
        System.out.println("Last modified date/time: " +
                data.lastModifiedTime());
        System.out.println("Last accessed date/time: " +
                data.lastAccessTime());
    }
}

```

```

        System.out.println("Unique file identifier (if available): "
                           + data.fileKey());
    }
}

```

The majority of these attributes should be familiar to you from earlier in this chapter.

The only ones that are new are `isOther()`, `lastAccessTime()`, `creationTime()`, and `fileKey()`. The `isOther()` method is used to check for paths that are not files, directories, or symbolic links, such as paths that refer to resources or devices in some file systems. The `lastAccessTime()` and `creationTime()` methods return other date/time information about the file. The `fileKey()` method returns a file system value that represents a unique identifier for the file within the file system or null if it is not supported by the file system.

Modifying Attributes

While the `Files.readAttributes()` method is useful for reading file data, it does not provide a direct mechanism for modifying file attributes.

NIO.2 provides the `Files.getFileAttributeView(Path, Class<V>)` method, which returns a view object that we can use to update the file system dependent attributes. We can also use the view object to read the associated file system attributes by calling `readAttributes()` on the view object.

BasicFileAttributeView

`BasicFileAttributeView` is used to modify a file's set of date/time values. In general, we cannot modify the other basic attributes directly, since this would change the property of the file system object. For example, we cannot set a property to change a directory into a file, since this leaves the files in the future in an ambiguous state. Likewise, we cannot change the size of the object without modifying its contents.

Here is a sample application that reads a file's basic attributes and increments the file's last-modified date/time values by 10,000 milliseconds, or 10 seconds:

```

import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
public class BasicFileAttributeViewSample {
    public static void main(String[] args) throws IOException {
        Path path = Paths.get("/Projects/Index.xml");
        BasicFileAttributeView view = Files.getFileAttributeView(
            path, BasicFileAttributeView.class);
        BasicFileAttributes data = view.readAttributes();
        FileTime lastModifiedTime = FileTime.fromMillis(
            data.lastModifiedTime().toMillis() + 10_000);
        view.setTimes(lastModifiedTime, null, null);
    }
}

```

Notice that although we called Files.getFileAttributeView(), we were still able to retrieve a BasicFileAttributes object by calling readAttributes() on the resulting view.

Since there is only one update method, setTimes(FileTime lastModifiedTime, FileTime lastAccessTime, FileTime createTime) in the BasicFileAttributeView class, and it takes three arguments, we need to pass three values to the method.

NIO.2 allows us to pass null for any date/time value that we do not wish to modify. For example, the following line of code would change only the last-modified date/time, leaving the other file date/time values unaffected:

```
view.setTimes(lastModifiedTime, null, null);
```

NIO.2 Stream Methods

Prior to Java 8, the techniques used to perform complex file operations in NIO.2, such as searching for a file within a directory tree, were extremely verbose and often required you to define an entire class to perform a simple task.

When Java 8 was released, new methods that rely on streams were added to the NIO.2 specification that allow you to perform many of these complex operations with a single line of code.

Directory Walking

File systems are organised in a hierarchical manner. For example, a directory can contain files and other directories, which can in turn contain other files and directories. Every record in a file system has exactly one parent, with the exception of the root directory.

This is organised as a tree with a single root node and many branches and leaves, as shown here:

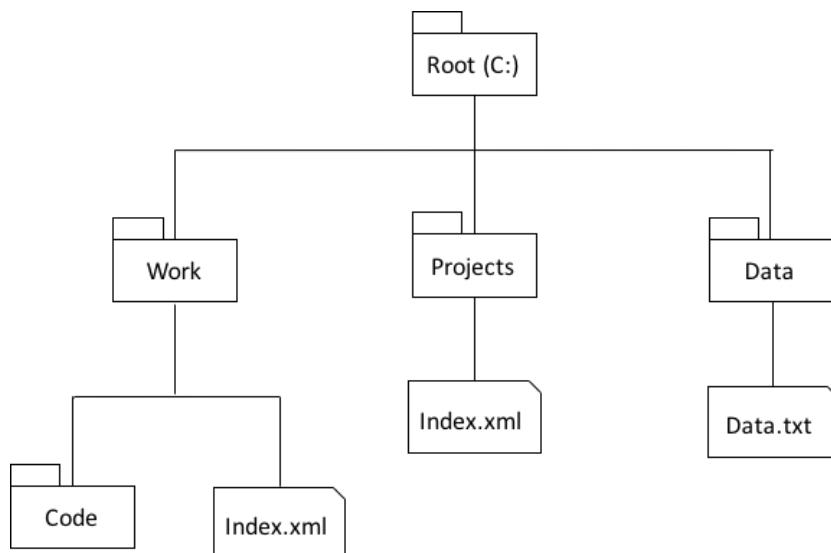


Figure 9-2 Directory Tree

A common task in a file system is to iterate over the descendants of a particular file path, either recording information about them or, more commonly, filtering them for a specific set of files. For example, you may want to search a folder and print a list of all of the .xml files. Furthermore, file systems store file records in a hierarchical manner. Generally speaking, if you want to search for a file, you have to start with a parent directory, read its child elements, then read their children, and so on.

Walking or traversing a directory is the process by which you start with a parent directory and iterate over all of its descendants until some condition is met or there are no more elements over which to iterate.

The starting path is usually a relevant directory to the application as it would be time consuming to search the entire file system.

Search Strategies

There are two common strategies associated with walking a directory tree: a depth-first search and a breadth-first search.

A depth-first search traverses the structure from the root to an arbitrary leaf and then navigates back up toward the root, traversing fully down any paths it skipped along the way. The search depth is the distance from the root to current node. For performance reasons, some processes have a maximum search depth that is used to limit how many levels deep the search goes before stopping.

Alternatively, a breadth-first search starts at the root and processes all elements of each particular depth, or distance from the root, before proceeding to the next depth level. The results are ordered by depth, with all nodes at depth 1 read before all nodes at depth 2, and so on.

The Streams API uses depth-first searching with a default maximum depth of Integer.MAX_VALUE.

Walking a Directory

As you saw earlier in the course, Java 8 includes a new Streams API for performing complex operations in a single line of code using functional programming and lambda expressions.

The first newly added NIO.2 stream-based method that we will cover is one used to traverse a directory. The Files.walk(path) method returns a Stream<Path> object that traverses the directory in a depth-first, lazy manner.

By lazy, we mean the set of elements is built and read while the directory is being traversed. For example, until a specific subdirectory is reached, its child elements are not loaded. This performance enhancement allows the process to be run on directories with a large number of descendants in a reasonable manner.

Here is an example of using a stream to walk a directory structure:

```
Path path = Paths.get("/bigcats");
try {
    Files.walk(path)
        .filter(p -> p.toString().endsWith(".xml"))
        .forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

This example iterates over a directory and outputs all of the files that end with a java extension. You can see that the method also throws an IOException, as there could be a problem reading the underlying file system. Sample output for this method could be:

```
/Work/Index.xml
/Projects/Index.xml
```

you may have come across the FileVisitor interface pattern, which was able to achieve roughly the same thing. However the new Stream API technique can do in one line what would normally require an entire class definition to do.

By default, the method iterates up to Integer.MAX_VALUE directories deep, although there is an overloaded version of walk(Path, int) that takes a maximum directory depth integer value as the second parameter.

A value of 0 indicates the current path record itself. In the previous example, you would need to specify a value of at least 1 to print any child record. In practice, you may want to set a limit to prevent your application from searching too deeply on a large directory structure and taking too much time.

You see that the Stream<Path> object returned by the walk() method visits every descendant path, with the filter being applied as each path is encountered. In the next section, you will see that there is a more useful method for filtering files available in NIO.2.

Circular Filesystem Paths

Unlike the earlier NIO.2 methods, the walk() method will not traverse symbolic links by default.

Following symbolic links could result in a directory tree that includes other, seemingly unrelated directories in the search. For example, a symbolic link to the root directory in a subdirectory means that every file in the system may be traversed.

Worse yet, symbolic links could lead to a cycle. A cycle is an infinite circular dependency in which an entry in a directory is an ancestor of the directory.

If you have a situation where you need to change the default behaviour and traverse symbolic links, NIO.2 offers the FOLLOW_LINKS option as a vararg to the walk() method. It is recommended to specify an appropriate depth limit when this option is used. Also, be aware that when this option is used, the walk() method will track the paths it has visited, throwing a FileSystemLoopException if a cycle is detected.

Searching a Directory

In the previous example, we applied a filter to the Stream<Path> object to filter the results, although NIO.2 provides a more direct method.

The Files.find(Path, int, BiPredicate) method behaves in a similar manner as the Files.walk() method, except that it requires the depth value to be explicitly set along with a BiPredicate to filter the data. Like walk(), find() also supports the FOLLOW_LINK vararg option.

A BiPredicate is an interface that takes two generic objects and returns a boolean value of the form (T, U) -> boolean. In this case, the two object types are Path and BasicFileAttributes, which you saw earlier in the chapter. NIO.2 automatically loads the BasicFileAttributes object for you, allowing you to write complex lambda expressions that have direct access to this object.

Here's some sample code:

```
Path path = Paths.get("/bi/gcats");
long dateFilter = 1420070400000L;
try {
    Stream<Path> stream = Files.find(path, 10,
        (p, a) -> p.toString().endsWith(".xml")
        && a.lastModifiedTime().toMillis() > dateFilter);
    stream.forEach(System.out::println);
} catch (Exception e) {
    // Handle file I/O exception...
}
```

This example is similar to the previous `Files.walk()` example in that it will search a directory for files that end with the `.xml` extension. It is more advanced, though, in that it applies a last-modified-time filter using the `BasicFileAttributes` object. Finally, it sets the directory depth limit for search to 10, rather than relying on the default `Integer.MAX_VALUE` value that the `Files.walk()` method uses.

Listing Directory Contents

You may remember the `listFiles()` method that operated on a `java.io.File` instance and returned a list of `File` objects representing the contents of the directory that are direct children of the parent.

Although you could use the `Files.walk()` method with a maximum depth limit of 1 to perform this same task, NIO.2 includes a new stream method, `Files.list(Path)`, that does this for you.

Consider the following code snippet, assuming that the current working directory is `/Work`:

```
try {
    Path path = Paths.get("Code");
    Files.list(path)
        .filter(p -> !Files.isDirectory(p))
        .map(p -> p.toAbsolutePath())
        .forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The code snippet iterates over a directory, outputting the full path of the files that it contains. Depending on the contents of the file system, the output might look something like the following:

```
/Work/Code>HelloWorld.java
/Work/Code>HelloWorld.class
/Work/Code/TestConsole.java
/Work/Code/TestConsole.class
```

Contrast this method with the Files.walk() method, which traverses all subdirectories. Files.list() searches one level deep and is similar to java.io.File.listFiles(), except that it relies on streams.

Printing File Contents

Earlier in the chapter, we presented Files.readAllLines() and commented that using it to read a very large file could result in an OutOfMemoryError problem.

Luckily, NIO.2 includes a Files.lines(Path) method that returns a Stream<String> object and does not suffer from this same issue. The contents of the file are read and processed lazily, which means that only a small portion of the file is stored in memory at any given time.

Let's look at Files.lines(), which is equivalent to the previous Files.readAllLines() sample code:

```
Path path = Paths.get("/Work/Index.xml");
try {
    Files.lines(path).forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

The first thing you may notice is that this example is a lot shorter, accomplishing in a single line what took multiple lines earlier. It is also more performant on large files, since it does not require the entire file to be read and stored in memory.

Taking things one step further, we can leverage other stream methods for a more powerful example:

```
Path path = Paths.get("/Work/Index.xml");
try {
    System.out.println(Files.lines(path)
        .filter(s -> s.startsWith("HIGH"))
        .map(s -> s.substring(5))
        .collect(Collectors.toList()));
} catch (IOException e) {
    // Handle file I/O exception...
}
```

This sample code now searches for lines in the file that start with HIGH, outputting everything after it to a single list that is printed to the user. You can see that lambda expressions coupled with NIO.2 allow us to perform very complex file operations concisely.

IO File and NIO.2 Methods

Here is a table showing the equivalent methods between File class and NIO.2:

File Method	NIO.2 Method
file.exists()	Files.exists(path)
file.getName()	path.getFileName()
file.getAbsolutePath()	path.toAbsolutePath()
file.isDirectory()	Files.isDirectory(path)
file.isFile()	Files.isRegularFile(path)
file.isHidden()	Files.isHidden(path)
file.length()	Files.size(path)
file.lastModified()	Files.getLastModifiedTime(path)
file.setLastModified(time)	Files.setLastModifiedTime(path,fileTime)
file.delete()	Files.delete(path)
file.renameTo(otherFile)	Files.move(path,otherPath)
file.mkdir()	Files.createDirectory(path)
file.mkdirs()	Files.createDirectories(path)
file.listFiles()	Files.list(path)

Summary

This chapter introduced NIO.2 for working with files and directories using the Path interface. You should now know what the NIO.2 Path interface is and how it differs from the legacy java.io.File class. You should be familiar with how to create and use Path objects, including how to combine or resolve them with other Path objects.

This chapter looked at various static methods available in the Files helper class. As discussed, the name of the function often tells you exactly what it does. We explained that most of these methods are capable of throwing an IOException and many take optional vararg enum values.

You also saw how NIO.2 provides methods for reading and writing file metadata using views. Java uses views to retrieve all of the file system attributes for a file without numerous round-trips to the operating system. NIO.2 also includes support for operating system specific file attributes, such as those found in Windows, Mac, and Linux file systems.

You should now be familiar with the BasicFileAttributes and BasicFileAttributeView classes.

With the introduction of functional programming in Java 8, the NIO.2 Files class was updated with new methods that use the lambda expressions and streams to process files and directories. You should know how to apply the Streams API in NIO.2 to walk a directory tree, search for files, and list the contents of a directory or file.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

CHAPTER 10

JDBC

Introduction

JDBC stands for Java Database Connectivity.

This chapter will introduce you to the basics of accessing databases from Java. You will cover the key interfaces for how to connect, perform queries, and process the results.

Data is information. A piece of data is one fact, such as your first name. A database is an organised collection of data. A filing cabinet is a type of database. It has different information stored in each folders and organised in some way, often alphabetically.

Perhaps you have stored data in a spreadsheet where there are multiple columns each containing a different bit of information. Each row is a new record of something – invoices, jobs etc. This is a bit like a database table but a database has many tables and they are linked together.

A relational database is a database that is organised into tables, which consist of rows and columns. You can think of each table as a tab of a spreadsheet file.

In this course we will use a small sample database. It has two tables: employee and department. Each employee is in a department and each department is made up of employees. These two tables are linked together to enforce the business rule that ‘every employee must be in a department’.

Here is an extract from the database:

department	
department_id INTEGER	name VARCHAR(255)
1	HR
2	Sales
3	IT
4	Strategy

employee				
employee_id INTEGER	first_name VARCHAR(255)	last_name VARCHAR(255)	department_id INTEGER	job_title VARCHAR(255)
1	Marta	Torres	1	Manager
2	Mel	Blank	1	Assistant
5	Kelly	Raman	2	Sales Representative
7	Ali	Udoka	4	Business Architect

Figure 10-1 Database Tables

The following SQL (Structured Query Language) code was used to set up the database tables:

```

CREATE TABLE department (department_id INTEGER PRIMARY KEY,
                        name VARCHAR(255))
CREATE TABLE employee (employee_id INTEGER PRIMARY KEY,
                      first_name VARCHAR(255), last_name VARCHAR(255),
                      department_id INTEGER REFERENCES department(department_id),
                      job_title VARCHAR(255))

```

The following SQL was used to populate the tables with data:

```
INSERT INTO department VALUES (1, 'HR')
INSERT INTO department VALUES (2, 'Sales')
...
INSERT INTO employee VALUES (1, 'Marta', 'Torres', 1, 'Manager')
INSERT INTO employee VALUES (2, 'Mel', 'Blank', 1, 'Assistant')
...
```

SQL commands or statements that affect the structure such as the CREATE statements that set up the tables above are called DDL (Data Definition Language).

There are four types of SQL statement for working with the data in tables, known as DML (Data Manipulation Language).

- **INSERT:** Add a new row to the table
- **SELECT:** Retrieve data from the table
- **UPDATE:** Change the data in rows in the table
- **DELETE:** Remove rows from the table

You saw the INSERT statement used to populate the tables above.

Here is an example of using the SELECT statement:

```
SELECT * FROM EMPLOYEE
```

This returns all the rows in the employee table but you can be selective:

```
SELECT * FROM EMPLOYEE WHERE DEPARTMENT_ID = 1
```

This returns only those employees in the HR department.

JDBC Interfaces

To interact with databases using Java you need to know four key interfaces of JDBC.

The interfaces are declared in the JDK. This is just like all the other interfaces and classes that you've seen in this book.

As you know, interfaces need a concrete class to implement them in order to be useful. These concrete classes come from the database vendor's JDBC driver.

Each database has a different JAR file with these classes. For example, Oracle's JAR is called something like ojdbc6.jar. MySQL's JAR is called something like mysql-connector-java-5.1.36.jar. The exact name depends on the version of the driver JAR.

This driver JAR contains an implementation of these key interfaces along with some others. The key is that the provided implementations know how to communicate with a database. Since they all implement the interfaces, all you have to do is code to the interface.

You don't actually need to know what the implementing classes are called in any real database. The point is that you shouldn't know. With JDBC, you use only the interfaces in your code and never the implementation classes directly. In fact, they might not even be public classes.

The main four interfaces are:

- Driver: Knows how to get a connection to the database
- Connection: Knows how to communicate with the database
- Statement: Knows how to run the SQL
- ResultSet: Captures what was returned by a SELECT query

There is one real class that you will use called java.sql.DriverManager. All the JDBC database interfaces and classes are in the package java.sql.

In this next example, we show you what JDBC code looks like end to end. If you are new to JDBC, just notice that three of the four interfaces are in the code.

```
package jdbc;
import java.sql.*;
public class DatabaseConnection {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:derby:hr";
        String query = "select last_name from employee";
        try (Connection conn = DriverManager.getConnection(url)) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(query));
            while (rs.next()) {
                System.out.println(rs.getString(1));
            }
        }
    }
}
```

Connecting to a Database

The first step in doing anything with a database is establishing a connection to it.

First you need to build a JDBC URL to specify which database to use. Then you need to get a Connection to the database.

Building the JDBC URL

To access a website, you need to know the URL of the website. To access your email, you need to know your username and password. JDBC is no different. In order to access a database, you need to know this information about it.

Unlike Web URLs, JDBC URLs have a variety of formats. They have three parts in common, as shown here:

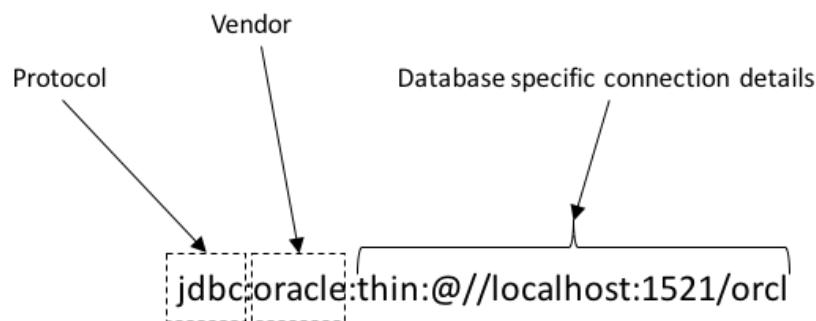


Figure 10-2 JDBC URL

The first part is always the same, it is the protocol `jdbc`. The second part is the name of the database such as `derby`, `mysql`, or `postgres`. The third part is “the rest of it,” which is a database-specific format. Colons separate the three parts.

The third part typically contains the location and the name of the database. The syntax varies. You need to know about the three main parts. You don’t need to memorize the vendor-specific part. You’ve already seen one such URL:

```
jdbc: derby: hr
```

Notice the three parts. It starts with `jdbc`, then comes `derby`, and it ends with the database name. Other examples are shown here:

```
jdbc: postgresql : //l ocal host/sa  
jdbc: oracle: thi n: @123. 123. 123. 123: 1521: sa  
jdbc: mysql : //l ocal host: 3306/sa?profleSQL=true
```

Getting the Database Connection

There are two main ways to get a Connection: `DriverManager` or `DataSource`. `DriverManager` is the one covered in the OCP exam.

Do not use a DriverManager in professional code however. A DataSource is a factory, and it has more features than DriverManager. For example, it can pool connections or store the database connection info outside the application.

The DriverManager class is in the JDK, as it is an API that comes with Java.

It uses the factory pattern, which means that you call a static method to get a Connection. The factory method has an easy-to-remember name, getConnection().

To get a Connection from the embedded database, you write the following:

```
import java.sql.*;
public class TestConnect {
    public static void main(String[] args) throws SQLException {
        Connection conn = DriverManager.getConnection("jdbc:derby:hr");
        System.out.println(conn);
    }
}
```

Running this example as java TestConnect will give you an error that begins with this:

```
Exception in thread "main" java.sql.SQLException: No suitable driver found for jdbc:derby:zoo at
java.sql.DriverManager.getConnection(DriverManager.java:689) at
java.sql.DriverManager.getConnection(DriverManager.java:270)
```

The class SQLException means “something went wrong when connecting to or accessing the database.” In this case, we didn’t tell Java where to find the database driver JAR file. Remember that the implementation class for Connection is found inside a driver JAR.

We try this again by adding the classpath with:

```
java -cp "<java_home>/db/lib/derby.jar:." TestConnect
```

Remember to substitute the location of where Java is installed on your computer for <java_home>. (If you are on Windows, replace the colon with a semicolon.) This time the program runs successfully and prints something like the following:

```
org.apache.derby.impl.jdbc.EmbedConnection@1372082959
(XID = 156), (SESSIONID = 1), (DATABASE = zoo), (DRDAID = null)
```

The details of the output aren’t important. Just notice that the class is not Connection. It is a vendor implementation of Connection.

Getting a Statement

In order to run SQL, you need to tell a Statement about it. Getting a Statement from a Connection is easy:

```
Statement stmt = conn.createStatement();
```

As you will remember, Statement is one of the four core interfaces on the exam. It represents a SQL statement that you want to run using the Connection.

That's the simple signature. There's another one that you need to know for the exam:

```
Statement stmt = conn.createStatement( ResultSet.TYPE_FORWARD_ONLY,  
    ResultSet.CONCUR_READ_ONLY);
```

This signature takes two parameters. The first is the ResultSet type, and the other is the ResultSet concurrency mode. You have to know all of the choices for these parameters and the order in which they are specified. Let's look at the choices for these parameters.

ResultSet Type

By default, a ResultSet is in TYPE_FORWARD_ONLY mode. This is what you need most of the time. You can go through the data once in the order in which it was retrieved.

Two other modes that you can request when creating a Statement are TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE. Both allow you to go through the data in any order. You can go both forward and backward. You can even go to a specific spot in the data. Think of this like scrolling in a browser. You can scroll up and down. You can go to a specific spot in the result.

The difference between TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE is what happens when data changes in the actual database while you are busy scrolling. With TYPE_SCROLL_INSENSITIVE, you have a static view of what the ResultSet looked like when you did the query. If the data changed in the table, you will see it as it was when you did the query.

With TYPE_SCROLL_SENSITIVE, you would see the latest data when scrolling through the ResultSet.

ResultSet Concurrency Mode

By default, a ResultSet is in CONCUR_READ_ONLY mode. This is what you need most of the time. It means that you can't update the result set. Most of the time, you will use INSERT, UPDATE, or DELETE SQL statements to change the database rather than a ResultSet.

There is one other mode that you can request when creating a Statement. Unsurprisingly, it lets you modify the database through the ResultSet. It is called CONCUR_UPDATABLE.

Databases and JDBC drivers are not required to support CONCUR_UPDATABLE.

If the mode you request isn't available, the driver can downgrade you. This means that if you ask for CONCUR_UPDATABLE, you will likely get a Statement that is CONCUR_READ_ONLY.

Executing SQL Statements

Once you have a Statement object, you can run a SQL statement.

The way you run SQL varies depending on what kind of SQL statement it is. You may not be an expert on SQL, but you do need to know what the first keyword means.

Let's start out with statements that change the data in a table. That would be SQL statements that begin with DELETE, INSERT, or UPDATE. They typically use a method called executeUpdate(). The name is a little tricky because the SQL UPDATE statement is not the only statement that uses this method.

The method takes the SQL statement to run as a parameter. It returns the number of rows that were inserted, deleted, or changed. Here's an example of all three update types:

```
Statement stmt = conn.createStatement();
int result = stmt.executeUpdate(
    "insert into department values(5, 'Production')");
System.out.println(result); // 1
result = stmt.executeUpdate(
    "update employee set job_title = '' where job_title = 'None'");
System.out.println(result); // 0
result = stmt.executeUpdate(
    "delete from employee where employee_id = 101");
System.out.println(result); // 1
```

Next, let's look at a SQL statement that begins with SELECT. This time, we use the executeQuery() method:

```
ResultSet rs = stmt.executeQuery("select * from employee");
```

Since we are running query to get a result, the return type is ResultSet. In the next section, we will show you how to process the ResultSet.

There's a third method called execute() that can run either a query or an update. It returns a boolean so that we know whether there is a ResultSet. That way, we can call the proper method to get more detail. The pattern looks like this:

```
boolean isResultSet = stmt.execute(sql);
if (isResultSet) {
    ResultSet rs = stmt.getResultSet();
    System.out.println("ran a query");
} else {
    int result = stmt.getUpdateCount();
    System.out.println("ran an update");
}
```

If sql is a SELECT, the boolean is true and we can get the ResultSet. If it is not a SELECT, we can get the number of rows updated.

PreparedStatement

In many situations it is inadvisable to use Statement directly. You should use a subclass called PreparedStatement. This subclass has three advantages: performance, security, and readability.

Performance: In most programs you run similar queries multiple times. A PreparedStatement figures out a plan to run the SQL well and remembers it.

Security: Suppose you have this method:

```
private static void Delete(Connecti on conn, String name)
                           throws SQLException {
    Statement stmt = conn.createStatement();
    String sql = "del ete from department where name = ' " + name + " ' ";
    System.out.println(sql);
    stmt.executeUpdate(sql); }
```

This method appears to delete the row that matches the given name. Imagine that this program lets a user type in the name. If the user's String is "Old HR", this works out well and one row gets deleted. What happens if the user's String is "any' or 1 = 1 or name='any"? The generated SQL is

```
del ete from department where name = ' any' or 1 = 1 or name=' any'
```

This deletes every row in the table. That's not good. In fact, it is so bad that it has a name: SQL injection. Upon first glance, the solution is to prevent single quotes in the user's input. It turns out to be more complicated than that because the bad guys know many ways of doing bad things. Luckily, you can just write this:

```
PreparedStatement ps = conn.prepareStatement("del ete from department
                                             where name = ?");
ps.setString(1, name);
ps.execute();
```

The JDBC driver takes care of all the escaping for you. This is convenient.

Readability: It's nice not to have to deal with string concatenation in building a query string with lots of variables.

Using a ResultSet

By far, the most common type of ResultSet is of type forward-only. We will start by looking at how to get the data from one of these. Then after going through the different methods to get columns by type, you will see how to work with a scrollable ResultSet.

Reading a ResultSet

When working with a forward-only ResultSet, most of the time you will write a loop to look at each row. The code looks like this:

```
Map<Integer, String> idToNameMap = new HashMap<>();
ResultSet rs = stmt.executeQuery("select department_id, name from
                                  department");
while(rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    idToNameMap.put(id, name);
}
System.out.println(idToNameMap); // {1=HR, 2=Sales, 3=IT, 4=Strategy}
```

There are a few things to notice here. First, we use the `executeQuery()` method since we want to have a ResultSet returned. On the next line we loop through the results. Each time through the loop represents one row in the ResultSet. The `getX` methods demonstrate how to get the column data for a given row.

A ResultSet has a cursor, which points to the current location in the data. The cursor starts out pointing to the location before the first row in the ResultSet. On the first loop iteration, `rs.next()` returns true and the cursor moves to point to the first row of data. On the second loop iteration, `rs.next()` returns true again and the cursor moves to point to the second row of data. The next call to `rs.next()` returns false. The cursor advances past the end of the data. The false signifies that there is no data available to get.

There is another way to access the columns. You can use an index instead of a column name. The column name is better because it is clearer what is going on when reading the code. It also allows you to change the SQL to reorder the columns.

Attempting to access a column that does not exist throws a `SQLException`, as does getting data from a ResultSet when it isn't pointing at a valid row.

Getting Column Data

There are lots of `getX` methods on the ResultSet interface. They are easy to remember since they are called `get`, followed by the name of the type you are getting.

This table shows the `get` methods that you need to know. The first column shows the method name, and the second column shows the type that Java returns. The third column shows the type name that could be in the database. There is some variation by database, so check your specific database documentation.

Method	Return Type	Database Type
getBoolean	boolean	BOOLEAN
getDate	java.sql.Date	DATE
getDouble	double	DOUBLE
getInt	int	INTEGER
getLong	long	BIGINT
getObject	Object	any type
getString	String	CHAR, VARCHAR
getTime	java.sql.Time	TIME
getTimeStamp	java.sql.Timestamp	TIMESTAMP

Databases typically have three ways to store date and time information as shown in the table above.

As the names suggest, DATE has just the date and TIME has just the time. TIMESTAMP has both.

If you call getTime() on a TIMESTAMP column then only the time part is returned. Similarly if you call getDate() on a TIMESTAMP column then you only get the date part.

You can convert these to the Java 8 java.time.LocalDate and java.time.LocalTime data types. This is accomplished with the instance methods toLocalDate() and toLocalTime().

If you call getTimeStamp() on a TIMESTAMP column then you get both date and time and the java.sql.Timestamp class also has an instance method called toLocalDateTime() that returns a java.time.LocalDateTime object.

Finally, the getObject method can return any type. For a primitive, it uses the wrapper class. Let's look at an example:

```
String query = "select department_id, name from department";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
    Object idField = rs.getObject("department_id");
    Object nameField = rs.getObject("name");
    if (idField instanceof Integer) {
        int id = (Integer) idField;
        System.out.println(id);
    }
    if (nameField instanceof String) {
        String name = (String) nameField;
        System.out.println(name);
    }
}
```

This code uses getObject() to get an unknown type and then test its type with instanceof before casting it to a more specific variable type.

Scrollable ResultSet

A scrollable ResultSet allows you to position the cursor at any row.

You've already learned the next() method. A scrollable resultset also has a previous() method, which does the opposite. It moves backward one row and returns true if pointing to a valid row of data.

There are also methods to start at the beginning and end of the ResultSet. The first() and last() methods return a boolean for whether they were successful at finding a row. The beforeFirst() and afterLast() methods have a return type of void, since it is always possible to get to a spot that doesn't have data.

Here is some sample code:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
String query = "select id from two_row_table order by id";
ResultSet rs = stmt.executeQuery(query);
rs.afterLast();
System.out.println(rs.previous()); // true
System.out.println(rs.getInt(1)); // 2
System.out.println(rs.previous()); // true
System.out.println(rs.getInt(1)); // 1
System.out.println(rs.last()); // true
System.out.println(rs.getInt(1)); // 2
System.out.println(rs.first()); // true
System.out.println(rs.getInt(1)); // 1
rs.beforeFirst();
System.out.println(rs.getInt(1)); // throws SQLException
```

This table only has two rows but even if it had thousands, calling beforeFirst() and then trying to call a getX() method will always cause an exception to be thrown.

It is perfectly possible to get an empty ResultSet, especially if you use the WHERE clause in your SQL. This filters the results and so perhaps no records match your criteria.

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
String query = "select id from two_row_table order by id where id = 0";
ResultSet rs = stmt.executeQuery(query);
System.out.println(rs.first()); // false
System.out.println(rs.last()); // false
```

When the cursor tries to move to the first or last row, the methods return false. There aren't any rows, which makes it impossible to point to a row of data.

Another method that you need to know is absolute(). It takes the row number to which you want to move the cursor as a parameter. A positive number moves the cursor to that numbered row. Zero moves the cursor to a location immediately before the first row. A negative number starts counting from the end of the RowSet.

The signature is:

```
boolean absolute(int row) throws SQLException
```

Finally, there is a `relative()` method that moves forward or backward the requested number of rows. It returns a boolean if the cursor is pointing to a row with data.

Closing Database Resources

It is very important to close resources when you are finished with them. This is equally true for JDBC resources. JDBC resources e.g. a Connection, are expensive to create. Not closing them creates a resource leak that will eventually slow down your program.

Repeating the example from earlier in the chapter, we have the following:

```
public static void main(String[] args) throws SQLException {
    String url = "jdbc:derby:hr";
    try (Connection conn = DriverManager.getConnection(url)) {
        Statement stmt = conn.createStatement();
        ResultSet rs =
            stmt.executeQuery("select name from department"));
        while (rs.next()) {
            System.out.println(rs.getString(1));
        }
    }
}
```

Notice how this code uses the try-with-resources syntax.

Remember that a try-with-resources statement closes the resources in the reverse order from which they were opened. This means that the ResultSet is closed first, followed by the Statement, and then the Connection. This is the correct order to close resources.

While it is a good habit to close all three resources, it isn't strictly necessary. Closing a JDBC resource should close any resources that it created. In particular, the following are true:

- Closing a Connection also closes the Statement and ResultSet.
- Closing a Statement also closes the ResultSet.

There's another way to close a ResultSet. JDBC automatically closes a ResultSet when you run another SQL statement from the same Statement.

SQL Exceptions

Up until this point in the chapter, we've overlooked the fact that your code might cause an exception to be thrown. Ok, perhaps you noticed that a checked SQLException might be thrown by any JDBC method, but none of the code so far has actually dealt with it.

We just declared it and let the caller deal with it. Now let's try catching the exception:

```
String url = "jdbc:derby:hr";
String query = "select not_a_column from department";
try (Connection conn = DriverManager.getConnection(url)) {
    Statement stmt = conn.createStatement(query);
    ResultSet rs = stmt.executeQuery() {
        while (rs.next()) {
            System.out.println(rs.getString(1));
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
        System.out.println(e.getSQLState());
        System.out.println(e.getErrorCode());
    }
}
```

The output looks like this:

```
ERROR: column "not_a_column" does not exist Position: 8
42703 0
```

Each of these methods gives you a different piece of information.

The getMessage() method returns a human-readable message as to what went wrong. The getSQLState() method returns a code as to what went wrong. You can look up the name of your database and the SQL state on the Web to get more information about the error.

However, getErrorCode() is a database-specific code and on this database, it doesn't do anything.

Summary

There are four key SQL statements: SELECT reads data, INSERT creates a new row, UPDATE changes existing data, and DELETE removes existing data.

JDBC uses four key interfaces: Driver, Connection, Statement, and ResultSet. The interfaces are part of the Java API. A database-specific JAR file provides the implementations.

To connect to a database, you need the JDBC URL. A JDBC URL has three parts separated by colons. The first part is jdbc. The second part is the name of the vendor/product. The third part varies by database, but it includes the location and name of the database. The location is either localhost or an IP address followed by an optional port.

The DriverManager class provides a factory method called getConnection() to get a Connection implementation.

Modern driver JARs contain a file in META-INF/service called java.sql.Driver. This is the name of the implementation class of Driver. Older JARs do not, and they require Class.forName() to load the driver.

There are three ResultSet types that you can request when creating a Statement. If the type you request isn't available, JDBC will downgrade your request to one that is available. The default, TYPE_FORWARD_ONLY, means that you can only go through the data in order. TYPE_SCROLL_INSENSITIVE means that you can go through the data in any order, but you won't see changes made in the database while you are scrolling. TYPE_SCROLL_SENSITIVE means that you can go through the data in any order, and you will see changes made in the database.

You can request either of two modes for ResultSet concurrency when creating a Statement. Again, JDBC will downgrade your request if needed. The default, CONCUR_READ_ONLY, means that you can read the ResultSet but not write to it. CONCUR_UPDATABLE means that you can both read and write to it.

When running a SELECT SQL statement, the executeQuery() method returns a ResultSet. When running a DELETE, INSERT, or UPDATE SQL statement, the executeUpdate() method returns the number of rows that were affected. There is also an execute() method that returns a boolean to indicate whether the statement was a query.

For a forward-only result set, call rs.next() from an if statement or while loop to set the cursor position. To get data from a column, call a method like getString(1) or getString("a"). Column indexes begin with 1, not 0. Aside from the primitive getters, there are getDate(), getTime(), and getTimeStamp(). They return just the date, just the time, or both, respectively. Also, getObject() can return any type.

For a scrollable result set, you can use methods to move to an absolute() position or relative() position. Scrolling to next() and previous() are also allowed. There are also methods to go to the first() and last() rows. All of these methods return true

if the cursor is pointing to a row with data. Other methods allow you to go outside the ResultSet with beforeFirst() and afterLast().

It is important to close JDBC resources when finished with them to avoid leaking resources.

Closing a Connection automatically closes the Statement and ResultSet objects. Closing a Statement automatically closes the ResultSet object. Also, running another SQL statement closes the previous ResultSet object from that Statement.

Exercises

Please refer to Appendix 1 and complete the exercises found there as directed by your trainer.

Chapter 11

Practical Exercises

Section 1 (Java Class Design)

Exercise 1.1

Organisation of exercises in directories:

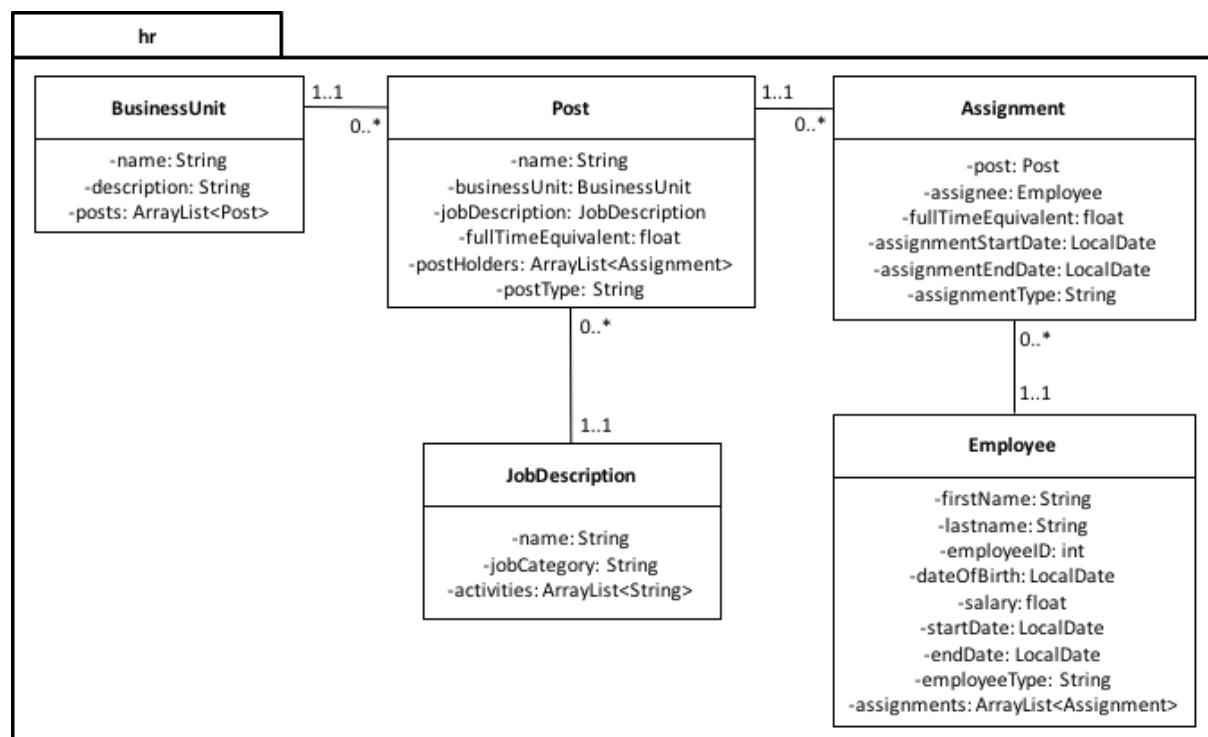
It is a good idea to keep the code for each exercise separate. The code builds up and some sections are overwritten as more techniques are learned. Keeping the code separate makes it possible to look back and see the solution to a particular problem.

To achieve this, make a copy of the previous exercise directory before starting on a new exercise. The suggested structure is based on a starting point or base directory at C:\labs\javacode. Under this directory there will be a subdirectory for each exercise called Ex1.1, Ex1.2 etc. This includes the challenge exercises.

Case study:

The aim of the exercises is to build and run the HR system using Java. This means putting the main concepts covered in the course into practice as the system is coded and tested.

Here is a class diagram for a Human Resource (HR) System showing some of the classes and the relationships between them.



1. At the command prompt, navigate to C:\labs (cd \labs) and create a directory called javaCode. Navigate into the javaCode directory.
2. Create a subdirectory called Ex1.1 in the javaCode directory
3. Create a subdirectory called hr in the Ex1.1 directory. Stay in the Ex1.1 directory.
4. Using an editor (Notepad++, Brackets etc.), create the Employee class in a file called Employee.java in the hr subdirectory.
5. Put the class in the hr package by adding the following statement as the first line in the file:

```
package hr;
```

6. Add the attributes (fields) for the Employee class, use the private access modifier (for encapsulation) and the correct data types as follows:
 - a. firstName (String)
 - b. lastName (String)
 - c. employeeId (int)
 - d. dateOfBirth (LocalDate)
 - e. salary (float)
 - f. startDate (LocalDate)
 - g. endDate (LocalDate)
 - h. employeeType (String)
 - i. assignments (ArrayList<Assignment>)
7. Initialise the assignments field with a new empty ArrayList as shown ere:

```
ArrayList<Assignment> assignments = new ArrayList<>();
```

8. This class requires the LocalDate and ArrayList classes to be imported from other packages so add the following import statements immediately after the package statement and before the class declaration:

```
import java.time.LocalDate;
import java.util.ArrayList;
```

9. Add getters for all the fields using the ‘get’ prefix followed by the field name with the initial letter capitalised so for example the getter for the firstName field would be:

```
public String getFirstName() {
    return firstName;
}
```

10. Add an additional getter called isCurrent that returns true if the employee is currently employed by the organisation. This can be worked out by comparing

the current date with the startDate and endDate fields. If the startDate is in the future then the employee has not yet started and if the endDate is in the past then the employee has left. Note that the LocalDate class provides some useful methods as follows:

- a. `now()` Static method that obtains the current date from the system clock in the default time-zone.
 - b. `of(int year, int month, int dayOfMonth)` Static method that returns a date based on the values given.
 - c. `isAfter(LocalDate other)` Checks if this date is after the specified other date.
 - d. `isBefore(LocalDate other)` Checks if this date is before the specified other date.
 - e. `isEqual(LocalDate other)` Checks if this date is the same as the specified other date.
11. Add a constructor for the Employee class that accepts firstName, lastName, employeeId, dateOfBirth and startDate. It should look like this:
- ```
public Employee(String firstName, String lastName,
 int employeeId, LocalDate dateOfBirth,
 LocalDate startDate){
 this.firstName = firstName;
 ...
}
```
12. Create setters for all the fields that are not set in the constructor except for the assignments field. For example, the setter for salary should look like this:
- ```
public void setSalary(float salary){
    this.salary = salary;
}
```
13. Create the Assignment class in the hr package but do not put any code into it yet. This is required as the assignments field in the Employee class is an ArrayList of Assignment objects. This will not compile without the Assignment class.
14. Create a new class called EmployeeTest in a package called test that will use the classes in the hr package. The new class will not be in the hr package so you will have to import all necessary hr classes.
15. Add a main method to the EmployeeTest class. It should look like this:

```
public static void main(String[] args){ ... }
```

16. Use the EmployeeTest class to test the operation of the Employee class. In the main method use the new keyword to create a test employee and then print out if the employee is current or not. The code should look like this:

```
...
Employee employee = new Employee ("Marta", "Torres ", 100,
    LocalDate.of(1980, 9, 6), LocalDate.now());
System.out.println("Employee: " + employee.getFirstName()
    + " is " + ((employee.isCurrent())?"": "not ")
    + "currently employed");
...
...
```

17. Remember to import all necessary classes including Employee before you compile and test the EmployeeTest class.
18. Try adding more Employee objects to prove that the isCurrent() method works correctly. Test as many conditions as you can, e.g. employees who have left and those who haven't started yet.
19. It is worth creating a print() method in the EmployeeTest class to avoid many repeated calls to System.out.println(). It could look like this:

```
public static void print(Employee employee){
    System.out.println("Employee name: "
        + employee.getFirstName() + " "
        + employee.getLastName()
        + ", Start date: " + employee.getStartDate()
        + ", End date: " + employee.getEndDate()
        + ((employee.isCurrent())?", Current":", Not
        current"));
}
```

20. Create a new class called EmployeeDAO in the hr package that creates and stores an ArrayList of 20 Employee objects with a variety of properties. This class will create the objects in the constructor although typically DAO objects retrieve objects from a database.
21. Use EmployeeDAO in EmployeeTest to test a broader range of Employee objects.

Exercise 1.2

1. Create a `toString()` method for the `Employee` class that returns a list of the field values (except assignments) enclosed in square brackets. Include the `isCurrent()` value at the end.
2. Use the `@Override` annotation so the compiler will check that it is correctly overridden from the `toString` method in the `Object` class.
3. Test the `toString()` method using the `EmployeeTest` class by replacing the `.`.
4. Create an `equals()` method for the `Employee` class that compares it with another `Employee` object and returns true. The comparison should be based on the `employeeId` field.
5. Ensure the compiler checks that the `equals()` method is correctly overridden.
6. Test the `equals()` method.
7. Finally, it is important to override the `hashCode()` method. Review the javadocs for the `hashCode()` and `equals()` methods of the `Object` class (search on “java 8 object”) to see the requirements for consistency between these two methods.
8. Write a suitable overridden `hashCode()` method for the `Employee` object.
9. Use the compiler annotation to check it is being correctly overridden.
10. Test the `hashmap()` and `equals()` methods to ensure they meet the requirements as stated in the Javadoc for `Object`.

Exercise 1.3

1. Create an enum for the `employeeType` to constrain the possible values it can contain. Call the enum `EmployeeType` and put the code in a file called `EmployeeType.java` in the `hr` package.
2. The possible values are `PERMANENT`, `FIXED_TERM`, `INTERN` and `CONTRACTOR`.
3. Modify the `Employee` class to use the `Enum` and test the `Employee` class.
4. Accountants have estimated the ongoing total cost of employment for permanent employees is 1.75 times their salary. Fixed-term costs are slightly lower at 1.5 times salary. Interns are only paid expenses which are not counted as an employment cost. Contractors are paid slightly higher but there is no overhead. Please note that these estimates are very much simplified.

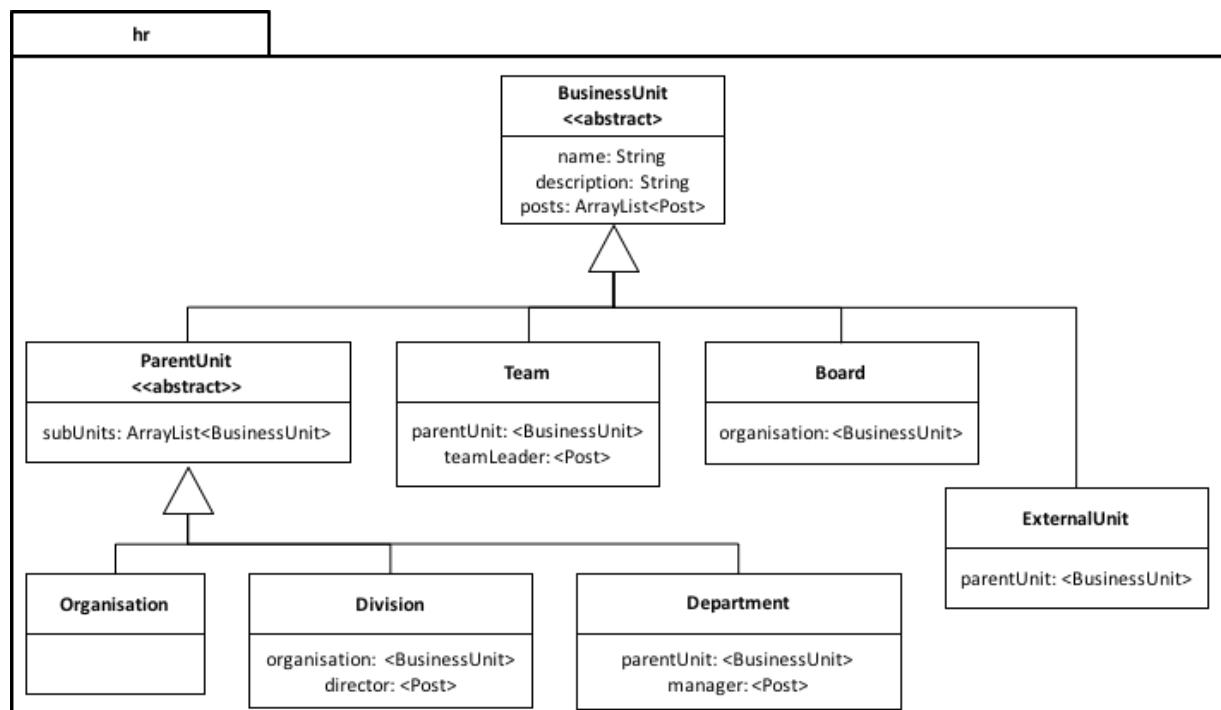
5. Use a switch statement to calculate the total cost of ownership for a selection of employees.
6. To do this, use a static method in the EmployeeTest class called printTotalCosts. This method should take a varargs parameter so you can pass any number of Employee objects into the method.
7. Ensure the report produced by the method shows the cost for each employee and the total cost at the bottom.

Exercise 1.3a (Challenge Exercise)

1. Add a float variable called totalCostRatio to the EmployeeType.
2. Create a constructor for the EmployeeType that takes a float and assigns it to the totalCostRatio variable.
3. Use the totalCostRatio variable to calculate total employment costs instead of using a switch statement in the printTotalCosts static method in the EmployeeTest class.

Exercise 1.4

Here is a class diagram showing the abstract BusinessUnit class and its subclasses:



1. Create the abstract BusinessUnit class and add the name, description and posts fields using the private modifier to encapsulate the data.
2. Initialise the posts field with a new empty ArrayList.
3. Create a constructor that takes arguments for the name and description and sets their values.
4. Create getters for all the fields.
5. Create an addPost() method that takes a Post object and adds it to the posts ArrayList. The addPost() method will set the businessUnit property of the Post object using the setBusinessUnit() method.
6. Create the Post class based on the class diagram shown previously in Exercise 1.1 including its fields, fully encapsulated, with getter methods and an appropriate toString() method.
7. Create a setter method for the businessUnit property and ensure its visibility is limited to other classes in the hr package.
8. Add a constructor that takes the name as an argument and sets the name field.
9. Create the JobDescription class as an empty class.
10. Create the abstract ParentUnit class which inherits from BusinessUnit.
11. Add the subUnits ArrayList field. Ensure it is correctly encapsulated. Initialise subUnits with an empty ArrayList.
12. Add a constructor that calls the super-class constructor using the super() keyword.
13. Create a method called addSubUnit() that takes a BusinessUnit object and adds it to the subUnits ArrayList.
14. Create a getter for the subUnits ArrayList.
15. Create the Organisation, Division and Department classes with appropriate constructors that allow the fields to be initialised.
16. Add getters for the fields in the Division and Department classes.
17. Create an appropriate toString() method for the BusinessUnit class.

18. Override the `toString()` method in those subclasses of `BusinessUnit` that have additional fields (do not include `ArrayList` fields). The parent `toString()` method may be used as a starting point.
19. Create a new class called `OrgChart` in the `hrSystem` package. Test the creation of the `BusinessUnit` classes using `OrgChart` by adding a `main()` method to the `OrgChart` class and using it to create examples of the different classes.
20. Use `OrgChart` to create a structure with an `Organisation` object at the top with some divisions, departments and sub-departments below.

Exercise 1.5

In this exercise you will override the `addSubUnit()` method to enforce the following business rules:

- Organisation can only have Division and Board as sub units
 - Division can only have Department, Team and Board as sub units
 - Department can only have Department, Team and ExternalUnit as sub units
1. Create the `Board`, `Team` and `ExternalUnit` classes as subclasses of `BusinessUnit` with the fields shown on the class diagram. Add the appropriate constructors and getters to allow them to be instantiated and their fields initialised.
 2. Override (with compiler checks) the `addSubUnit()` method in `Organisation` and provide code to verify the business rule that only `Division` and `Board` objects can be added as sub units.
 3. Override the `Division` class's `addSubUnit()` method to ensure only `Department`, `Team` and `Board` objects may be added as sub units.
 4. Enforce the business rule for `Department` in a similar way.
 5. Test the enforcement of the newly added business rules using the `OrgChart` class.

Exercise 1.5a (Challenge Exercise)

1. What other business rules need to be enforced within the hierarchy of `BusinessUnit` and its sub classes?
2. Make a list of business rules and suggestions of how each rule could be enforced by changes to the code.
3. Prioritise the code changes required to enforce the newly identified business rules and then code and test as many as you can.

Exercise 1.5b

Everyone needs to make the following changes which may have been identified in Exercise 1.5a:

1. Modify the Team, ExternalUnit and Department classes to change the type of the parentUnit field from BusinessUnit to ParentUnit.
2. Modify the Board and Division classes to change the type of the organisation field from BusinessUnit to Organisation.

Exercise 1.6

In this exercise, you will provide code to calculate the head count and vacancies for the various BusinessUnit subclasses.

Headcount may be measured in two ways:

- The simple number of budgeted heads irrespective of whether they are full or part time
- The full-time-equivalent value of all the Posts allocated to this BusinessUnit

The number of vacancies may also be calculated in either of these two ways and is the number or FTE of Posts without assignments.

Both head count and vacancies may be calculated for the current BusinessUnit or may include all sub units below this point.

Teams are a special BusinessUnit as their head count and vacancies must be included in the calculations for the parentUnit. It is also possible to obtain the head count and vacancies for a team in isolation.

BusinessUnit classes that have a director, manager or teamLeader should include that Post in the calculations.

1. Create the Assignment class including its fields, constructors and overriding the `toString()` method. Note that all fields except assignmentEndDate and assignmentType are mandatory for the constructor.
2. Create a method in the BusinessUnit class called `getHeadCount()` that calculates the headcount by counting the number of Posts.
3. Create a new Class called HeadCount in the hrSystem package that tests the `getHeadCount()` method. Note that you will have to add some Posts to a BusinessUnit subclass object in order to perform the test.

4. Override the getHeadCount() method in the Division, Department and Team to include the Post object assigned to the director, manager or teamLeader respectively. Note that if these fields are null then it should not be counted as the post has not been created for this BusinessUnit.
5. Test the changes using the HeadCount class by creating an ArrayList (or array) of BusinessUnit objects and looping through them calling the getHeadCount() method.
6. Modify the getHeadCount() method in Division and Department to include the head count of any teams they have as sub units.

Exercise 1.6a (Challenge Exercise)

Create an additional method called getHeadCountFte() in the BusinessUnit class to calculate the head count as a Full Time Equivalent value and override it in Division, Department and Team as in Exercise 1.6.

Section 2 (Java Design Patterns)

Exercise 2.1

In this exercise, you will create a Java interface to define which objects can be transferred within the organisational structure.

The three classes of objects that could be transferred are person, team and post. Each will require a target business unit to which to be transferred. The implementation of this operation will vary but the signature of the method to be called will be the same:

```
public void transfer(BusinessUnit target)
```

1. Create a new interface in the hr package called Transferable.
2. Add an abstract method transfer() that takes a target BusinessUnit object argument and has no return value.
3. Starting with the Team class you will implement the code to transfer objects between business units. First change the Team class to implement the Transferable interface. Check that the Team class no longer compiles.
4. Add an empty method that matches the signature of the transfer() method of the Transferable interface. Check that the Team class compiles as a result.
5. There is currently no way to remove a sub unit from a parent unit and this will be necessary when transferring Team objects. Create a method in the ParentUnit class called removeSubUnit() that takes a BusinessUnit object, removes it from the subUnits ArrayList and returns true if it was removed or false if not. Essentially a return value of false indicates that the BusinessUnit object was not a sub unit of the object on which the removeSubUnit() method was called.
6. Provide code for the transfer() method of the Team class that performs the following steps:
 - a. Check the target is of a suitable type
 - b. Remove this object as a sub unit of its current parent
 - c. Add this object as a sub unit of the target
 - d. Set the parent unit of this object to be the target
7. Test the removeSubUnit() and transfer() methods using a new class in the hrSystem package called Transfer.

Exercise 2.2

Now you will apply the same logic to make Post objects transferrable between BusinessUnit objects.

1. Modify the Post class to implement the Transferrable interface.
2. Unfortunately there is currently no way to remove Post objects from a BusinessUnit and this functionality will be required to implement the transfer() method. Modify the BusinessUnit class to provide a removePost() method that works in a similar way to the removeSubUnit() method you created for the ParentUnit class earlier.
3. Provide code for the transfer() method in the Post class. Many of the steps are the same as for the Team class but any instance of BusinessUnit can own a Post object so no further checking is required.
4. Test the new removePost() and transfer() methods using the Transfer class.

Exercise 2.3

In this exercise, we will return to the calculation of the total cost of employment which was previously based on the employeeType field of the Employee class. The employeeType contains enum values which are linked to a ratio used to calculate the total cost.

This will now be refactored as a new method of the Employee class called getTotalCost(). To add flexibility, we will create a Functional Interface with the single method needed for this functionality.

1. Create a new Functional Interface called TotalCost with the @FunctionalInterface annotation and add the calculateTotalCost() method which takes an Employee object and returns a float.
2. Modify the Employee class to add a new field called totalCost of type TotalCost.
3. Create an overloaded constructor for Employee that takes an extra argument of type TotalCost and sets the value of the totalCost field.
4. Create a new method in the Employee class called getTotalCost() that simply delegates to the calculateTotalCost() method of the object in the totalCost field, passing in the current Employee object using the this keyword.
5. Now use an initialisation block to set the default value of the totalCost field. This will be done using the formula used in the static method printTotalCosts() of the

EmployeeTest class in the hrSystem package. Use an anonymous inner class that implements the TotalCost interface to initialise the totalCost field.

6. Test the execution of getTotalCost and compare the results with the static method in the Employees class.
7. Now try creating an employee using an alternative implementation of the TotalCost interface as the final argument to the constructor. This can be done either by creating a new class or using an anonymous inner class. Imagine that the employee has an additional cost of £870.95 due to a software licence subscription and their overhead costs are 25% of their salary. This means the getTotalCost() method returns (salary * 1.25 + 870.95).
8. Now replace the class passed into the Employee constructor with a lambda expression and test to ensure that the same result is returned.

Exercise 2.4

It is a business rule that there can only be one Organisation object in an organisation chart. With the current design of the Organisation class it is possible to create multiple instances. In order to enforce the business rule, it has been decided to use the Singleton pattern.

1. Modify the Organisation class to add a new private static field called organisation of type Organisation.
2. Make the constructor private.
3. Make the class final as it is not possible to inherit from a class with a private constructor.
4. Create a new static method called getInstance() that returns a reference to the organisation field. This method should check if the field is null and if so initialise it using the constructor with the new keyword and then return a reference to the instance. If the field is not null then the method should just return the instance reference. This technique is called lazy instantiation. getInstance should take the constructor arguments to be used on the first instantiation.
5. Test the new Singleton Pattern implementation of the Organisation class:
 - a. First, check that Organisation can no longer be directly instantiated
 - b. Second, check that getInstance() returns an Organisation object
 - c. Finally, check that subsequent calls to getInstance() return the same object

6. Rewrite the code in the OrgChart class to use the new functionality. Note that any other classes that use the Organisation class such as HeadCount also need to be rewritten.

Exercise 2.5

So far, JobDescription has not been fully built out but now we want to introduce the functionality that Post objects have a JobDescription object. It has been decided that JobDescription objects will be immutable once created. This means setting everything in the constructor and only allowing public getter methods so nothing can be changed through the API.

1. Create the fields and getter methods for the JobDescription class according to the class diagram. Make the fields final so they are immutable.
2. Create a constructor that assigns its arguments to the class fields. Use a varargs argument for the activities ArrayList so the constructor can be called with any number of strings with which to populate the ArrayList.
3. Override the `toString()` method to include all fields including activities.
4. Test the constructor and `toString()` method with a new class called `JobDescriptionTest` in the `hrSystem` package.
5. Add code to the `JobDescriptions` class to test `getActivities()` and store the result in a suitable ArrayList. Print out the ArrayList to make sure it matches the activities of the `JobDescription` object.
6. Use the `add()` method of the ArrayList to add a new activity. Print out the ArrayList to show the new activity has been added successfully. Now print out the `JobDescription` object. You should see that the `JobDescription` object has been modified even though it is supposed to be immutable. This is because the getter method is returning a reference to the ArrayList in memory.
7. Modify the `JobDescription` class so that the `getActivities()` method returns a copy of the ArrayList in the `activities` field.
8. Run the `JobDescriptions` class again to see if the problem has been solved.
9. This problem may occur wherever there is a HAS-A relationship between classes. In this instance, we wanted the `JobDescription` class to be immutable but one of its members is an ArrayList which is mutable. This is also a problem for encapsulation of member fields that are mutable. For example, `BusinessUnit` has a private (encapsulated) ArrayList field called `posts`. We are controlling the addition and removal of `Post` objects through public methods. However users of

BusinessUnit objects can bypass these methods and do what they like with the ArrayList by obtaining a reference to it using getPosts(). If we want to allow a getter method like getPosts() but enforce encapsulation then we need to return a copy of the ArrayList as with the JobDescriptions class example. Therefore, you should re-examine the classes in the hr package to eliminate this problem.

10. Immutable classes should be declared as final to prevent subclassing which could change the behaviour to mutable. Therefore, you should make the JobDescription class final.

Exercise 2.6

The Employee class has many fields even at this early stage of the application lifecycle and it is likely to grow in complexity. This means the constructors will be long and numerous, especially if any of the fields are optional. The proposed solution is to use the Builder Pattern and create an EmployeeBuilder class.

1. Create a new class called EmployeeBuilder in the hr package.
2. Add the same set of fields as the Employee class and add setter methods for all the fields. The setter methods should all return the current EmployeeBuilder object. There is no need for a constructor.
3. Add a build() method that constructs an Employee object using the field values and returns the newly built object.
4. Compile and test the EmployeeBuilder class with a new class called EmployeeBuilderTest in the hrSystem package.
5. Modify the Employees class to use this new method of constructing Employee objects.

Exercise 2.7

The design already has multiple implementations of BusinessUnit (ParentUnit, Team, Board, Organisation, Division, Department and ExternalUnit) and the architect thinks there may be more variations on these in the future. To de-couple the implementations from the design it has been decided to introduce a new class that uses the Factory Method pattern (often just called the Factory pattern).

1. Create a new class called BusinessUnitFactory with a single static method called getBusinessUnit() that returns an object of type BusinessUnit.

2. The arguments for the factory method need to indicate the type of object to return. N.B. If other data is available such as a configuration file, the time, date or location then this can be used to decide which object to return.

As usual there are a number of options here. In some situations, the subclass type is determined by a string parameter. In this case, the type can be determined by the parameters to the factory method which will also be used to construct the class. The rules are as follows:

- a. All BusinessUnit classes have a name and description which are String arguments
- b. 1 additional argument of type ArrayList<BusinessUnit> indicates Organisation
- c. 1 additional argument of type Department indicates ExternalUnit
- d. 1 additional argument of type Organisation or Division indicates Board
- e. 2 additional arguments of type Post and ParentUnit indicate Team
- f. 3 additional arguments of type ArrayList<BusinessUnit>, Post and Organisation indicate Division
- g. 3 additional arguments of type ArrayList<BusinessUnit>, Post and ParentUnit indicate Department

Therefore, the technique used here will be overloading.

Create suitable overloaded getBusinessUnit() methods to create all 6 of the concrete subclasses of BusinessUnit in the hr package.

3. Create a new class called BusinessUnitFactoryTest in the hrSystem package and use it to build an organisation chart using the BusinessUnitFactory class. You may use the logic and some of the code from the OrgChart class.

Section 3 (Generics and Collections)

Exercise 3.1

In this exercise, we will look at techniques to sort and search collections.

1. Create a new class called EmployeeSortTest in the test package.
2. Obtain an ArrayList of Employee objects using the EmployeeDAO class.
3. Attempt to sort the ArrayList using the Collections.sort() method and note the compiler error.
4. The error results from the fact that Employee does not implement Comparable and therefore the Collections.sort() method does not know how to sort Employee objects.
5. Amend the Employee class to implement Comparable and override the compare() method based on the employeeId field.
6. Recompile and run EmployeeSortTest. Note that the resulting sort order is by employeeId.
7. Create two new Employee objects, one with an employeeId that is already in the ArrayList and one that is not.
8. Use the Collections.binarySearch() method to search for the two new objects and print out the return value. Note that this is either the index for a found object or the negative insertion point if the object was not found.
9. Modify the EmployeeSortTest to accept arguments from the command line.
10. The new class should accept all the fields needed to create a new Employee object then add it to the ArrayList at the appropriate position and re-print the list. Note this will require the use of wrapper classes such as Float and Integer as well as the parse() static method of the LocalDate class.
11. Now we want the ArrayList in a different order. Use the overloaded Collections.sort() method that takes a Comparator object as the second argument.
12. Sort the ArrayList by surname using a lambda expression for the Comparator argument and print out the list.
13. Create another sorted list based on the salary field.
14. (Challenge exercise) Design a way for the user to specify the sort order.

Exercise 3.2

A new requirement has been identified. Users of the HR System need to be able to specify an order for lists and give a reason for the ordering of each item in the list. This will be useful when deciding on BusinessUnits that need new investment as well as ranking Employee objects which are most suitable to fill a Post. Since this ordering or ranking could apply to objects of many different classes we will use generics to define the new class to contain the information. The PreferenceList needs to be in rank order so we will use a TreeMap which does this automatically.

1. Create a new class called PreferenceList in the hr package.
2. Define the class to be of a generic type by using the diamond syntax as follows:

```
public class PreferenceList<T> { ... }
```

3. Each item in the PreferenceList will contain an object of type T together with a rank and reason. The rank is an Integer and the reason is a String. One way to combine these objects is to create an inner class with two fields, one of the generic type T and one String.
4. Create an inner class called Preference with two fields, a String called reason and a field of type T called item.
5. Add a TreeMap field of type <Integer, Preference> to contain the objects.
6. To manage the list, implement the following methods:
 - a. T add(Integer rank, T item, String reason)
If the rank is already used then add the new entry and demote the existing entry and any below as necessary. Returns the displaced item or null.
 - b. T remove(Integer rank)
 - c. T set(Integer rank, T item, String reason)
Replaces and returns the existing item. If the rank is empty then the new item is just added and null is returned.
 - d. T getItem(Integer rank)
 - e. String getReason(Integer rank)
7. Test these methods with a new class called PreferenceListTest in the test package. Ensure to test that the PreferenceList works with objects of different classes e.g. that you can create a PreferenceList<BusinessUnit> as well as a PreferenceList<Employee>.
8. Create a new static method in PreferenceListTest to print a PreferenceList with any type of object in it using generic wildcards.

Section 4 (Functional Programming and Stream API)

Exercise 4.1

In this exercise, we will look at techniques to use Stream API to filter Employee objects using a variety of criteria.

1. Create a new class called EmployeeFilterTest in the test package.
2. Use the EmployeeDAO to obtain a List of Employee objects.
3. Generate a stream from the List using the stream() method.
4. Filter the stream using a predicate to find only current employees and print the out.
5. Use the count() method to show how many current employees are in the List.
6. (Challenge exercise) Use the min() and max() methods to find the longest and shortest serving employees.
7. (Challenge exercise) Use the limit() method to show the 4 highest and 4 lowest paid employees.

Section 5 (Dates, Strings and Localisation)

Exercise 5.1

In this exercise, we will look at some Java 8 date and time API techniques to manage employee review dates. Each employee needs a review with their manager after three months of employment and every six months thereafter.

1. Add a LocalDate field called lastReviewDate to the Employee class and provide a getter method.
2. Add a getNextReviewDate() method to return the next review date based on the business rule described above.
3. Create a new class called EmployeeReviewTest in the test package.
4. Use the EmployeeDAO to obtain a List of Employee objects.
5. Show a list of employees with last and next review dates.
6. Add a review(LocalDate reviewDate) to the Employee class to set the lastReviewDate field. Provide appropriate validation for this method and throw an exception if an invalid date is provided.
7. Test the operation of the review() method and ensure that the getLastReviewDate() and getNextReviewDate() methods still give the correct return values.

Exercise 5.2

In this exercise, you will produce formatted output of Employee objects in columns with easy-to-read dates and numbers.

1. Create a new class in the test package called FormatTest.
2. Use the EmployeeDAO class to obtain a list of Employee objects.
3. Use appropriate PrintWriter methods, e.g. printf() or format(), print() and println() to produce a report showing the details of employees in aligned columns with suitable column headings.
4. (Challenge exercise) Allow the user to specify the sort order when running the program. Provide a visual indication of which column is being used for sorting.

Exercise 5.3 (Challenge exercise)

In this exercise, you will modify the FormatTest program to include localisation information that is reflected in the output.

1. Modify the FormatTest class to accept optional locale information from the user on the command-line.
2. Create appropriate resource bundles for some locales (at least en_GB, fr_CA, en_CA and fr_FR) and ensure that the column headings are translated into French for the French language resource bundles.
3. If the user has supplied a locale then use this to change the column headings and formatting for numbers and dates.

Section 6 (Exceptions and Assertions)

Exercise 6.1

In this exercise, you will create a custom checked Exception class for the hr package and use it to provide information to users of the hr package classes.

1. Create a new class called HrException that subclasses Exception.
2. Provide two constructors (no-args and String message) and use these to call the matching super constructor.
3. Use the HrException with a suitable message to indicate if an invalid date is passed to the Employee review() method.

Exercise 6.2 (Challenge exercise)

In this exercise you will make EmployeeDAO more efficient by making it release its memory-intensive data storage. This is an improvement on setting it to null as this relies on garbage collection to free the memory.

1. Modify the EmployeeDAO class to make it implement the AutoCloseable interface.
2. Override the close() method of AutoCloseable to empty the ArrayList called employees and set it to null.
3. Make the close() method throw an exception if employees is already null.
4. Test EmployeeDAO by using it in a try-with-resources block.
5. Test for suppressed exceptions by forcing your code to throw an exception that closes EmployeeDAO but make sure that it was already closed so it throws an exception during the close() method.
6. Print out the main and suppressed exceptions.

Section 7 (Concurrency)

Exercise 7.1

In this exercise, you will use threads to process multiple aspects of Employee objects.

7. Create a new class called CalcSalaries in the hr package that implements Runnable.
8. Add a constructor that takes a List of Employee objects to process.
9. Write code that calculates the total and average salary for the Employee objects in the List
10. Create a new class called EmployeeThreadTest in the test package that creates an ArrayList<Employee> and passes it to the CalcSalaries constructor.
11. Create and start a Thread object that runs CalcSalaries.
12. Provide timing details to show how long the CalcSalaries takes to run.
13. Increase the size of the ArrayList to approximately 1000 and record the relationship between the number of Employee objects being processed and the time taken.
14. Add a second Runnable class called CalcLengthOfService to calculate the average length of service of employees, again accepting a List<Employee> in the constructor.
15. Modify the EmployeeThreadTest to run the CalcLengthOfService Runnable in a separate thread.
16. Record the time taken to perform these two tasks
17. (Challenge exercise) Use an ExecutorService to allow multiple threads to perform the two tasks given above.
18. Does this improve the completion time?
19. Are there any concurrency issues? If so can they be fixed with synchronized blocks/methods or by using a concurrent collection?

Section 8 (IO)

Exercise 8.1

In this exercise, you will produce a report that is stored in a file to record the state of employed staff at a specific time. This could then be emailed to managers on a regular basis.

1. Use the code produced in Exercise 5.2 above as the starting point for this exercise.
2. Create a new class called ReportFileProducerTest in the test package.
3. Use the current date and time to produce a string to use as report the file name. The format should be “EmployeeReportYYYYMMdd.txt” where YYYYMMdd represents the date the report is produced.
4. Write code to produce the formatted report from Exercise 5.2 but instead of printing the report to the screen, print it into a file.

Exercise 8.2

The report file produced above will overwrite any existing file of the same name. Since more than one may be required in the same day, there is a new requirement for sequential file numbers to be added. Any subsequent files must have a numeric suffix starting at 2.

1. Modify ReportFileProducerTest so it checks to see if today's report file already exists.
2. If so do not overwrite it but create a new file with the next available suffix number that has not yet been used.

Section 9 (NIO.2)

Exercise 9.1

In this exercise, you will allow the user to specify where they would like the file produced from Exercise 8.2 to be placed.

3. Modify ReportFileProducerTest to accept an optional directory path on the command line.
4. If a directory path is provided, check that it is a valid path and whether it exists and report back to the user.
5. If the directory does not exist, create it and report back to the user.
6. Output the report file to the directory indicated, following the same rules as before about sequential suffixes and report back to the user the full path name of the file produced.
7. (Challenge exercise) If the directory does not exist, ask the user to confirm its creation using Console input.

Section 10 (JDBC)

Exercise 10.1

In this exercise, you will retrieve data from the HR system in a database.

1. Create a new class called EmployeeDAODB in the hr package that has the same public methods and constructors as EmployeeDAO.
2. Add code to load the ArrayList from the hr database using the connection string "jdbc:derby:hr".
3. Create a new class called EmployeeDAODBTest in the test package and ensure that you can get Employee objects and print them out as you were able to do with EmployeeDAO.
4. Add new methods to the EmployeeDAODB called findAll(), findById() and findByLastName() that return a List of Employee objects matching the criteria passed in (int for id, String for name).

Exercise 10.2

In this exercise, you will persist data from the HR system in the database.

1. Modify EmployeeDAODB to add a method called insert() that takes an employee object and adds it to the database.
2. Ensure the code catches any exceptions and reports back to the caller with an appropriate message. For example, if the Employee object is missing any mandatory information or if any database constraints such as primary key uniqueness are violated.
3. Add a second method called insertAll() that takes a List<Employee> and inserts all elements returning an int representing the number of records added.
4. Check that the records have been successfully added (or not) using the findAll() method.

Exercise 10.3 (Challenge exercise)

In this exercise, you will modify existing data in the hr database.

1. Modify EmployeeDAODB to add a method called update() that takes an employee object, finds an existing record and updates the record with any changes.
2. Add a delete() method that takes an int representing the employeeId, finds a matching record and deletes it from the database.
3. Check that the records have been successfully updated and deleted using the findAll() method.

StayAhead Training Limited
6 Long Lane
London
EC1A 9HF

020 7600 6116

www.stayahead.com

All registered trademarks are acknowledged
Copyright © StayAhead Training Limited.