

Java 8 Programmer II

Study Guide:

Exam 1Z0-809

Copyright © 2016 by Esteban Herrera

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means including photocopying, recording, or information storage and retrieval without permission in writing from the author.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This publication is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. This publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein; these changes will be incorporated in new editions of the publication.

The author has taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

ISBN-13: 978-1530845996
ISBN-10: 1530845998

TABLE OF CONTENTS

Acknowledgments	ix
Introduction	xi

Part One | **Class Design**

One . . . Encapsulation and Immutable Classes	3
Two . . . Inheritance and Polymorphism	31
Three . . . Inner Classes	59
Four . . . Interfaces	101
Five . . . Enumerations	145

Part Two | **Generics and Collections**

Six . . . Generics	165
Seven . . . Collections	185

Part Three | **Lambda Expressions**

Eight . . . Functional Interfaces	203
Nine . . . Lambda Expressions	221
Ten . . . Java Built-In Lambda Interfaces	243
Eleven . . . Method References	291

Part Four | **Streams and Collections**

Twelve . . . Streams	323
Thirteen . . . Iterating and Filtering Collections	347
Fourteen . . . Optional Class	363
Fifteen . . . Data Search	375
Sixteen . . . Stream Operations on Collections	387
Seventeen . . . Peeking, Mapping, Reducing and Collecting	405
Eighteen . . . Parallel Streams	441

Part Five | **Exceptions and Assertions**

Nineteen . . . Exceptions **461**

Twenty . . . Assertions **493**

Part Six | **Date/Time API**

Twenty-one . . . Core Date/Time Classes **507**

Twenty-two . . . Time Zones and Formatting **539**

Part Seven | **Java I/O**

Twenty-three . . . Java I/O Fundamentals **573**

Twenty-four . . . NIO.2 **597**

Twenty-five . . . Files and Streams **623**

Part Eight | **Threads and Concurrency**

Twenty-six . . . Thread Basics **639**

Twenty-seven . . . Concurrency **657**

Twenty-eight . . . Fork/Join Framework **677**

Part Nine | **JDBC and Localization**

Twenty-nine . . . JDBC API **693**

Thirty . . . Localization **717**

ACKNOWLEDGMENTS

Just like when developing software, getting feedback early in the process of writing this book was vital to make a better product.

I want to thank David Kühner, who help me fix some errors in my OCPJ notes for the exam (from which this book was born). Johnathan James (<https://github.com/haxwell>) which helped me proof-read the book and corrected a lot of errors. Luigi Cortese (<http://www.devsedge.net>), who provide a lot of useful feedback to make the book better, and Bevin James, for his words of encouragement, which help me finish the book.

Finally, to my family, especially my wife Sara, for their continued support and love.

INTRODUCTION

The exam for the Oracle Certified Professional, Java SE 8 Programmer certification is **HARD**.

So I wanted to write an understandable and concise guide. One that would cover all the certification objectives. One that would cut all the fluff and that would explain in simple terms hard concepts. One that is easy to read. One that would be accessible to anyone willing to learn.

That's why I make assumptions like that you're already familiar with the basics of the language. That you know the prerequisites of the exam, how to schedule it, and in general, how all this work.

That's why I write short paragraphs. If you're reading the print version, you'll notice that the font is bigger and there's most space than in most programming books. If you're reading the e-book version, I tried my best to format the code and have syntax-highlighting.

And that's why you can read this book for **FREE** online at:

<http://ocpj8.javastudyguide.com>

The book is also on Github, so you can fork it and create pull requests:

<https://github.com/eh3rrera/ocpj8-book>

Most study guides follow the topic structure of the certification exam. This book doesn't. The chapters are organized in a way I feel it's more natural to present the topics.

Each chapter has lots of code examples, key points, and practice questions. Here are some suggestions:

- It's important that you try all the examples by yourself (coding them by hand, preferably) and play with them so that you can better understand the concept they are showing.
- The exam cover a lot of topics, you can get a good overview by going to the key points section at the end of each chapter.
- In total, this book has around 160 practice questions. I think that's a little number. Try to practice as much as you can, this is key to the pass the exam, there are a lot of good simulators in the market.

If you want to take the upgrades exams (1Z0-810 from Java 7 and 1Z0-813 from Java 6 and below), this book can also help you because the topics are pretty much the same. You can find more information at:

<http://ocpj8.javastudyguide.com/upgrade.html>

Good luck with your exam and thank you for buying this book. You can contact me anytime if you have any question, or just to say hi.

Esteban Herrera
@eh3rrera
estebanhb2@yahoo.com.mx

Part ONE

Class Design

Chapter ONE

Encapsulation and Immutable Classes

Exam Objectives

- Implement encapsulation.
- Create and use singleton classes and immutable classes.
- Develop code that uses static keyword on initialize blocks, variables, methods, and classes.
- Develop code that uses final keyword.

ENCAPSULATION

As we know, Java is an object-oriented language, and all code must be inside a class.

```
class MyClass {  
    String myField = "you";  
    void myMethod() {  
        System.out.println("Hi " + myField);  
    }  
}
```

Therefore, we have to create an object to use it:

```
MyClass myClass = new MyClass();  
myClass.myField = "James";  
myClass.myMethod();
```

One important concept in object-oriented programming languages is encapsulation, the ability to hide or protect an object's data.

Most of the time, when someone talks about encapsulation most people tend to think about private variables and public getters and setters and how overkilling this is, but actually, encapsulation is more than that, and it's helpful to create high-quality designs.

Let's consider the previous example.

Chapter ONE

First of all, it's good to hide as much as possible the internal implementation of a class. The reason, mitigate the impact of change.

For example, what if `myField` changes its type from `String` to `int`?

```
myClass.myField = 1;
```

If we were using this attribute in fifty classes, we would have to make fifty changes in our program.

But if we hide the attribute and use a setter method instead, we could avoid the fifty changes:

```
// Hiding the attr to the outside world with the private keyword
private int myField = 0;

void setMyField(String val) { // Still accepting a String
    try {
        myField = Integer.parseInt(val);
    } catch(NumberFormatException e) {
        myField = 0;
    }
}
```

You implement this attribute hiding by using access modifiers.

Java supports four access modifiers:

- **public**
- **private**
- **protected**
- **default** (when no modifier is specified)
-

You can apply these modifiers to classes, attributes and methods according to the following table:

	Class/ Interface	Class		Interface	
		Attrib	Method	Attrib	Method
public	X	X	X	X	X
private		X	X		
protected		X	X		
default	X	X	X	X	X

As you can see, all the access modifiers can be applied to attributes and methods of a class, but not necessarily to their interface counterparts. Also, class and interface definitions can only have a public or default modifier. Why? Let's define first these access modifiers.

If something is declared as `public`, it can be accessed from any other class in our application, regardless of the package or the module in which it is defined.

If something is defined as `private`, it can only be accessed inside the class that defines it, not from other classes in the same packages and not from classes that inherit the class. `private` is the most restrictive access modifier.

If something is defined as `protected`, it can be only accessed by the class that defines it, its subclasses and classes of the same package. It doesn't matter if the subclass is in another package, which makes this modifier less restrictive than `private`.

If something doesn't have a modifier, it has a default access, also known as package access, because it can only be accessed by classes within the same package. If a subclass is defined in another package, it cannot see the default access attribute or method. That is the only difference with the `protected` modifier, which makes it more restrictive.

Chapter ONE

A code example may explain this better:

```
package street21;
public class House {
    protected int number;
    private String reference;

    void printNumber() {
        System.out.println("Num: " + number);
    }
    public void printInformation() {
        System.out.println("Num: " + number);
        System.out.println("Ref: " + reference);
    }
}
class BlueHouse extends House {
    public String getColor() { return "BLUE"; }
}

...
package city;
import street21.*;
class GenericHouse extends House {
    public static void main(String args[]) {
        GenericHouse h = new GenericHouse();
        h.number = 100;
        h.reference = ""; // Compile-time error
        h.printNumber(); // Compile-time error
        h.printInformation();
        BlueHouse bh = new BlueHouse(); // Compile-time error
        bh.getColor(); // Compile-time error
    }
}
```

- `h.number` compiles because this attribute is protected and `GenericHouse` can access it because it extends `House`.
- `h.reference` doesn't compile because this attribute is private.
- `h.printNumber()` doesn't compile because this method has default (package) access.
- `h.printInformation()` compiles because this method is public.
- `BlueHouse bh = new BlueHouse()` doesn't compile because the class has default (package) access.
- `bh.getColor()` doesn't compile because although the method is public, the class that contains it, it's not.

Now, can you see why some modifiers apply to certain elements, and others don't?

Would a private or protected class make sense in an object-oriented language?

How about a private method inside an interface where most, if not all methods, have no implementation?

Think about it.

Here's a summary of the rules:

	Members same class	Subclass same package	Subclass different package	Another class same package	Another class different package
public	X	X	X	X	X
private	X				
protected	X	X	X	X	
default	X	X		X	

WHAT IS A SINGLETON?

There will be times when you might want to have only one instance for a particular class. Such a class is a *singleton* class and is a design pattern.

There are some classes of Java that are written using the singleton pattern, for example:

- `java.lang.Runtime`
- `java.awt.Desktop`

In Java, broadly speaking, there are two ways to implement a singleton class:

- With a private constructor and a static factory method
- As an enum

Let's start with the private constructor way. Although it might seem simple at first, it poses many challenges, all of them related to keeping only one instance of a singleton class through all the application life cycle.

By having a private constructor, the singleton class makes sure that no other class creates an instance of it. A private method (or in this case, the constructor) can only be used inside the class that defines it.

```
class Singleton {  
    private Singleton() {}  
}
```

So the instance has to be created inside of the class itself.

This can be done in two ways. Using a private static attribute and a method to get it:

```
class Singleton {  
    private static final Singleton instance = new Singleton();  
    private Singleton() {}  
    public static Singleton getInstance() { return instance; }  
}
```

The attribute has to be `private` so no other class can use it, only through its getter.

It has to be `static` so the instance can be created when the class is loaded before anyone can use it and because `static` members belong to the class and not to a particular instance.

And it has to be `final` so a new instance cannot be created later.

A variation of this implementation is to use a `static` inner class (we will review this type of class on Chapter 3):

```
class Singleton {  
    private Singleton() {}  
    private static class Holder {  
        private static final Singleton instance  
            = new Singleton();  
    }  
    public static Singleton getInstance() {  
        return Holder.instance;  
    }  
}
```

The advantage of this is that the instance won't be created until the inner class is referenced for the first time.

Chapter ONE

However, there will be times when for example, creating the object is not as simple as calling new, so the other way is to create the instance inside the get method:

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() { }  
    public static Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
            // more code to create the instance...  
        }  
        return instance;  
    }  
}
```

The first time this method is called, the instance will be created. But with this approach, we face another problem. In a multithreading environment, if two or more threads are executing this method in parallel, there's a significant risk of ending up with multiple instances of the class.

One solution is to synchronize the method so only one thread at a time can access it.

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() { }  
    public static synchronized Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

The problem with this is that strictly speaking, is not very efficient. We only need synchronization the first time, when the instance is created, not every single time the method is called.

An improvement is to lock only the portion of the code that creates the instance. For this to work properly, we have to double check if the instance is null, one without locking (to check if the instance is already created), and then another one inside a synchronized block (to safely create the instance).

Here's how this would look:

```
class Singleton {  
    private static Singleton instance;  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if(instance == null) {  
            synchronized (Singleton.class) {  
                if(instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

But again, this implementation is not perfect yet. This time, the problem is at the Java Virtual Machine (JVM) level. The JVM, or sometimes the compiler, can optimize the code by reordering or caching the value of variables (and not making the updates visible).

Chapter ONE

The solution is to use the `volatile` keyword, which guarantees that any read/write operation of a variable shared by many threads would be atomic and not cached.

```
class Singleton {  
    private static volatile Singleton instance;  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if(instance == null) {  
            synchronized (Singleton.class) {  
                if(instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

As you can see, it was a lot of trouble to implement a singleton correctly when you want or have to defer the instantiation of the class until you first use it (also called lazy initialization). And we are not going to cover serialization and how to keep a singleton in a cluster.

So if you don't need it, either use the first two methods (create the instance when declaring the variable or use the holder inner class) or the easier (and recommended) way, an enumeration (enum).

We'll review enums in Chapter 5, for now, just knowing that enums are singletons is enough.

IMMUTABLE OBJECTS

There will be other times when you might not want to modify the values or state of an object when used by multiple classes. Such an object will be an *immutable* object.

There are some immutable classes in the Java JDK, for example:

- `java.lang.String`
- Wrappers classes (like `Integer`)

Immutable objects cannot change after they are created. This means that they cannot have setter methods or public variables, so the only way to set its properties is through the constructor.

Immutability also means that if a method has to change the properties of the object, as the object or its values cannot change, it has to return a copy of the object with the new values (this is just how the `String` class works).

Another point to consider is inheritance. If the immutable class can be inherited, the subclass can change the methods to modify the instances of the class, so an immutable class cannot allow this.

In summary, an immutable object:

- Sets all of its properties through a constructor.
- Does not define setter methods.
- Declares all its attributes `private` (and sometimes `final`).
- Has a class declared `final` to prevent inheritance.
- Protects access to any mutable state. For example, if it has a `List` member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change).

THE STATIC KEYWORD

Think of something static as something belonging to the class and not to a particular instance of that class.

If we are talking about attributes, a static attribute is shared across all instances of the class (because, again, it doesn't belong to an instance).

Compare the output of this code:

```
public class Example {  
    private int attr = 0;  
  
    public void printAttr() {  
        System.out.println(attr);  
    }  
  
    public static void main(String args[]) {  
        Example e1 = new Example();  
        e1.attr = 10;  
        e1.printAttr();  
        Example e2 = new Example();  
        e2.printAttr();  
    }  
}
```

Output:

10
0

To the output of the code that uses a static variable:

```
public class Example {  
    private static int attr = 0;  
  
    public void printAttr() {  
        System.out.println(attr);  
    }  
  
    public static void main(String args[]) {  
        Example e1 = new Example();  
        e1.attr = 10;  
        e1.printAttr();  
        Example e2 = new Example();  
        e2.printAttr();  
    }  
}
```

Output:

```
10  
10
```

As you can see, the value is retained when another instance uses the static variable.

When talking about static methods, the method belongs to the class, which means that we don't need an instance to call it (the same applies to attributes, by the way).

Chapter ONE

```
public class Example {  
    private static int attr = 0;  
  
    public static void printAttr() {  
        System.out.println(attr);  
    }  
  
    public static void main(String args[]) {  
        Example e1 = new Example();  
        e1.attr = 10;  
        e1.printAttr();  
        // Referencing the method statically  
        Example.printAttr();  
    }  
}
```

Output:

```
10  
10
```

However, if you look closely, *printAttr* uses a static attribute, and that's the catch with static methods, they can't use instance variables, just static ones.

This makes perfect sense, if you can access a static method with just the class, there's no guarantee an instance exists, and even if it does, how do you link an attribute with its instance when you only have the class name?

Using the same logic, the keywords *super* and *this*, cannot be used either.

Static classes will be covered in Chapter 3, but there's another construct that can be marked as static, an initializer block.

A static (initializer) block looks like this:

```
public class Example {  
    private static int number;  
  
    static {  
        number = 100;  
    }  
    ...  
}
```

A block is executed when the class is initialized and in the order they are declared (if there's more than one).

Just like static methods, they cannot reference instance attributes, or use the keywords super and this.

In addition to that, static blocks cannot contain a return statement, and it's a compile-time error if the block cannot complete normally (for example, due to an uncaught exception).

FINAL KEYWORD

The final keyword can be applied to variables, methods, and classes.

When final is applied to variables, you cannot change the value of the variable after its initialization. These variables can be attributes (static and non-static) or parameters. Final attributes can be initialized either when declared, inside a constructor, or inside an initializer block.

```
public class Example {  
    private final int number = 0;  
    private final String name;  
    private final int total;  
    private final int id;  
  
    {  
        name = "Name";  
    }  
  
    public Example() {  
        number = 1; // Compile-time error  
        total = 10;  
    }  
  
    public void exampleMethod(final int id) {  
        id = 5; // Compile-time error  
        this.id = id; // Compile-time error  
    }  
}
```

When `final` is applied to a method, this cannot be overridden.

```
public class Human {  
    final public void talk() { }  
    public void eat() { }  
    public void sleep() { }  
}  
...  
public class Woman extends Human {  
    public void talk() { } // Compile-time error  
    public void eat() { }  
    public void sleep() { }  
}
```

In turn, when `final` is applied to a class, you cannot subclass it. This is used when you don't want someone to change the behavior of a class by subclassing it. Two examples in the JDK are `java.lang.String` and `java.lang.Integer`.

```
public final class Human {  
    ...  
}  
...  
public class Woman extends Human { // Compile-time error  
    ...  
}
```

In summary:

- A `final` variable can only be initialized once and cannot change its value after that.
- A `final` method cannot be overridden by subclasses.
- A `final` class cannot be subclassed.

KEY POINTS

- Encapsulation is the ability to hide or protect an object's data.
- Java supports four access modifiers: public, private, protected, default (when nothing is specified, also called package-level).
- If something is declared as public, it can be accessed from any other class of our application. Any class, regardless of the package or the module it is defined.
- If something is defined as private, it can only be accessed inside the class that defines it. Not from other classes in the same packages and not from classes that inherit the class. private is the most restrictive access modifier.
- If something is defined as protected, it can be only accessed by the class that defines it, its subclasses and classes of the same package. It doesn't matter if the subclass is in another package, which makes this modifier, less restrictive than private.
- If something doesn't have a modifier, it has default access also known as package access, because it can only be accessed by classes within the same package. If a subclass is defined in another package, it cannot see the default access attribute or method. That is the only difference with the protected modifier, making it more restrictive.
- A singleton class guarantees that there's only one instance

of the class during the lifetime of the application.

- An immutable object cannot change its values after it is created. It:
 - Sets all of its properties through a constructor
 - Does not define setter methods
 - Declares all its attributes `private` (and sometimes `final`)
 - Has a class declared `final` to prevent inheritance
 - Protects access to any mutable state. For example, if it has a `List` member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change)
- The `static` keyword can be applied to attributes, methods, blocks and classes. A `static` member belongs to the class where it is declared, not to a particular instance.
- The `final` keyword can be applied to variables (so they cannot change their value after initialized), methods (so they cannot be overridden), and classes (so they cannot be subclassed).

SELF TEST

1. Given:

```
public class Question_1_1 {  
    private final int a; // 1  
    static final int b = 1; // 2  
    public Question_1_1() {  
        System.out.print(a); // 3  
        System.out.print(b); // 4  
    }  
}
```

What is the result?

- A. 01
- B. Compilation fails on the line marked as // 1
- C. Compilation fails on the line marked as // 2
- D. Compilation fails on the line marked as // 3
- E. Compilation fails on the line marked as // 4

2. Which of the following state the correct order from the more restricted modifier to the more unrestricted?

- A. private, default, public, protected
- B. protected, private, default, public
- C. default, protected, private, public
- D. private, default, protected, public

3. Given:

```
public final class Question_1_3 {  
    private final int n;  
    public Question_1_3() { }  
    public void setN(int n) { this.n = n; }  
}
```

Which of the following is true?

- A. The class is immutable
- B. The class is not immutable
- C. Compilation fails
- D. An exception occurs at runtime

4. Given:

```
public class Question_1_4 {  
    private final List<Integer> list = new ArrayList<>();  
    public void add() {  
        list.add(0);  
    }  
    public static void main(String[] args) {  
        Question_1_4 q = new Question_1_4();  
        q.add();  
    }  
}
```

Which of the following is true?

- A. Attribute `list` contains one element after `main` is executed
- B. The class is immutable
- C. Compilation fails
- D. An exception occurs at runtime

Chapter ONE

5. Given:

```
public class Question_1_5 {  
    private String s = "Hi";  
    public static void main(String[] args) {  
        Question_1_5 q = new Question_1_5();  
        q.s = "Bye"; // 1  
        System.out.println(q.s); // 2  
    }  
}
```

What is the result?

- A. Hi
- B. Bye
- C. Compilation fails on the declaration marked as // 1
- D. Compilation fails on the declaration marked as // 2

6. Given:

```
public class Question_1_6 {  
    private static int a;  
    private int b;  
    static {  
        a = 1;  
        b = 2;  
    }  
    public static void main(String[] args) {  
        Question_1_6 q1 = new Question_1_6();  
        Question_1_6 q2 = new Question_1_6();  
        q2.b = 1;  
        System.out.println(q1.a + q2.b);  
    }  
}
```

What is the result?

- A. 0
- B. 3
- C. 2
- D. Compilation fails

ANSWERS

1. The correct answer is D.

As a final attribute, a is required to have been initialized, so compilation fails on the line marked as // 3.

2. The correct answer is D.

Private is the most restrictive of all access modifiers. public is the least restrictive of all.

Between default and protected, the latter is less restrictive since even if a subclass belongs to another package than the superclass, it can access protected members, in contrast to the default access (which has a same-package only level).

3. The correct answer is C.

As a final variable, n must be initialized either when declared, in a constructor, or a block. This, and the fact that changing its value is not allowed (in the setter method) generates a compile-time error.

4. The correct answer is A.

Although attribute list is final, this keyword only makes its reference immutable; it cannot be assigned another object (like a new List), but the values inside the object can change (if they are not final themselves). This (and other details) makes the class mutable.

5. The correct answer is B.

Attribute s is private, however, since the main method is in the same class, it can use the attribute without any problem.

6. The correct answer is D.

Attribute b is not static. A static block cannot reference a non-static variable. That's the reason compilation fails.

Chapter ONE

Chapter TWO

Inheritance and Polymorphism

Exam Objectives

- Implement inheritance including visibility modifiers and composition.
- Override hashCode, equals, and toString methods from Object class.
- Implement polymorphism.
- Develop code that uses abstract classes and methods.

INHERITANCE

At the core of an object-oriented language, there's the concept of inheritance.

In simple terms, inheritance refers to an **IS-A** relationship where a class (called superclass) provides common attributes and methods to derived or more specialized classes (called subclass).

In Java, a class is only allowed to inherit from a single superclass (singular inheritance). Of course, the only exception is `java.lang.Object`, which has no superclass. This class is the superclass of all classes.

The keyword `extends` is used to specify this relationship. For example, a hammer **IS-A** tool, so we can model this as:

```
class Tool {  
    public int size;  
}  
  
class Hammer extends Tool {  
}
```

As `size` is a `public` attribute, it's inherited by `Hammer`:

```
Hammer hammer = new Hammer();  
hammer.size = 10;
```

Chapter TWO

From the previous chapter, we know that only private and members with default visibility when the subclass is defined in a different package than the superclass, are not inherited.

An attribute or method is inherited with the same visibility level as the one defined in the superclass. However, in the case of methods, you can change them to be more visible, but you cannot make them less visible:

```
class Tool {  
    public int size;  
    public int getSize() { return size; }  
}  
  
class Hammer extends Tool {  
    private int size; // No problem!  
    // Compile-time error  
    private int getSize() { return size; }  
}
```

There's no problem for attribute because we're creating a **NEW** attribute in Hammer that **HIDES** the one inherited from Tool when the name is the same.

Here are the things you can do in a subclass:

- Inherited attributes can be used directly, just like any other.
- An attribute can be declared in the subclass with the same name as the one in the superclass, thus hiding it.
- New attributes that are not in the superclass can be declared in the subclass.
- Inherited methods can be directly used as they are.

- A new instance method can be declared in the subclass that has the same signature as the one in the superclass, thus overriding it.
- A new static method can be declared in the subclass that has the same signature as the one in the superclass, thus hiding it.
- New methods that are not in the superclass can be declared in the subclass.
- A constructor can be declared in the subclass that invokes the constructor of the superclass, either implicitly or by using the keyword super.

So for methods, reducing their visibility is not allowed because they are handled in a different way, in other words, methods are either overridden or overloaded.

Besides, think about it. Because of encapsulation, attributes are supposed to be hidden, but with methods, if a subclass doesn't have a method of the superclass, the subclass cannot be used wherever the superclass is used. This is called the *Liskov substitution principle*, which is important in polymorphism, and we'll review after talking about overridden and overloaded methods.

Implementing an interface is in some ways is a type of inheritance because they have some common characteristics, but by doing it, the relationship becomes **HAS-A**. We'll talk more about them in Chapter 4.

OVERLOADING AND OVERRIDING

The difference between overloading and overriding has to do a lot with method signatures.

In a few words, the *method signature* is the name of the method and the list of its parameters (types and number of parameters included). Note that return types are not included in this definition.

We talk about overloading when a method changes method signature, by changing the list of parameters of another method (that might be inherited) while keeping the same name.

```
class Hotel {  
    public void reserveRoom(int rooms) { ... }  
}  
  
class ThreeStarHotel extends Hotel {  
    // Method overload #1  
    public void reserveRoom(List<Room> rooms)  
    {  
        ...  
    }  
    // Method overload #2  
    public void reserveRoom(int rooms, int numberPeople)  
    {  
        ...  
    }  
}
```

Changing just the return type will generate a compile error:

```
class ThreeStarHotel extends Hotel {  
    // Compile-time error, reserveRoom is seen as duplicated  
    public void reserveRoom(List<Room> rooms) { ... }  
    public boolean reserveRoom(List<Room> rooms) { ... }  
  
}
```

Exceptions in the throws clause are not considered when overloading, so again, changing just the exception list will throw a compile error:

```
class ThreeStarHotel extends Hotel {  
    // Compile-time error, reserveRoom is seen as duplicated  
    public void reserveRoom(List<Room> rooms)  
        throws RuntimeException {  
        ...  
    }  
    public boolean reserveRoom(List<Room> rooms)  
        throws NullPointerException {  
        ...  
    }  
}
```

When an overloaded method is called, the compiler has to decide which version of the method is going to call. The first obvious candidate is to call the method that exactly matches the number and types of the arguments. But what happens when there isn't an exact match?

Chapter TWO

The rule to remember is that Java will look for the **CLOSEST** match **FIRST** (this means a larger type, a superclass, an autoboxed type, or the **MORE** particular type).

For example, when this class is executed:

```
class Print {
    static void printType(short param) {
        System.out.println("short");
    }
    static void printType(long param) {
        System.out.println("long");
    }
    static void printType(Integer param) {
        System.out.println("Integer");
    }
    static void printType(CharSequence param) {
        System.out.println("CharSequence");
    }

    public static void main(String[] args) {
        byte b = 1;
        int i = 1;
        Integer integer = 1;
        String s = "1";

        printType(b);
        printType(i);
        printType(integer);
        printType(s);
    }
}
```

The output is:

```
short  
long  
Integer  
CharSequence
```

In the first method call, the argument type is `byte`. There's no method taking a `byte`, so the closest larger type is `short`.

In the second method call, the argument type is `int`. There's no method taking an `int`, so the closest larger type is `long` (note that this has higher precedence than `Integer`).

In the third method call, the argument type is `Integer`. There's a method that takes an `Integer`, so this is called.

In the last method call, the argument type is `String`. There's no method taking a `String`, so the closest superclass is `CharSequence`.

If it can't find a match or if the compiler cannot decide because the call is ambiguous, a compile error is thrown. For example, considering the previous class, the following will cause an error because there isn't a larger type than `double` and it can't be autoboxed to an `Integer`:

```
// Can't find a match  
double d = 1.0;  
printType(d);
```

Chapter TWO

The following is an example of an ambiguous call, assuming the methods:

```
static void printType(float param, double param2) {  
    System.out.println("float-double");  
}  
static void printType(double param, float param2) {  
    System.out.println("double-float");  
}  
  
...  
  
// Ambiguous call  
printType(1,1);
```

Constructors of a class can also be overloaded. In fact, you can call one constructor from another with the `this` keyword:

```
class Print {  
    Print() {  
        this("Calling with default argument");  
    }  
    Print(String s) {  
        System.out.println(s);  
    }  
}
```

We talk about overriding when the method signature is the same, but for some reason, we want to redefine an **INSTANCE** method in the subclass.

```
class Hotel {  
    public void reserveRoom(int rooms) { ... }  
}  
  
class ThreeStarHotel extends Hotel {  
    // Method override  
    public void reserveRoom(int rooms) {  
        ...  
    }  
}
```

If a static method with the same signature as a static method in the superclass is defined in the subclass, then the method is **HIDDEN** instead of overridden.

There are some rules when overriding a method.

The access modifier must be the same or with more visibility:

```
class Hotel {  
    public void reserveRoom(int rooms) { ... }  
}  
  
class ThreeStarHotel extends Hotel {  
    // Compile-time error  
    protected void reserveRoom(int rooms) {  
        ...  
    }  
}
```

Chapter TWO

The return type must be the same or a subtype:

```
class Hotel {  
    public Integer reserveRoom(int rooms) { ... }  
}  
  
class ThreeStarHotel extends Hotel {  
    // Compile-time error  
    public Number reserveRoom(int rooms) {  
        ...  
    }  
}
```

Exceptions in the throws clause must be the same, less, or subclasses of those exceptions:

```
class Hotel {  
    public void reserveRoom(int rooms) throws IOException {  
        ...  
    }  
}  
  
class ThreeStarHotel extends Hotel {  
    // Compile-time error  
    public void reserveRoom(int rooms) throws Exception {  
        ...  
    }  
}
```

Overriding is a critical concept in polymorphism, but before touching this topic, let's see some important methods from java.lang.Object that most of the time we'll need to override.

OBJECT CLASS METHODS

In Java, all objects inherit from `java.lang.Object`.

This class has the following methods that can be overridden (redefined):

- `protected Object clone() throws CloneNotSupportedException`
- `protected void finalize() throws Throwable`
- `public int hashCode()`
- `public boolean equals(Object obj)`
- `public String toString()`

The most significant methods, the ones you almost always would want to redefine, are `hashCode`, `equals`, and `toString`.

public int hashCode()

It returns a hash code value for the object. The returned value must have the following contract:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are not equal according to the `equals(java.lang.Object)` method, then calling the

Chapter TWO

hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

public boolean equals(Object obj)

Indicates whether another object is equal to the object that calls the method. It's necessary to override the hashCode method whenever this method is overridden since the contract for the hashCode method states that equal objects must have equal hash codes. This method is:

- Reflexive: for any non-null reference value x, x.equals(x) should return true.
- 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.
- 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.
- Consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or 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.

public String toString()

It returns a string representation of the object. The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object.

To override these methods just follow the general rules for overriding:

- The access modifier must be the same or more accessible
- The return type must be either the same or a subclass
- The name must be the same
- The argument list types must be the same
- The same exceptions or their subclasses are allowed to be thrown

In a few words, define the method just as it appears in the `java.lang.Object` class.

POLYMORPHISM

Polymorphism is the ability for an object to vary its behavior based on its type. This is best demonstrated with an example:

```
class HumanBeing {  
    public void dress() {  
        System.out.println("Dressing a human being");  
    }  
}  
  
class Man extends HumanBeing {  
    public void dress() {  
        System.out.println("Put on a shirt");  
        System.out.println("Put on some jeans");  
    }  
}  
  
class Woman extends HumanBeing {  
    public void dress() {  
        System.out.println("Put on a dress");  
    }  
}  
  
class Baby extends HumanBeing {  
    public void dress() {  
        System.out.println(  
            "I don't know how to dress!");  
    }  
}
```

And now let's create some human beings to see polymorphism in action:

```
HumanBeing[] someHumans = new HumanBeing[3];
someHumans[0] = new Man();
someHumans[1] = new Woman();
someHumans[2] = new Baby();

for(int i = 0; i < someHumans.length; i++) {
    someHumans[i].dress();
    System.out.println();
}
```

The output:

```
Put on a shirt
Put on some jeans
Put on a dress
I don't know how to dress!
```

Even though `HumanBeing` is used, the JVM decides at runtime which method to call based on the type of the object assigned, not the variable's reference type.

This is called *virtual method invocation*, a fancy name for overriding.

Overriding is also known as *dynamic polymorphism* because the type of the object is decided at **RUN** time.

In contrast, overloading is also called *static polymorphism* because it's resolved at **COMPILE** time.

ABSTRACT CLASSES AND METHODS

If we examine the previous example, I think we'll agree that the implementation of the `dress()` method in the class `HumanBeing` doesn't sound exactly right.

Most of the time, we'll be working with something more concrete, like a `Man` or a `Woman` so there's no need to instantiate the `HumanBeing` class directly, however, a common abstraction of those classes may be useful. Using an abstract class (or method) is the best option to model these cases.

Abstract classes **CANNOT** be instantiated, only subclassed. They are declared with the `abstract` keyword:

```
abstract class AClass { }
```

Abstract methods are declared **WITHOUT** an implementation (body), like this:

```
abstract void AMethod();
```

So in the previous example, it's better to model the whole `HumanBeing` class as abstract so no one can use directly:

```
abstract class HumanBeing {
    public abstract void dress();
}
```

Now, the following would cause a compile error:

```
HumanBeing human = new HumanBeing();
```

And it makes sense; there can't be no guarantees that an abstract class will have all its methods implemented. Calling an unimplemented method would be an epic fail.

Here are the rules when working with abstract methods and classes:

The `abstract` keyword can only be applied to classes or non-static methods.

```
abstract class AClass {  
    public static abstract void AMethod(); // Compile-time error  
}
```

An abstract class doesn't need to declare abstract methods to be declared abstract.

```
abstract class AClass {} // No problem
```

If a class includes abstract methods, then the class itself must be declared abstract.

```
class AClass { // Compile-time error  
    public abstract void AMethod();  
}
```

If the subclass of an abstract class doesn't provide an implementation for all abstract methods, the subclass must also be declared abstract.

```
class Man extends HumanBeing {} // Compile-time error
```

Methods of an interface are considered abstract, so an abstract class that implements an interface can implement some or none of the interface methods.

```
abstract class AClass implements Runnable {} // No problem
```

KEY POINTS

- Inheritance refers to an **IS-A** relationship where a class (called superclass) provides common attributes and methods to derived or more specialized classes (called subclass).
- Here are the things you can do in a subclass:
 - Inherited attributes can be used directly, just like any other.
 - An attribute can be declared in the subclass with the same name as the one in the superclass, thus hiding it.
 - New attributes that are not in the superclass can be declared in the subclass.
 - Inherited methods can be used directly as they are.
 - A new instance method can be declared in the subclass that has the same signature as the one in the superclass, thus overriding it.
 - A new static method can be declared in the subclass that has the same signature as the one in the superclass, thus hiding it.
 - New methods that are not in the superclass can be declared in the subclass.
 - A constructor can be declared in the subclass that invokes the constructor of the superclass, either implicitly or by using the keyword super.
- The method signature is the name of the method and the list of its parameters (types and number of parameters included). Return types are not included in this definition.
- We talk about overloading when a method changes the list of parameters of another method (that might be inherited) while

keeping the same name.

- We talk about overriding when the method signature is the same, but for some reason, we want to redefine an **INSTANCE** method in the subclass.
- The most important methods of `java.lang.Object` that most classes must redefine are:
 - `public int hashCode()`
 - `public boolean equals(Object obj)`
 - `public String toString()`
- With polymorphism, subclasses can define their own behaviors (different than the ones of the methods of the superclass), and the JVM will call the appropriate method for the object. This behavior is referred to as *virtual method invocation*.
- Abstract classes **CANNOT** be instantiated, only subclassed. Abstract methods are declared **WITHOUT** an implementation (body).
- The `abstract` keyword can only be applied to classes or non-static methods.
- An abstract class doesn't need to declare abstract methods to be declared abstract.
- If a class includes abstract methods, then the class itself must be declared abstract.
- If the subclass of an abstract class doesn't provide an implementation for all abstract methods, the subclass must also be declared abstract.
- Methods of an interface are considered abstract, so an abstract class that implements an interface can implement some or none of the interface methods.

SELF TEST

1. Given:

```
public class Question_2_1 {  
    protected int id;  
    protected String name;  
  
    protected boolean equals(Question_2_1 q) {  
        return this.name.equals(q.name);  
    }  
  
    public static void main(String[] args) {  
        Question_2_1 q1 = new Question_2_1();  
        Question_2_1 q2 = new Question_2_1();  
        q1.name = "q1";  
        q2.name = "q1";  
  
        if(q1.equals((Object)q2)) {  
            System.out.println("true");  
        } else {  
            System.out.println("false");  
        }  
    }  
}
```

What is the result?

- A. true
- B. false
- C. Compilation fails
- D. An exception occurs at runtime

2. Which of the following is a method of `java.lang.Object` that can be overridden?

- A. `public String toString(Object obj)`
- B. `public int equals(Object obj)`
- C. `public int hashCode(Object obj)`
- D. `public int hashCode()`

3. Given:

```
public class Question_2_3 {  
    public static void print(Integer i) {  
        System.out.println("Integer");  
    }  
    public static void print(Object o) {  
        System.out.println("Object");  
    }  
    public static void main(String[] args) {  
        print(null);  
    }  
}
```

What is the result?

- A. Integer
- B. Object
- C. Compilation fails
- D. An exception occurs at runtime

Chapter TWO

4. Given:

```
class SuperClass {  
    public static void print() {  
        System.out.println("Superclass");  
    }  
}  
  
public class Question_2_4 extends SuperClass {  
    public static void print() {  
        System.out.println("Subclass");  
    }  
    public static void main(String[] args) {  
        print();  
    }  
}
```

What is the result?

- A. Superclass
- B. Subclass
- C. Compilation fails
- D. An exception occurs at runtime

5. Given:

```
abstract class SuperClass2 {  
    public static void print() {  
        System.out.println("Superclass");  
    }  
}  
  
class SubClass extends SuperClass2 {}  
  
public class Question_2_5 {  
    public static void main(String[] args) {  
        SubClass subclass = new SubClass();  
        subclass.print();  
    }  
}
```

What is the result?

- A. Superclass
- B. Compilation fails because an abstract class cannot have static methods
- C. Compilation fails because Subclass doesn't implement method print()
- D. Compilation fails because Subclass doesn't have a method print()
- E. An exception occurs at runtime

6. Given:

```
abstract class SuperClass3 {  
    public void print() {  
        System.out.println("Superclass");  
    }  
}  
  
public class Question_2_6 extends SuperClass3 {  
    public void print() {  
        System.out.println("Subclass");  
    }  
    public static void main(String[] args) {  
        Question_2_6 q = new Question_2_6();  
        ((SuperClass3)q).print();  
    }  
}
```

What is the result?

- A. Superclass
- B. Subclass
- C. Compilation fails
- D. An exception occurs at runtime

ANSWERS

1. The correct answer is B.

The method:

```
protected boolean equals(Question_1_3 q)
```

Doesn't override the equals method from `java.lang.Object` (look at the access modifier). For that reason, the if condition fails. Note that if the method call were:

```
if(q1.equals(q2)) { ... }
```

We will be calling the method with the `protected` modifier, making the condition true.

2. The correct answer is D.

The proper signatures of the methods are:

- `public int hashCode()`
- `public boolean equals(Object obj)`
- `public String toString()`

So the only correct answer is option D.

3. The correct answer is A.

You could say that the overload is ambiguous, and compilation fails, since `null` can be assigned to any type, but in fact, there's a rule that says the in overloading the most **SPECIFIC** one is chosen. Since `Integer` is more specific than `Object`, the former is selected.

4. The correct answer is B.

Static methods are hidden not overridden. Since the main method is in the subclass, the subclass method is called.

The call to Question_2_4.print() yields the same result:

Subclass

The call to SuperClass.print() yields:

Superclass

5. The correct answer is A.

Static methods are allowed in abstract classes, the only restriction being that they cannot be abstract. All methods that are accessible are inherited by subclasses. If an inherited method is static, the only difference is that the method can be hidden instead of overridden.

6. The correct answer is B.

Even when the variable is cast to SuperClass3, Java chooses to execute the method of the subclass thanks to polymorphism. If we want to execute the method of the superclass, we have to instantiate an object from that class.

Chapter TWO

Chapter THREE

Inner Classes

Exam Objectives

- Create inner classes including static inner class, local class, nested class, and anonymous inner class.

CLASSES

In Java we have classes:

```
class Computer {  
  
}
```

And classes have two types of members, attributes (or fields) and methods (or functions):

```
class Computer {  
    int serialNumber;  
  
    void executeCommand() {  
        // Do something  
    }  
}
```

This way, our programs are a collection of classes.

```
class Computer {  
    // Here goes the definition of the class  
}  
class Desktop {  
    // Here goes the definition of the class  
}  
class Printer {  
    // Here goes the definition of the class  
}  
class Programmer {  
    // Here goes the definition of the class  
}
```

INNER CLASSES

Java gives us flexibility in the way we can design our classes.

For this, there's a third type of member a class can have, an **INNER CLASS**.

```
class Computer {  
    String serialNumber;  
  
    void executeCommand() {}  
  
    class Processor {  
        // Here goes the definition of the class  
    }  
}
```

Inner classes are also known as nested classes. In theory, you could have many levels of classes.

I have a hard time thinking about what would be the benefit of having more than one level of inner classes.

```
class LevelOne {  
    class LevelTwo {  
        class LevelThree {  
            class LevelFour {  
                /** Finally do something */  
            }  
        }  
    }  
}
```

Another thing. We always talk about inner **CLASSES**, but actually, we can have inner **ABSTRACT CLASSES**, inner **INTERFACES**, and inner **ENUMS**.

```
class Computer {  
    abstract class Processor { }  
  
    interface Pluggable { }  
  
    enum PORTS {  
        USB2, USB3, ESATA, HDMI  
    }  
}
```

But we are going to focus on simple inner classes. There are four types of them:

- **STATIC** inner classes
- **NON-STATIC** inner classes
- **LOCAL** classes
- **ANONYMOUS** classes

DEFINING A STATIC INNER CLASS

```
class Computer {  
    static class Mouse {  
    }  
}
```

USING A STATIC INNER CLASS

STATIC INNER CLASSES ARE
ACCESSED THROUGH THEIR
ENCLOSING CLASS

```
Computer.Mouse m =  
new Computer.Mouse();
```

Chapter THREE

Static classes are **INDEPENDENT** of their enclosing class. They are like ordinary classes, only that they just happen to be inside another class.

In fact, you can think of the enclosing class as a kind of a package. You can import the name of the enclosing class and use the static inner class like a normal class. Just remember that the inner static class must be a public member so that it can be accessed from another package.

```
import com.example.Computer.*;  
  
public class Test {  
    Mouse m = new Mouse();  
  
    /** Rest of the definition */  
}
```

And they can also be marked as private, protected or without a modifier, so they are accessible only in the package (default accessibility).

```
public class Computer {  
    private static class Component {}  
    protected static class MotherBoard {}  
    static class Slot {}  
}
```

Of course, by being a member of a class, the static inner class have access to the other members of the enclosing class, but only if they are **STATIC**.

```
public class Computer {  
    private static String serialNumber = "1234X";  
    public static class Mouse {  
        void printSN() {  
            System.out.println("MOUSE-" + serialNumber);  
        }  
    }  
}
```

Why?

Think about it, if the static class is independent of its enclosing class, it doesn't need an instance of this, so only the static members could be used because they are associated with the class, not to a particular instance.

For that reason, a static inner class is often used as a utility class that contains common methods shared by all the objects of a class.

If you use the static inner class inside the class that defines it, you can use it in any method, block or constructor, no matter if it's static or not, since the inner class is not tied to a particular instance.

DEFINING A NON-STATIC INNER CLASS

```
class Computer {  
    class HardDrive {  
    }  
}
```

USING A NON-STATIC INNER CLASS

NON-STATIC INNER CLASSES ARE ACCESSED THROUGH AN INSTANCE OF THEIR ENCLOSING CLASS

```
Computer c =  
    new Computer();  
Computer.HardDrive hd =  
    c.new HardDrive();
```

CHECK OUT THE TYPE OF THE INNER CLASS
AND HOW THE NEW OPERATOR IS USED

Chapter THREE

Non-static inner classes are just called inner classes.

Instances of an inner class only exist **WITHIN** an instance of the enclosing class. It's the same that when you want to use a method of a class, you **FIRST** need an instance of that class.

Once you have an instance of the enclosing class, you use the new operator in a (weird) different way than you typically use it.

```
Computer computer = new Computer();
Computer.HardDrive hardDrive = computer.new HardDrive();
```

You can also use the import trick to writing less, but you still need to create the inner class as always.

```
import com.example.Computer.*;

public class Test {
    Computer computer = new Computer();
    HardDrive hd = computer.new HardDrive();

    /** Rest of the definition */
}
```

Another way to get an instance of an inner class is to use a method of the enclosing class to create it, avoiding that weird syntax.

```
public class Computer {  
    class HardDrive { }  
    public HardDrive getHardDrive() {  
        return new HardDrive();  
    }  
}
```

By being a member of a class, the inner class has access to the other members of the enclosing class, but this time, it **DOESN'T** matter if they are static or not.

```
public class Computer {  
    private String brand = "XXX";  
    private static String serialNumber = "1234X";  
    public class HardDrive {  
        void printSN() {  
            System.out.println(  
                brand + "-MOUSE-" + serialNumber  
            );  
        }  
    }  
}
```

Why?

Because to use the inner class, an instance of the enclosing class is required, ensuring that the non-static members exist (static members can be accessed anyway).

Chapter THREE

Inner classes can also be marked as private, protected or without a modifier, so they are accessible only in the package. But most of the time, since they depend on the enclosing class, they are marked as private.

```
public class Computer {  
    private class Component {}  
    protected class MotherBoard {}  
    class Slot {}  
}
```

Another rule is that inner classes **CANNOT** contain static members.

```
public class Computer {  
    class HardDrive {  
        static int capacity; // Compile-time error here  
        static void printInfo() { // Compile-time error here  
            // Definition goes here  
        }  
    }  
}
```

Why?

Two reasons.

Static code is executed during class initialization, but you cannot initialize a non-static inner class without having an instance of the enclosing class.

Because an inner class belongs to **ONE** instance of the enclosing class. Having a static member means it can be shared across instances because the member belongs to the class, but since we are talking about an inner class that cannot be shared by many instances of the enclosing class, that is not possible.

The only exception is when you define a final static attribute. The final keyword makes all the difference; it makes a constant expression, but it only works with **ATTRIBUTES** and when assigning an **NON-NULL** value.

```
public class Computer {  
    class HardDrive {  
        final static int capacity = 120; // It does compile!  
        final static String brand = null; // Compile-time error here  
        final static void printInfo() { // Compile-time error here  
            // Definition goes here  
        }  
    }  
}
```

DEFINING A LOCAL CLASS

```
class Computer {  
    void process() {  
        class Processor {  
            }  
        }  
    }
```

USING A LOCAL CLASS

LOCAL CLASSES CAN ONLY BE USED
INSIDE THE METHOD OR BLOCK
THAT DEFINES THEM

```
void process() {  
    class Core {}  
    Core core = new Core();  
}
```

Chapter THREE

Local classes are local because they can only be used in the method or block where they are declared. Blocks are practically anything between curly braces.

```
void method() {  
    class MethodLocalClass { }  
    MethodLocalClass mlc = new MethodLocalClass();  
    if ( 1 == 1 ) {  
        class IfLocalClass { }  
        IfLocalClass ilc = new IfLocalClass();  
    }  
    while ( true ) {  
        class WhileLocalClass { }  
        WhileLocalClass wlc = new WhileLocalClass();  
    }  
}
```

Also, notice where the instances of the local classes are created. The local class has to be used **BELLOW** its definition. Otherwise, the compiler won't be able to find it.

Because a local inner class is not a member of a class, it **CANNOT** be declared with an access level, and it wouldn't make sense anyway since they are only accessible where they are declared. However, a local class can be declared as abstract or final (but not at the same time).

Local classes require an instance of their enclosing class so the method or block in which they are defined can be executed. For this reason, they can access the members of the enclosing class, but they cannot declare static members (only static final attributes), just like inner classes.

```
class Computer {  
    private String serialNumber = "1234XX";  
  
    void process() {  
        class Processor {  
            Processor() {  
                System.out.println(  
                    "Processor #1 of computer " + serialNumber  
                );  
            }  
        }  
    }  
}
```

If the local class is declared inside a method, it can access the variables and parameters of the method **ONLY** if they are declared `final` or are *effectively final*.

Effectively final is a term that means that a variable or parameter is not changed after it's initialized, its declaration does not use the `final` keyword.

Why?

Because an instance of a local class can be alive even after the method or block in which is defined has finished its execution (for example, if a reference is saved in an object with greater scope). For this reason, the local class must keep an internal copy of the variables it uses, and the only way to ensure that both copies always hold the same value it's by making the variable `final`.

Chapter THREE

So, the following code is valid because `taskName` is declared `final` while `n` doesn't change and it's considered *effectively final*.

```
void process(int n) {  
    final String taskName = "Task #1";  
    class Processor {  
        Processor() {  
            System.out.println(  
                "Processor " + n + " processing " + taskName  
            );  
        }  
    }  
}
```

But if we modify the value of `n` somewhere, an error will be generated.

```
void process(int n) {  
    final String taskName = "Task #1";  
    class Processor {  
        Processor() {  
            System.out.println(  
                "Processor " + n +      // Compile-time error  
                " processing " + taskName  
            );  
        }  
    }  
    n = 4;  
}
```

Effectively final is only concerned with references, not objects or their content, because at the end of the day, we are referencing the same object.

```
void process(int n) {  
    StringBuffer taskName = new StringBuffer("Task #1");  
    class Processor {  
        Processor() {  
            System.out.println(  
                "Processor " + n +  
                " processing " + taskName // It does compile!  
            );  
        }  
    }  
    taskName.append("1"); // This is valid!  
    //Uncommenting the following line will generate an error  
    //taskName = new StringBuffer("Task #2");  
}
```

If you're still not sure about a declaration being *effectively final*, try adding the `final` modifier to it. If the program continues to behave in the same way, then the declaration is *effectively final*.

If the class is declared in a static method, static rules also apply, meaning that the local class only has access to the static members of the enclosing class.

DEFINING AN ANONYMOUS CLASS

```
Computer comp = new Computer() {  
    void process() {  
        // Here goes the definition  
    }  
};
```

LOOK HOW IT ENDS WITH A
SEMICOLON, LIKE ANY OTHER
STATEMENT IN JAVA

THE NEW OPERATOR IS FOLLOWED BY THE NAME OF AN INTERFACE OR A CLASS AND THE ARGUMENTS TO A CONSTRUCTOR (OR EMPTY PARENTHESES IF IT'S AN INTERFACE)

THE BODY OF THE CLASS IMPLEMENTS
THE INTERFACE OR EXTENDS THE CLASS
REFERENCED

Chapter THREE

An anonymous class is called that way because it doesn't have a name. However, an anonymous class expression doesn't declare a new class. It either **IMPLEMENTS** an existing interface or **EXTENDS** an existing class. So

```
new Computer() { }
```

is like writing

```
class [NO_NAME_CLASS] extends Computer { }
```

And if we're working with an interface

```
new Runnable() { }
```

is like writing

```
class [NO_NAME_CLASS] implements Runnable { }
```

Also, an anonymous class can be used in a declaration or a method call.

```
class Program {
    void start(Computer c) {
        // Definition goes here
    }
    public static void main(String args[]) {
        Program program = new Program();
        program.start(new Computer() {
            void process() { /** Redefinition goes here */ }
        });
    }
}
```

Since they don't have a name (well, actually the compiler gives them a random name when it creates the .class file), anonymous classes can't have **CONSTRUCTORS**. If you want to run some initializing code, you have to do it with an initializer block.

```
Computer t = new Computer() {  
    {  
        // Initializing code  
    }  
    void process() { /** Redefinition goes here */ }  
};
```

Because anonymous classes are a type of local classes, they have the same rules:

- They can access the members of their enclosing class
- They cannot declare static members (only if they are final static variables)
- They can only access local variables (variables or parameters defined in a method) if they are final or *effectively final*.

But one thing you have to be careful with, is inheritance.

Chapter THREE

When you use an anonymous class (a subclass object), you're using a superclass reference. With this reference, you can use the attributes and methods declared in that type.

But what happens when you declare a new method on the anonymous class?

```
class Task {  
    void doIt() {  
        /** Here goes the definition */  
    }  
}  
  
class Launcher {  
    public static void main(String args[]) {  
        Task task = new Task() {  
            void redoIt() {  
                /** Here goes the definition */  
            }  
        };  
  
        task.doIt();      // It's OK  
        task.redoIt();   // Compile-time error!  
    }  
}
```

The program will fail. The reference doesn't know about the method `redoIt()` because it's not defined in the superclass.

Typically, you would cast the type to the subclass where the new method is defined.

```
SubClass object = (SubClass) objectWithSuperClassReference;  
object.methodOnlyDefinedInTheSubclass();
```

But with an anonymous class, how does the cast is done?

It can't be done; the class has no name, so there is no way we can use the methods defined in the declaration of the anonymous class. We can only use the methods declared in the **SUPERCLASS** (be it an interface or class).

Using an anonymous class is mostly about style. If the class has a short body, it only implements one interface (if we're working with interfaces), it doesn't declare new members, and the syntax makes your code clearer, you should consider using it instead of a local or an inner class.

SHADOWING

An important concept to take into account when working with inner classes (of any type) is what happens when a member of the inner class has the same name of a member of the enclosing class.

```
class Computer {  
    private String serialNumber = "1234XXX";  
  
    class HardDrive {  
        private String serialNumber = "1234DDD";  
  
        void printSN(String serialNumber) {  
            System.out.println("SN: " + serialNumber);  
        }  
    }  
}
```

In this case, the parameter `serialNumber` shadows the instance variable `serialNumber` of `HardDrive` that in turn, shadows the `serialNumber` of `Computer`.

As it's coded, the method `printSN()` will print its argument. A shadowed declaration needs something else to be properly referred.

We know that when an object wants to refer to itself, we need to use the keyword `this`.

So, if we use `this` inside an inner class, it will refer to the inner class itself.

If we need to reference the enclosing class, inside the inner class we can also use `this`, but in this way `NameOfTheEnclosingClass.this`.

```
class Computer {  
    private String serialNumber = "1234XXX";  
  
    class HardDrive {  
        private String serialNumber = "1234DDD";  
  
        void printSN(String serialNumber) {  
            System.out.println(  
                "SN: " + serialNumber  
            );  
            System.out.println(  
                "HardDrive SN: " + this.serialNumber  
            );  
            System.out.println(  
                "Computer SN: " + Computer.this.serialNumber  
            );  
        }  
    }  
}
```

Since it may cause confusion, it's better to avoid it and use descriptive variable names.

KEY POINTS

- Inner classes are declared inside another class. There are four types of them: static, non-static, local and anonymous classes.
- Static classes are just inner classes marked with the `static` keyword. However, they behave more like a top-level class than an inner class.
- You don't need an instance of the enclosing class to instantiate a static class:

```
EnclosingClass.StaticClass sc =  
    new EnclosingClass.StaticClass();
```

- A static class cannot access non-static members of the enclosing class since it doesn't require having an instance of the enclosing class to use it.
- A (non-static) inner class is like any other member of the enclosing class so that it can be marked with any access modifier.
- Outside the enclosing class' instance methods or blocks, to instantiate an inner class, you must first create an instance of the enclosing class and then:

```
EnclosingClass ec = new EnclosingClass();  
EnclosingClass.InnerClass ic = ec.new InnerClass();
```

- A local class is defined within a method or block of the enclosing class.
- The only modifiers that apply to a local class are abstract

and final (but not at the same time).

- You can only use a local class in the method or block where you define it, and only after its declaration.
- A local class can access the members of a class just like any other member of the class (static rules still apply).
- However, a local class can only access the parameters and local variables of a method if they are final or *effectively final*.
- *Effectively final* means that a variable cannot be modified after its initialization, even if it's not explicitly marked as final.
- Anonymous classes have no name, and they either extend an existing class or implement an interface:

```
ExistingClass ac = new ExistingClass() {  
    // Definition goes here  
};
```

- Anonymous classes cannot have constructors.
- Anonymous classes have the same rules as local classes regarding accessing members of the enclosing class and local variables of a method.
- The only methods you can call on an anonymous class are those defined in the reference type (the superclass or the interface), even though anonymous classes can define their own methods.

SELF TEST

1. Given:

```
public class Question_3_1 {  
    interface ITest { // 1  
        void m();  
    }  
    public static void main(String args[]) {  
        ITest t = new ITest() { // 2  
            public void m() {  
                System.out.println("m()");  
            }  
        }  
        t.m();  
    }  
}
```

What is the result?

- A. m()
- B. Compilation fails on the declaration marked as // 1
- C. Compilation fails on the declaration marked as // 2
- D. An exception occurs at runtime

2. Given:

```
public class Question_3_2 {  
    public static void main(String args[]) {  
        Question_3_2 q = new Question_3_2();  
        int i = 2;  
        q.method(i);  
        i = 4;  
    }  
  
    void method(int i) {  
        class A {  
            void helper() {  
                System.out.println(i);  
            }  
        }  
        new A().helper();  
    }  
}
```

What is the result?

- A. Compilation fails
- B. 2
- C. 4
- D. An exception occurs at runtime

Chapter THREE

3. Given:

```
public class Question_3_3 {  
    public static void main(String[] args) {  
        Question_3_3 q = new Question_3_3() {  
            public int sum(int a, int b) { // 1  
                return a + b;  
            }  
        };  
        q.sum(2,6); // 2  
    }  
}
```

What is the result?

- A. Compilation fails on the declaration marked as // 1
- B. Compilation fails on the line marked as // 2
- C. 8
- D. Nothing is printed

4. Given:

```
public class Question_3_4 {  
    public static class Inner {  
        private void doIt() {  
            System.out.println("doIt()");  
        }  
    }  
  
    public static void main(String[] args) {  
        Question_3_4.Inner i = new Inner();  
        i.doIt();  
    }  
}
```

What is the result?

- A. Compilation fails because an inner class cannot be static.
- B. Compilation fails because the Inner class is instantiated incorrectly inside method main.
- C. Compilation fails because the method doIt cannot be called in main because it is declared as private
- D. The program prints doIt()

Chapter THREE

5. Given:

```
class A {  
    class B {  
        class C {  
            void go() {  
                System.out.println("go!");  
            }  
        }  
    }  
}  
  
public class Question_3_5 {  
    public static void main(String[] args) {  
        A a = new A();  
        A.B b = a.new B(); // 1  
        B.C c = b.new C(); // 2  
        c.go(); // 3  
    }  
}
```

What is the result?

- A. Compilation first fails on the line // 1
- B. Compilation first fails on the line // 2
- C. Compilation fails on the line // 3
- D. go! is printed

6. Given:

```
public class Question_3_6 {  
    private class A { // 1  
        public int plusTwo(int n) {  
            return n + 2;  
        }  
    }  
  
    public static void main(String[] args) {  
        Question_3_6.A a = new A(); // 2  
        System.out.println(a.plusTwo(3));  
    }  
}
```

What is the result?

- A. Compilation fails on the line // 1
- B. Compilation fails on the line // 2
- C. 5
- D. An exception occurs at runtime

Chapter THREE

7. Given:

```
public class Question_3_7 {  
    public static void main(String[] args) {  
        abstract class A { // 1  
            public void m() {  
                System.out.println("m()");  
            }  
        }  
        public class AA extends A { } // 2  
    }  
}
```

What change would make this code compile?

- A. Remove the `abstract` keyword on the line // 1
- B. Add the `public` keyword on the line // 1
- C. Remove the `public` keyword on the line // 2
- D. None. This code compiles correctly

8. Given:

```
public class Question_3_8 {  
    int i = 2;  
    interface A {  
        int add();  
    }  
    public A create(int i) {  
        return new A() {  
            public int add() {  
                return i + 4;  
            }  
        };  
    }  
    public static void main(String[] args) {  
        A a = new Question_3_8().create(8);  
        System.out.println(a.add());  
    }  
}
```

What is the result?

- A. 6
- B. 12
- C. Compilation fails
- D. An exception occurs at runtime

ANSWERS

1. The correct answer is C.

There's nothing wrong with the `ITest` interface; you can declare inner interfaces without problems. What is wrong is that the declaration of the anonymous class is missing the closing semicolon.

Remember, the declaration of an anonymous class is a Java expression, like declaring an `int` variable, so it has to end with a semicolon after the curly brace.

2. The correct answer is B.

Local classes (the ones defined inside a method) and anonymous classes (the ones defined without a name) can only access variables of the enclosing context that are final or effectively final (which means that a variable cannot be modified after its initialization).

The variable `i` is modified after the method call so that it won't qualify as an effectively final variable. However, this is not the variable used inside `method()`, since in Java, all parameters are passed by value. As the parameter is not modified inside the method, it is effectively final and can be used inside local class A.

3. The correct answer is B.

The method `main` defines an anonymous class that is a subclass

of type Question_3_3. Since Question_3_3 doesn't define any methods, nothing is overridden and, although the method sum is defined inside the anonymous class, no methods apart from the ones inherited from Object can be called.

4. The correct answer is D.

There's nothing wrong with the static inner class declaration or the way it is instantiated. Since all happens in the same class, the main method can access the private members of the inner class, so there's no problem with that either.

5. The correct answer is A.

This program first fails on the line marked as // 1 because when instantiating an inner class from outside its enclosing class, you don't use its compound name (in this case A.B) on the right side of the statement. So instead of:

```
A.B b = a.new A.B();
```

It's just:

```
A.B b = a.new B();
```

If you want to call method go(), the correct code is:

```
A a = new A();
A.B bb = a.new B();
A.B.C cc = bb.new C();
cc.go();
```

6. The correct answer is B.

The inner class A can be marked as private and method main can access it because both are members of Question_3_6 class.

However, since main is a static method, nothing guarantees that there will be an instance of Question_3_6, so you must explicitly create one to instantiate the inner class A. So instead of:

```
Question_3_6.A a = new A();
```

You have to use:

```
Question_3_6 q = new Question_3_6();
Question_3_6.A a = q.new A();
```

7. The correct answer is C.

Local classes cannot have access modifiers. The only allowed modifiers are abstract and final.

8. The correct answer is B.

Method add of the anonymous class returned by method create uses the effectively final (as it's never modified) parameter i, not the instance variable of the same name.

Since all the code is valid, 12 is printed.

Chapter FOUR

Interfaces

Exam Objectives

- Develop code that declares, implements and/or extends interfaces and use the `@Override` annotation.

WHAT'S AN INTERFACE?

The first time you look an interface, you'll probably think that it's like a class with just methods definitions:

```
interface Monitorable {  
    void monitor();  
}
```

And for practical terms you're right.

An interface is a data type that just defines (abstract) methods that one class must implement.

Although conceptually, it's more interesting than that, because this allows you to define what a class can do without saying how to do it. That's why it's said that an interface is a *contract*.

Any class that implements an interface must provide an implementation for all the methods of the interface, otherwise, the class has to be marked as abstract.

As a class is defined with the `class` keyword, an interface is defined with the `interface` keyword.

If a class wants to implement an interface, it has to specify it with the `implements` keyword.

DEFINING AN INTERFACE

INTERFACE NAME

```
public interface Monitorable {  
    public static final int ID = 0;  
  
    public abstract void monitor();  
}
```

INTERFACES CAN DEFINE FIELDS AND METHODS.

IMPLEMENTING AN INTERFACE

CLASS NAME

INTERFACE NAME

```
class Server
    implements Monitorable {

    public void monitor() {
        // Implementation code
    }

    }  
METHOD SIGNATURE HAS TO BE THE SAME AS  
THE INTERFACE METHOD.
```

Chapter FOUR

Like a class, an interface has either public or default accessibility:

```
public interface PublicAccessInterface {  
    // ...  
}  
  
interface DefaultAccessInterface {  
    // ...  
}
```

Interfaces are abstract by default (you don't have to specify it):

```
public abstract interface PublicAccessInterface {  
    // This is the same as the definition above  
}
```

This means two things:

- You can't instantiate an interface directly, it has to be implemented by a class to use it.
- An interface cannot be marked as final.

The methods defined in an interface are by default **PUBLIC** and **ABSTRACT**, the compiler will treat them as such even if you don't specify it.

So even though you define an interface like this:

```
interface Monitorable {  
    void monitor();  
    void setup();  
}
```

For the compiler, the interface will look like this:

```
interface Monitorable {  
    public abstract void monitor();  
    public abstract void setup();  
}
```

That's the reason you must mark the method as `public` when it's implemented:

```
class Server implements Monitorable {  
    public void monitor() {  
        // Implementation  
    }  
    public void setup() {  
        // Implementation  
    }  
}
```

Also, that's the reason that, if one or more of the interface's methods are not implemented, you must mark the class (and the methods) as `abstract`:

```
abstract class Server implements Monitorable {  
    public void monitor() {  
        // Implementation  
    }  
    public abstract void setup();  
}
```

Chapter FOUR

Fields declared in an interface are by default **PUBLIC**, **STATIC**, and **FINAL**. Like methods, the compiler will treat them as such even if you don't specify it.

This means that fields are **CONSTANTS** instead of **VARIABLES**:

```
interface Monitorable {  
    int ID = 0; // You have to assign a value at creation time  
}  
  
class Resource implements Monitorable {  
    void change() {  
        ID = 5; // This WON'T compile  
    }  
}
```

This also means that the following declarations are all equivalent inside an interface:

```
int ID = 0;  
public int ID = 0;  
static int ID = 0;  
final int ID = 0;  
public static int ID = 0;  
public final int ID = 0;  
static final int ID = 0;  
public static final int ID = 0;
```

So watch out for declarations that won't compile, like these:

```
interface Monitorable {  
    private int ID = 0; // It cannot be private  
    int TIMEOUT; // It's final, so you have to provide a value  
}
```

There are two rules regarding inheritances and interfaces:

- A class can implement (not extend from) any number of interfaces.
- An interface can extend any number of interfaces, but it cannot extend from a class.

Implementing an interface is a type of inheritance. When a class implements an interface, we can use it like this to take advantage of polymorphism:

```
interface Monitorable {  
    void monitor();  
}  
  
class Disk implements Monitorable {  
    public void monitor() {  
        System.out.println("Monitoring Disk");  
    }  
}  
  
class Server implements Monitorable {  
    public void monitor() {  
        System.out.println("Monitoring Server");  
    }  
}  
  
public class Test {  
    public static void main(String args[]) {  
        Monitorable m = new Disk();  
        m.monitor();  
        m = new Server(); // Change implementation  
        m.monitor();  
    }  
}
```

Chapter FOUR

This is the output:

```
Monitoring Disk  
Monitoring Server
```

A class cannot extend from more than one class, but it can implement more than one interface. You can see the reason with an example. Consider:

```
class Truck {  
    public void accelerate() {  
        System.out.println("Accelerating truck...");  
    }  
}  
  
class CompactCar {  
    public void accelerate() {  
        System.out.println("Accelerating compact car...");  
    }  
}
```

If you could inherit from multiple classes:

```
class Car extends Truck, CompactCar {  
    public void run() {  
        accelerate();  
        // ...  
    }  
}
```

Which `accelerate()` method would Java choose?

This is a problem that the designers of Java decided to avoid by not allowing multiple inheritance.

But what if we were using interfaces?

```
interface Truck {  
    void accelerate();  
}  
  
interface CompactCar {  
    void accelerate();  
}  
  
class Car implements Truck, CompactCar {  
    public void accelerate() {  
        System.out.println("Accelerating car");  
    }  
    // ...  
}
```

As the interfaces don't provide an implementation, there's only one (Car's implementation) so there's no conflict and we avoid the problem altogether!

And if the methods have the same name but different parameters:

```
interface Truck {  
    void accelerate();  
}  
  
interface CompactCar {  
    void accelerate(int speed);  
}
```

They're considered two different methods because the method's signature is different, so the implementing class has to implement both versions of the methods:

Chapter FOUR

```
class Car implements Truck, CompactCar {  
    public void accelerate() {  
        // implementation  
    }  
    public void accelerate(int speed) {  
        // implementation  
    }  
}
```

However, when the methods only differ in their return type, since the return type is not considered in the method's signature, the Java compiler will generate an error:

```
interface Truck {  
    void accelerate();  
}  
interface CompactCar {  
    int accelerate();  
}  
  
class Car implements Truck, CompactCar {  
    // Java will complain about duplicate methods  
    // and incompatible return types  
    public void accelerate() {  
        // implementation  
    }  
    public int accelerate() {  
        // implementation  
    }  
}
```

Optionally, to make things clearer, we can use the `@Override` annotation.

```
interface CompactCar {  
    int accelerate();  
}  
  
class Car implements CompactCar {  
    @Override  
    public int accelerate() {  
        // implementation  
    }  
}
```

`@Override` indicates that a method overrides a method declaration in a supertype, either in an interface or a parent class.

If the annotated method doesn't override or implement the method correctly, the compiler will generate an error.

This is the only function of the annotation. It's useful in cases like when there's a not-so-obvious error caused by a typo:

```
interface CompactCar {  
    int accelerate();  
}  
  
class Car implements CompactCar {  
    // Compiler will mark an error about method  
    // "acelerate" not overriding or implementing something  
    @Override  
    public int acelerate() {  
        // implementation  
    }  
}
```

Chapter FOUR

Finally, an interface can only extend other interfaces.

```
interface Monitoreable {
    void monitor();
}

interface Pluggable {
    void plug();
}

interface Resource extends Monitoreable, Pluggable {
    void printInfo();
}
```

A non-abstract class implementing Resource, must implement all the methods of the three interfaces:

```
class Disk implements Resource {
    public void monitor() {
        // implementation
    }

    public void plug() {
        // implementation
    }

    public void printInfo() {
        // implementation
    }
}
```

WHAT'S NEW IN JAVA 8?

Assume we have an interface like this:

```
interface Processable {  
    void processInSequence();  
}
```

And an implementation:

```
class Task implements Processable {  
    public void processInSequence() {  
        System.out.println("Processing in sequence");  
    }  
}
```

We know that when a class implements an interface, unless the class is marked as abstract, it has to implement **ALL** the methods of that interface.

So if we add another method to Processable, for example:

```
interface Processable {  
    void processInSequence();  
    void processInParallel();  
}
```

We have to update the class to avoid a compilation error:

Chapter FOUR

```
class Task implements Processable {  
    public void processInSequence() {  
        System.out.println("Processing in sequence");  
    }  
    public void processInParallel() {  
        System.out.println("Processing in parallel");  
    }  
}
```

That was easy. But think about the following:

- What if we have hundreds of classes implementing Processable?
- What if we can't update or don't have access to the code for some reason?
- What if the new method is not needed or doesn't make sense for some implementations?

These are real problems, sometimes not easy to solve.

However, Java 8 gives us *default* methods. We don't have to provide implementations for them because they are non-abstract methods.

In other words, interfaces now allow methods with a **BODY**. And this is not as simple as it sounds.

DEFAULT METHODS

The main reason for adding *default* methods to interfaces was to support *interface evolution*, to add new functionality to interfaces and at the same time, ensuring compatibility with the code written for older versions.

There are two other side effects worth mentioning:

- **We can now design *optional* methods.** We can have methods with limited or default functionality so the classes implementing the interfaces can decide if they keep that functionality or provide another one.
- **We can have *utility* methods directly on the interface.** Methods that get or create resources, for example, made just for convenience and possibly implemented in terms of non-default methods of the interface.

However, now that interfaces can provide behavior, the difference with abstract classes is not very clear in some cases. There are still two significant differences:

- A class can only extend from **ONE** abstract class, but it can implement **MULTIPLE** interfaces.
- An abstract class can have a state through **INSTANCE** variables (fields). An interface **CAN'T**.

Default methods come with many rules, especially regarding inheritance. But let's start with their syntax.

DEFINING A DEFAULT METHOD

```
interface Processable {  
    void processInSequence();  
  
    default void processInParallel() {  
        /** Default implementation  
         goes here */  
    }  
}
```

DEFAULT METHODS ARE PUBLIC
IMPLICITLY, JUST AS ANY OTHER
METHOD OF AN INTERFACE

AN INTERFACE CAN HAVE ANY NUMBER OF
ABSTRACT AND DEFAULT METHODS

ALL METHODS WITH
THE KEYWORD
DEFAULT MUST HAVE
A BODY

Chapter FOUR

By making `processInParallel()` a *default* method, the implementing class gets it automatically. Here's the complete example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processing in parallel");
    }
}

public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t.processInSequence();
        t.processInParallel(); // It compiles just fine
    }
}
```

This is the most simple scenario, where the implementing class inherits the *default* method.

Before presenting more complex scenarios, let's see what the restrictions are when using *default* methods are.

Default methods cannot be final.

If a method is final, it cannot be overridden by the implementing classes, which doesn't favor the primary objective of *default* methods.

Default methods cannot be synchronized.

This was a deliberate decision by the designers of the language. If a method is made synchronized in the interface, it would mean that all the implementing classes would inherit this behavior. But this decision should belong to the implementation; the interface has no reasonable basis for assuming what the synchronization policy should be.

Default methods are always public.

Like other methods of an interface. Contrast this with an abstract class, where you can choose the visibility of the method.

You cannot have default methods for the Object's class methods.

An interface cannot provide default implementations for:

```
boolean equals(Object o)  
int hashCode()  
String toString()
```

If an interface has methods with those signatures, the compiler will throw an error. The reason is that those methods are all about the object's state. Since interfaces do not have a state, these methods should be in the implementing classes.

And now, to the more complex scenarios.

CLASS OVERIDES DEFAULT METHOD

Classes always **WIN** over interfaces. If a class overrides a default method, the class method will be the one used. For example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}

public class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public void processInParallel() {
        System.out.println("Class parallel");
    }
    public static void main(String args[]) {
        Task t = new Task();
        t. processInParallel();
    }
}
```

The output would be:

```
Class parallel
```

This is true even if the class redefines the default method as abstract:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}
// Class Task has to be abstract
abstract class Task implements Processable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }
    public abstract void processInParallel();
}
```

If for some reason, you need to call the default implementation of the method, you can do it with the name of the interface followed by the keyword super:

```
public void processInParallel() {
    Processable.super.processInParallel();
}
```

This only works with default methods. Calling a non-default method in this way will result in a compilation error. Also, super must be used with a direct super interface of the class.

Chapter FOUR

Another scenario related to this rule is when an inherited instance method from a class overrides a default interface method:

```
interface Processable {
    default void processInParallel() {
        System.out.println("Default parallel");
    }
}

class Process {
    public void processInParallel() {
        System.out.println("Class parallel");
    }
}

public class Task extends Process implements Processable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
    }
}
```

The output is:

```
Class parallel
```

The method `processInParallel()` returns the string "Class parallel" since the class `Task` inherits the method `processInParallel()` from the class `Process`, which overrides the default method of the same name in the interface `Processable`.

INTERFACE INHERITANCE WITH DEFAULT METHODS

More specific interfaces (or classes) always **WIN** over less specific ones. The default methods of the shallower interfaces in an inheritance hierarchy will be used. For example:

```
interface Processable {  
    void processInSequence();  
    default void processInParallel() {  
        System.out.println("Processable parallel");  
    }  
}  
  
interface Parallelizable extends Processable {  
    default void processInParallel() {  
        System.out.println("Parallelizable parallel");  
    }  
}  
  
public class Task implements Parallelizable {  
    public void processInSequence() {  
        System.out.println("Processing in sequence");  
    }  
  
    public static void main(String args[]) {  
        Task t = new Task();  
        t.processInParallel();  
    }  
}
```

Chapter FOUR

The output would be:

```
Parallelizable parallel
```

The interface Parallelizable inherits the default method processInParallel(), but since it redefines the method when Task implements it, its implementation is the one called.

If Parallelizable defined processInParallel() as abstract:

```
interface Parallelizable extends Processable {  
    abstract void processInParallel();  
}
```

Then Task would have to implement the method (to not become an abstract class):

```
public class Task implements Parallelizable {  
    public void processInSequence() {  
        System.out.println("Processing in sequence");  
    }  
    public void processInParallel() {  
        System.out.println("Task parallelizable");  
    }  
    public static void main(String args[]) {  
        Task t = new Task();  
        t.processInParallel();  
    }  
}
```

The output would be:

```
Task parallelizable
```

MULTIPLE INTERFACE INHERITANCE WITH DEFAULT METHODS

Classes can implement multiple interfaces. What happens when two interfaces have the same default method? Which one does the implementing class will choose? For example:

```
interface Processable {
    void processInSequence();
    default void processInParallel() {
        System.out.println("Processable parallel");
    }
}

interface Parallelizable {
    default void processInParallel() {
        System.out.println("Parallelizable parallel");
    }
}

public class Task implements Processable, Parallelizable {
    public void processInSequence() {
        System.out.println("Processing in sequence");
    }

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

Chapter FOUR

It turns out that the result is a compiler error:

Duplicate **default** methods named `processInParallel` with the parameters () and () are inherited from the types `Parallelizable` and `Processable`.

The compiler doesn't know which to choose, so it generates an error. In this case, `Task` has to provide an implementation (honoring the previous rule of *more specific interfaces (or classes) always wins over less specific ones*) to override the interface default methods and solve the issue:

```
public class Task implements Processable, Parallelizable {  
    public void processInSequence() {  
        System.out.println("Processing in sequence");  
    }  
    public void processInParallel() {  
        System.out.println("Task parallelizable");  
    }  
    public static void main(String args[]) {  
        Task t = new Task();  
        t.processInParallel();  
    }  
}
```

And now, the output would be:

Task parallelizable

Of course, we can always call a default implementation with:

```
public void processInParallel() {  
    Procesable.super.processInParallel();  
}
```

STATIC METHODS

Whenever we refer to something static, we mean something that belongs to a class, not to a particular instance or object of that class. Static methods on interfaces follow the same concept; they belong to the interface where they are declared.

They were added to assist default methods and to better organize helper methods, because generally, helper or utility methods are defined in another class (like `java.util.Collections`), instead of where they naturally belong.

For example, the `java.util.Comparator` interface defines the static method:

```
static <T> Comparator<T>
comparingInt(ToIntFunction<? super T> keyExtractor)
```

Used by the default method:

```
default Comparator<T> thenComparingInt(ToIntFunction<?
super T> keyExtractor)
```

Interface static methods are not inherited, you must prefix the method with the interface name:

DEFINING A STATIC METHOD

```
interface Processable {  
  
    static void log() {  
        /** Implementation goes here */  
    }  
}
```

STATIC METHODS ARE PUBLIC
IMPLICITLY, JUST AS ANY OTHER
METHOD OF AN INTERFACE

STATIC METHODS IN INTERFACES ARE DEFINED JUST LIKE STATIC METHODS IN CLASSES, WITH THE KEYWORD STATIC

AN INTERFACE CAN HAVE ANY NUMBER OF STATIC METHODS

Chapter FOUR

```
interface Parallelizable {
    static void log(String s) {
        System.out.println(s);
    }
    default void processInParallel() {
        log("Parallelizable parallel");
    }
}

public class Task implements Parallelizable {
    public static void main(String args[]) {
        Task t = new Task();
        t.processInParallel();
        // t.log("The end"); Doesn't compile
        // Task.log("The end"); Doesn't compile either
        Parallelizable.log("The end"); // Compiles!
    }
}
```

The output is:

```
Parallelizable parallel
The end
```

KEY POINTS

- An interface is a data type that just defines (abstract) methods that one class must implement.
- An interface is defined with the `interface` keyword. If a class wants to implement an interface, it has to specify it with the `implements` keyword.
- An interface has either `public` or `default` accessibility and is `abstract` by default.
- The methods defined in an interface are by default `public` and `abstract`. The compiler will treat them as such even if you don't specify it.
- Fields declared in an interface are by default `public`, `static`, and `final`. Like methods, the compiler will treat them as such even if you don't specify it.
- This means that fields are constants instead of variables.
- A class can implement (not extend from) any number of interfaces
- An interface can extend any number of interfaces, but it cannot extend from a class.
- `@Override` indicates that a method overrides a method declaration in a supertype, either in an interface or a parent class.
- If the annotated method doesn't override or implement the method correctly, the compiler will generate an error.

Chapter FOUR

- Java 8 introduced default methods in interfaces to support *interface evolution*, adding new functionality to interfaces and at the same time, ensuring compatibility with the code written for older versions.
- Default methods are methods marked with the `default` keyword, and they must have a body. The implemented classes can use and optionally redefine these methods.
- Default methods are always `public`, but not `static`.
- Default methods cannot be `synchronized` or `final`.
- You cannot define default methods with the same signature as the `Object` class methods:

```
boolean equals(Object o);  
int hashCode();  
String toString();
```

- In an inheritance hierarchy, the most specific method is the one called.
- One case is if a class redefines a default method, the class method is the one that gets called.
- Another case is if an interface redefines a default method inherited from a super interface, the method of the subinterfaces is the one called.
- If a class implements two different interfaces with the same default method (same method signature), the class must redefine the method. Otherwise, a compiler error is

generated.

- If you want to call the default method of the interface from the implementing class (or extending interface), do it this way:

```
NameOfTheInterface.super.defaultMethod();
```

- Java 8 also introduced static methods in interfaces so they can contain helper or utility methods.
- Static methods are marked with the static keyword, and they also must have a body.
- Static methods in interfaces have the same meaning that static methods in classes, so they are not inherited.
- If you want to call a default method from a particular interface, do it this way:

```
NameOfTheInterface.staticMethod();
```

SELF TEST

1. Given:

```
interface A {  
    default int aMethod() {  
        return 0;  
    }  
}  
  
public class Test implements A {  
    public long aMethod() {  
        return 1;  
    }  
  
    public static void main(String args[]) {  
        Test t = new Test();  
        System.out.println(t.aMethod());  
    }  
}
```

What is the result?

- A. 0
- B. 1
- C. Compilation fails
- D. An exception occurs at runtime

2. Given:

```
interface B {  
    default static void test() {  
        System.out.println("B test");  
    }  
}  
  
public class Question_4_2 implements B {  
  
    public void test() {  
        System.out.println("Q test");  
    }  
  
    public static void main(String[] args) {  
        Question_4_2 q = new Question_4_2();  
        q.test();  
    }  
}
```

What is the result?

- A. B test
- B. Q test
- C. Compilation fails
- D. An exception occurs at runtime

Chapter FOUR

3. Given:

```
interface C {  
    default boolean equals(C obj) {  
        return obj == this;  
    }  
}  
  
public class Question_4_3 implements C {  
  
    public static void main(String[] args) {  
        Question_4_3 q = new Question_4_3();  
        System.out.println(q.equals(q));  
    }  
}
```

What is the result?

- A. true
- B. false
- C. Compilation fails
- D. An exception occurs at runtime

4. Given:

```
interface D {  
    default void print() {  
        System.out.println("D");  
    }  
}  
interface E extends D {  
    default void print() {  
        System.out.println("E");  
    }  
}  
  
public class Question_4_4 implements E {  
  
    public void print() {  
        E.super.print();  
    }  
  
    public static void main(String[] args) {  
        Question_4_4 q = new Question_4_4();  
        q.print();  
    }  
}
```

What is the result?

- A. D
- B. E
- C. D and then E
- D. Compilation fails
- E. An exception occurs at runtime

Chapter FOUR

5. Given:

```
interface F {  
    static void test() {  
        System.out.println("F test");  
    }  
}  
  
public class Question_4_5 implements F {  
  
    public void test() {  
        System.out.println("Q test");  
    }  
  
    public static void main(String[] args) {  
        F q = new Question_4_5();  
        q.test();  
    }  
}
```

What is the result?

- A. F test
- B. Q test
- C. Compilation fails
- D. An exception occurs at runtime

6. Given:

```
interface G {
    default void doIt() {
        System.out.println("G - Do It");
    }
}

interface H {
    void doIt();
}

public class Question_4_6 implements G, H {

    public void doIt() {
        System.out.println("Do It");
    }

    public static void main(String[] args) {
        Question_4_6 q = new Question_4_6();
        q.doIt();
    }
}
```

What is the result?

- A. G - Do It
- B. Do It
- C. Compilation fails
- D. An exception occurs at runtime

ANSWERS

1. The correct answer is C.

Since the method signature (method name plus parameter list) of `aMethod()` is the same in interface A and class Test, the compiler thinks the method in Test is overriding the default method. However, overriding a method includes the return type, which doesn't match (`long` vs. `int`).

2. The correct answer is C.

An interface method cannot be `default` and `static` at the same time. Therefore, a compile-time error is generated.

3. The correct answer is A.

There is nothing wrong with the code. The `equals` method is valid since it doesn't have the signature of `Object`'s `equals`. The `this` keyword can be used inside the default method (it refers to the object that implements the interface) and because we are testing the same instance (`q`), the code returns true.

4. The correct answer is B.

If you want to call the default method of the super interface from the implementing class, you have to do it with the following syntax:

```
NameOfTheInterface.super.defaultMethod();
```

However, this only works with the most direct super interface.

In the case of the example, interface E. If you try to use:

```
D.super.print();
```

The compiler would generate an error saying that you cannot bypass a more specific direct super type.

5. The correct answer is C.

Interface F defines a static method that should be used as:

```
F.test();
```

Since the type of variable q is F, you cannot use method test(). If the type of variable q were Question_4_5, the program would print Q test.

By the way, test() on Question_4_5 doesn't redefine test() on interface F, since static methods on interfaces are not inherited.

6. The correct answer is B.

In an inheritance hierarchy, the most specific method is the one called, which in this case, it's the one defined by Question_4_6. If we remove method doIt() in class Question_4_6, the program would stop compiling, since the default method doIt() inherited from G will conflict with the method inherited from H (the compiler wouldn't know which one you intended to run).

Chapter FOUR

Chapter FIVE

Enumerations

Exam Objectives

- Use enumerated types including methods, and constructors in an enum type.

ENUMERATIONS

Let's say our application uses three states or values for the volume of playing a song: high, medium, low. We typically model this with some "constants", like this:

```
public interface Volume {  
    public static final int HIGH = 100;  
    public static final int MEDIUM = 50;  
    public static final int LOW = 20;  
}
```

However, this is not a good implementation. Consider a method to change the volume:

```
public void changeVolumen(int volume) {  
    ...  
}  
...  
app.changeVolumen(Volume.HIGH);
```

What is stopping someone to call this method with an arbitrary value like:

```
app.changeVolume(10000);
```

Of course, we can implement some checks, but this just makes the problem more complicated. Luckily, enumerations (or enums for short) provide a nice solution to this issue.

Chapter FIVE

An *enum* is a type (like a class or an interface) that represents a **FIXED** list of values (you can think of these as constants).

So we can use an enum to represent the volume of a sound in our application (notice the use of `enum` instead of `interface`):

```
public enum Volume {  
    HIGH, MEDIUM, LOW  
}
```

Notice that since the values are "constants" (they are implicitly `public`, `static`, and `final`) the convention of using all caps is followed.

Also, notice that the values are comma-separated and because the `enum` only contains a list of values, the semicolon is optional after the last one.

This way, we can define the `changeVolume` method as:

```
public void changeVolume(Volume volume) {  
    ...  
}
```

Passing any other object that is not a `Volume`, will generate a compile-time error:

```
changeVolume(Volume.HIGH); // All good  
changeVolume(-1); // Compile-time error
```

The method is now type-safe, meaning that we can't assign invalid values.

By the way, because of something we review later, you cannot use the new operator to get a reference of an enum, so we just get the reference directly:

```
Volume level = Volume.LOW;
```

You can also get an enum from a string, for example:

```
Volume level = Volume.valueOf("LOW");
```

Just be careful, this method is case sensitive:

```
Volume level = Volume.valueOf("Low"); // Run-time exception
```

To get all the enums of a particular type use the method `values()`, that returns an array of enums in the same order in which they were declared and that pairs great with a for-each statement:

```
for(Volume v: Volume.values()) {  
    System.out.print(v.name() + ", ");  
}
```

The output:

```
HIGH, MEDIUM, LOW,
```

WORKING WITH ENUMS

In the first example, HIGH was equal to 100. But now that we are using enums, what is the value of HIGH?

If we print its value:

```
System.out.println(Volume.HIGH);
```

The output will be the name of the enum:

```
HIGH
```

This is equivalent to invoke the name() method that all enums have:

```
System.out.println(Volume.HIGH.name());
```

In other words, the `toString()` implementation of an enum calls the name() method.

Enums also have a zero-based value that corresponds to the order in which they're declared. You can get it with the ordinal() method:

```
System.out.println(Volume.HIGH.ordinal());
System.out.println(Volume.LOW.ordinal());
```

The output will be:

```
0
```

```
2
```

But this might not be enough for all cases (like in our example). So we can define a constructor (that is called the first time the enum is used) that accepts a parameter that will be stored in an instance variable:

```
public enum Volume {  
    HIGH(100), MEDIUM(50), LOW(20);  
    private int value;  
  
    private Volume(int value) {  
        this.value = value;  
    }  
  
    public int getValue() { return value; }  
}
```

The only restriction is the constructor must be private. Otherwise, the compiler will throw an error. Notice that a getter method was also added.

We can add a setter method also, but generally, it is not need, since enums work more like constants. However, keep in mind that changing the value of the instance variable is allowed, while reassigning the enum is not because they are implicitly final and cannot be changed after their creation:

```
Volume.HIGH = Volume.MEDIUM; // Compile-time error
```

Talking about compile-time errors, the following statements will also generate one:

Chapter FIVE

```
if(Volume.HIGH == 0) { // Use Volume.HIGH.ordinal()  
    ...  
}  
if(Volume.HIGH == 100) { // Use Volume.HIGH.getValue()  
    ...  
}  
// A enum can't extend a class  
public enum Volume extends AClass { ... }
```

To compare the value of an enum you can use either the ordinal or name methods, in addition to any other custom method. You can also compare an enum to another enum with either the == operator (because enums are final), the equals method, or by using a switch statement:

```
Volume level = Volume.HIGH;  
...  
if(Volume.HIGH == level) { // Or Volume.HIGH.equals(level)  
    ...  
}  
switch(level) {  
    // Notice that the only the name of the enum is used,  
    // in fact, Volume.HIGH for example, won't compile  
    case HIGH: ...  
    case MEDIUM: ...  
    case LOW: ...  
}
```

An enum type **CANNOT** extend from a class because implicitly, all enums extend from `java.lang.Enum`. What you **CAN** do is implement interfaces:

```
public enum Volume implements AnInterface { ... }
```

The closest we can get to extending a class when working with enums is overriding methods and implementing abstract methods. For example:

```
public enum Volume {
    HIGH(100) {
        public void printValue() {
            System.out.println("** Highest value**");
        }
        public void printDescription() {
            System.out.println("High Volume");
        }
    }, MEDIUM(50) {
        public void printDescription() {
            System.out.println("Medium Volume");
        }
    }, LOW(20) {
        public void printDescription() {
            System.out.println("Low Volume");
        }
    };
    private int value;

    private Volume(int value) {
        this.value = value;
    }

    public void printValue() { System.out.println(value); }
    public abstract void printDescription();
}
```

In the case of `printValue()`, `MEDIUM` and `LOW` will use the enum-level version that just prints the value while `HIGH` will use its own version. If the method is abstract, every enum has to implement it. Otherwise, a compile-time error will be thrown.

ENUMS AS SINGLETONS

Remember back in Chapter 1 when I mention that enums are singletons? Do you know why now?

- You cannot create an instance of an enum by using the new operator (because the constructor is private).
- An instance of an enum is created when the enum is first referenced.
- An enum can't be reassigned.
- I didn't mention it before, but enums are thread-safe by default (meaning that you don't need to do double checks when creating them).
- In Chapter 1, the impact of serialization on singletons wasn't really explored, but if you serialize a singleton, when you get it back with the default implementation of `readObject()`, this method will always return a new instance, so the singleton is not really one anymore. However, when serializing an enum, this won't happen.

Given those reasons, most people believe that most of the time, enums are the best way to implement the singleton design pattern in Java.

KEY POINTS

- An *enum* is a type that represents a **FIXED** list of values (you can think of these as constants), providing type safety.
- Enums can define constructors, but they must be **private**. Otherwise, a compile-time error will be thrown.
- Enums are implicitly **public**, **static** and **final**.
- An *enum* can be created from a *String* using the case-sensitive *valueOf()* method.
- To get all the enums of a certain type use the method *values()*, that returns an array of enums in the same order in which they were declared.
- When the *toString()* method is invoked, it prints the name of the enumeration.
- Enums can be compared against other enums using the **==** operator and the *equals()* method.
- Enums can be used in *switch* statements.
- Enums can implement interfaces, but they cannot extend from a class since they implicitly extend from *java.lang.Enum*.
- Enums are the easiest way to implement *singletons*.

SELF TEST

1. Given:

```
public enum Question_5_1 {  
    UP(1) {  
        public void printValue() {  
            System.out.println(value);  
        }  
    }, DOWN(0);  
    private int value;  
  
    private Question_5_1(int value) {  
        this.value = value;  
    }  
}
```

What is the result of executing Question_5_1.UP.printValue()?

- A. 1
- B. 0
- C. Compile-time error
- D. Run-time error

2. Given:

```
enum Color {  
    Blue, Green, Black  
}  
  
public class Question_5_2 {  
    public static void main(String[] args) {  
        Color c = Color.values()[0];  
        switch(c) {  
            case Blue: System.out.println(1); break;  
            case Green: System.out.println(2); break;  
            case Black: System.out.println(3); break;  
        }  
    }  
}
```

What is the result of executing Question_5_2?

- A. 1
- B. 2
- C. 3
- D. Compile-time error
- E. Run-time error

3. Which of the following statements is true?

- A. Enums are thread-safe.
- B. Enums can neither extend from a class nor implement an interface.
- C. Enums cannot define constructors.
- D. Enums cannot have setter methods.

Chapter FIVE

4. Given:

```
enum Level {
    HIGH(100), LOW(10);
    private int value;
    private Level(int value) {
        this.value = value;
        System.out.println(value);
    }
}
public class Question_5_4 {
    public static void main(String[] args) {
        Level l1 = Level.HIGH;
        Level l2 = Level.HIGH;
    }
}
```

What is the result of executing class Question_5_4?

- A. 100
- B. 100
 100
- C. 100
 10
- D. Compile-time error
- E. Nothing is printed

5. Given:

```
enum Color1 {
    RED, YELLOW
}
enum Color2 {
    RED, PINK
}
public class Question_5_5 {
    public static void main(String[] args) {
        if(Color1.RED.equals(Color2.RED)) {
            System.out.println(1);
        } else {
            System.out.println(0);
        }
    }
}
```

What is the result of executing Question_5_5?

- A. 1
- B. 0
- C. Compile-time error
- D. Run-time error

6. Which of the following statements are true?

- A. You can compare two enumerations using the == operator.
- B. Enums implicitly inherit from java.lang.Enum.
- C. You can't use the new operator inside an enum.
- D. You can't have abstract methods inside an enum.

ANSWERS

1. The correct answer is C.

An enum is implicitly static. This makes the `printValue()` method execute in a static context. So, the reference in the method `printValue()` to the variable `value` is invalid, because `value` is non-static.

2. The correct answer is A.

There's nothing wrong with the code. Since `values()` returns an array of enums in the same order in which they were declared, Blue is returned, and 1 is printed.

3. The correct answer is A.

Option A is true. Enums are thread-safe.

Option B is false. Enums can't extend from a class, but they can implement an interface.

Option C is false. Enums can define private constructors.

Option D is false. Enums can have setter methods.

4. The correct answer is C.

Enums are created just in time, the first time they are referenced. When variable `l1` is created, the enum `Level` is also created, calling the constructor of `HIGH` and `LOW` and printing 100 and 10.

5. The correct answer is B.

Enums constants from different enumerations are not equal, even if they share the same name.

6. The correct answers are A and B.

Option A is true. You can compare two enumerations using the == operator.

Option B is true. Enums implicitly inherit from java.lang.Enum.

Option C is false. You can use the new operator inside an enum, but you cannot use the new operator to create a reference to an enum.

Option D is false. You can have abstract methods inside an enum, but all enum constants must provide an implementation.

Chapter FIVE

Part TWO

Generics and Collections

Chapter SIX

Generics

Exam Objectives

- Create and use a generic class.

GENERICS

Without generics, you can declare a List like this:

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

Because a List, by default, accepts objects of any type, you can add elements of different types to it:

```
list.add("a");
list.add(new Integer(1));
list.add(Boolean.TRUE);
```

And get values like this:

```
String s = (String) list.get(0);
```

This can lead to ugly runtime errors and more complexity. Because of that, generics were added in Java 5 as a mechanism for type checking.

A generic is a type declared inside angle brackets, following the class name. For example:

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

By adding the generic type to List, we are telling the **COMPILER** to check that only String values can be added to the list:

```
list.add("a");           // OK
list.add(new Integer(1)); // Compile-time error
list.add(Boolean.TRUE); // Compile-time error
```

Chapter SIX

Since now we only have values of one type, we can safely get elements without a cast:

```
String s = list.get(0);
```

It's important to emphasize that generics are a thing of the compiler. At runtime, Java doesn't know about generics.

Under the hood, the compiler inserts all the checks and casts for you, but at runtime, a generic type is seen by Java as a `java.lang.Object` type.

In other words, the compiler verifies that you're working with the right type and then, generates code with the `java.lang.Object` type.

The process of replacing all references to generic types with `Object` is called *type erasure*.

Because of this, at runtime, `List<String>` and `List<Integer>` are the same, because the type information has been erased by the compiler (they are just seen as `List`).

Generics only work with objects. Something like the following won't compile:

```
List<int> list = new ArrayList<int>();
```

Finally, a class that accepts generics but is declared without one is said to be using a *raw type*:

```
List raw = new ArrayList(); // Raw type
List<String> generic = new ArrayList<String>(); // Generic type
```

THE DIAMOND OPERATOR

Since Java 7, instead of specifying the generic type on both sides of the assignment:

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

We can simplify the creation of the object by just writing:

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

The short form on the right side is called the *diamond operator* (because it looks like a diamond).

But be careful. The above example is different than:

```
// Without the diamond operator, the raw type is used
List<List<String>> generic = new ArrayList();
```

You can only use the diamond operator if the compiler can infer the parameter type(s) from the context. The good news is that in Java 8, type inference was improved:

```
void testGenericParam(List<String> list) { }

void test() {
    // In Java 7, this line generates a compile error
    // In Java 8, this line compiles fine
    testGenericParam(new ArrayList<>());
}
```

GENERIC CLASSES

Looking at the definition of `List` and a couple of its methods, we can see how this class is designed to work with generics:

```
public interface List<E> extends Collection<E> {  
    boolean add(E e);  
    Iterator<E> iterator();  
}
```

We can see how a generic type is defined for classes and interfaces. It's just a matter of declaring a *type parameter* next to the class (or interface) name.

By the way, `E` is just an identifier, like a named variable. It can be anything you want. However, the convention is to use single uppercase letters. Some common letters are:

- `E` for element
- `K` for a map key
- `V` for a map value
- `T, U` for data types

This way, when a `List` is declared like this:

```
List<String> list = null;
```

`E` is given the value of `String`, and wherever the type `E` is defined, `String` will be used.

So generic classes give us a lot of flexibility.

For example, consider this class:

```
class Holder {  
    private String s;  
    public Holder(String s) {  
        this.s = s;  
    }  
    public String getObject() {  
        return s;  
    }  
    public void printObject() {  
        System.out.println(s);  
    }  
}
```

There's nothing wrong with it, but it only accept objects of type `String`. What if later we need a class just like that, but that works with `Integer` types? Do we create an `Integer` version?

```
class IntegerHolder {  
    private Integer s;  
    public Holder(Integer s) {  
        this.s = s;  
    }  
    public Integer getObject() {  
        return s;  
    }  
    public void printObject() {  
        System.out.println(s);  
    }  
}
```

Duplicate code feels and looks wrong. An `Object` version? No, thank you, we will need to add casts everywhere.

Chapter SIX

Generics help us in cases like this. Just declare a type parameter:

```
class Holder<T> {  
    // ...  
}
```

And the generic type T will be available anywhere within the class:

```
class Holder<T> {  
    private T t;  
    public Holder(T t) {  
        this.t = t;  
    }  
    public T getObject() {  
        return t;  
    }  
    public void printObject() {  
        System.out.println(t);  
    }  
}
```

Now, when an instance is created, we just specify the type of T for that instance:

```
Holder<String> h1 = new Holder<>("Hi");  
Holder<Integer> h2 = new Holder<>(1);  
String s = h1.getObject();
```

If we don't specify a type parameter, we will be using the raw type (that uses the Object type):

```
Holder h3 = new Holder("Hi again");  
Object o = h3.getObject();
```

If we need it, we can have more than one type parameter:

```
class Holder<T, U> {  
    // ...  
}
```

GENERIC METHODS

We can also declare type parameters in any method (not for the whole class). But the syntax is a little different, for example:

```
class Utils {  
    public static <T> void print(T t) {  
        System.out.println(t);  
    }  
}
```

This defines a method that takes an argument of type T. Here are two more examples of generic methods:

```
<T> void genericMethod1(List<T> list) {}  
<T, U> T genericMethod2(U u) {  
    T t = null;  
    return t;  
}
```

When a method declares its own generic type, it has to be specified before the return type (in contrast to classes, which declare it after the class name).

To call the method of the first example, you can do it normally:

```
Utils().print(10);
```

Or by explicitly specifying the type between the dot and the name of the method:

```
Utils().<Integer>print(10);
```

WILDCARDS

Generics are useful in many cases, but not all. We have two main problems.

You could think that since `ArrayList` implements `List`, and because `String` is a subclass of `Object`, something like this is fine:

```
List<Object> list = new ArrayList<String>();
```

But it doesn't compile. An `ArrayList<String>` cannot be cast to `List<Object>` because when working with generics, you cannot assign a derived type to a base type; both types should be the same (either explicitly or by using the diamond operator).

Think about it this way: a list of type `List<Object>` can hold instances of `Object` and its subclasses. In other words, the list could hold any object type, not only strings. So you could have a list of strings and integers for example, which clearly violates type safety.

But if you change the declaration to use a wildcard parameter:

```
List<?> list = new ArrayList<String>();
```

It will compile.

The unbounded wildcard type (`<?>`) means that the type of the list is unknown so that it can match **ANY** type.

In fact you can consider in a way `List<?>` as the superclass of all Lists, since you can assign any type of List:

```

List<String> stringList = new ArrayList<>();
List<Integer> intList = new ArrayList<>();
List<?> unknownTypeList = stringList; // No problem
List<?> unknownTypeList = intList; // No problem either
for(Object o : unknownTypeList) { // Object?
    System.out.println(o);
}

```

Since the compiler doesn't know the type of the elements of `List<?>`, we have to use `Object` to assure there won't be any problem at runtime.

But don't think that `List<Object>` is the same as `List<?>`. It's not. With `List<Object>` the previous examples won't compile.

There's another difference. The following code won't compile:

```

List<?> list = new ArrayList<String>();
list.add("Hi"); // Compile-time error

```

Since the compiler cannot infer the type of the elements, it can't assure type safety (you can only insert `null` because it doesn't have a type).

To avoid this problem, the compiler generates an error when you try to modify the list. This way, when using an unbounded wildcard the list becomes **IMMUTABLE**.

This can be a problem or a benefit, depending on how you use it. This wildcard is used in arguments of methods where the code just uses methods of the generic class or from `Object`, not of a particular type, for example:

```

int getSize(List<?> list) { // You can pass any type of List here
    return list.size();
}

```

Chapter SIX

That was the first problem. The second problem is that when working with a type parameter, we can only use methods from `Object` since we don't know the exact type of the type parameter, for example:

```
class Printer<T> {
    public void print(T t) {
        System.out.println(t.toUpperCase()); // Error
        // What if T doesn't represent a String?
    }
}
```

The solution is to use the so-called bounded wildcards:

- `? extends T` (Upper-bounded wildcard)
- `? super T` (Lower-bounded wildcard)

By using these wildcards, you can relax a little the restrictions imposed by generics. This will also allow you to use some sort of polymorphism or subtyping with generics, and for that same reason, this is the trickiest part of the exam.

Let's start with the upper-bounded wildcard.

The error in the example above can be solved using the upper-bounded generic (not exactly a wildcard) this way:

```
class Printer<T extends String> {
    public void print(T t) {
        System.out.println(t.toUpperCase()); // OK!
    }
}
```

`<T extends String>` means that any class that extends (or implements when working with an interface) `String` (or `String` itself) can be used as

the type parameter. As T is replaced by String, it's safe to its methods:

```
Printer<String> p1 = new Printer<>(); // OK
Printer<Byte> p2 = new Printer<>(); // Error, Byte is not a String
```

The upper-bounded wildcard can also solve this problem:

```
List<Object> list = new ArrayList<String>(); // Error
List<? extends Object> list2 = new ArrayList<String>(); // OK!
```

Still, we can't modify the list:

```
list2.add("Hi"); // Compile-time error
```

The reason is the same. The compiler still can't know for sure what type will the list hold (we could add any type).

Notice then, that List<Number> is more restrictive than List<? extends Number>, in the sense that the former only accepts direct assignments of type List<Number>, but the latter, accepts direct assignments of List<Integer>, List<Float>, etc. For example:

```
List<Integer> listInteger = new ArrayList<>();
List<Float> listFloat = new ArrayList<>();
List<Number> listNumber = new ArrayList<>();
listNumber.add(new Integer(1));           // OK
listNumber.add(new Float(1.0F));          // OK
listNumber = listInteger;                // Error
listNumber = listFloat;                  // Error

List<? extends Number> listExtendsNum = new ArrayList<>();
// listExtendsNum.add(new Integer(1)); This would cause an error
listExtendsNum = listInteger;           // OK
listExtendsNum = listFloat;             // OK
```

Chapter SIX

Finally, we have the lower-bounded wildcard. If we have a list like this:

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

It means that list can be assigned to an Integer list (`List<Integer>`) or some supertype of Integer (like `List<Number>` or `List<Object>`).

This time, since you know that the list would be typed to at least an Integer, it's safe for the compiler to allow modifications to the list:

```
List<? super Integer> list = new ArrayList<>();
list.add(1);      // OK!
list.add(2);      // OK!
```

Think about it, even if the list's type is `List<Object>`, an Integer can be assigned to an Object or a Number (or another superclass if there were another one) for that matter.

And what types can we add to the list?

We can add instances of T or one of its subclasses because they are T also (in the example, Integer doesn't have subclasses, so we can only insert Integer instances).

So don't get confused, one thing is what can you assign and another thing is what you can add, for example:

```
List<Integer> listInteger = new ArrayList<>();
List<Object> listObject = new ArrayList<>();
List<? super Number> listSuperNum = new ArrayList<>();
listSuperNum.add(new Integer(1)); // OK
listSuperNum.add(new Float(1.0F)); // OK
listSuperNum = listInteger;      // Error!
listSuperNum = listObject;      // OK
```

GENERIC LIMITATIONS

We have talked about some of the limitations of generics, and other can be inferred from what we've reviewed, but anyway, here's a summary of all of them:

Generics don't work with primitive types:

```
List<int> list = new ArrayList<>(); // Use Wrappers instead
```

You cannot create an instance of a type parameter:

```
class Test<T> {  
    T var = new T(); // You don't know the type's constructors  
}
```

You cannot declare static fields of a type parameter:

```
class Test<T> {  
    // If a static member is shared by many instances,  
    // and each instance can declare a different type,  
    // what is the actual type of var?  
    static T var;  
}
```

Due to type erasure, you cannot use instanceof with generic types:

```
if(obj instanceof List<Integer>) { // Error  
}  
if (obj instanceof List<?>) { // It only works with the unbounded  
} // wildcard to verify that obj is a List
```

Chapter SIX

You cannot instantiate an array of generic types

```
class Test<T> {  
    T[] array; // OK  
    T[] array1 = new T[100]; // Error  
    List<String>[] array2 = new List<String>[10]; // Error  
}
```

You cannot create, catch, or throw generic types

```
class GenericException<T> extends Exception { } // Error  
  
<T extends Exception> void method() {  
    try {  
        // ...  
    } catch(T e) { // Error  
    }  
}
```

However, you can use a type parameter in a throws clause:

```
class Test<T extends Exception> {  
    public void method() throws T { } // OK  
}
```

You cannot overload a method where type erasure will leave the parameters with the same type:

```
class Test { // List<Integer> and List<Integer>  
            // will be converted to List at runtime  
    public void method(List<String> list) { }  
    public void method(List<Integer> list) { }  
}
```

KEY POINTS

- Generics are a mechanism for type checking at compile-time.
- The process of replacing all references to generic types at runtime with an Object type is called *type erasure*.
- A generic class used without a generic type argument (like `List list = null;`) is known as a *raw type*.
- The *diamond operator* (`<>`) can be used to simplify the use of generics when the type can be inferred by the compiler.
- It's possible to define a generic class or interface by declaring a *type parameter* next to the class or interface name.
- We can also declare type parameters in any method, specifying the type before the method return type (in contrast to classes, which declare it after the class name).
- The unbounded wildcard type (`<?>`) means that the type of the list is unknown so that it can match **ANY** type. This also means that for example, `List<?>` is a supertype of any List type (like `List<Integer>` or `List<Float>`).
- The upper-bounded wildcard (`? extends T`) means that you can assign either T or a subclass of T.
- The lower-bounded wildcard (`? super T`) means that you can assign either T or a superclass of T.

SELF TEST

1. Given:

```
public class Question_6_1 {  
    public static void main(String[] args) {  
        Question_6_1 q = new Question_6_1();  
        List<Integer> l = new ArrayList<>();  
        l.add(20);  
        l.add(30);  
        q.m1(l);  
    }  
    private void m1(List<?> l) {  
        m2(l); // 1  
    }  
    private <T> void m2(List<T> l) {  
        l.set(1, l.get(0)); // 2  
        System.out.println(l);  
    }  
}
```

What is the result?

- A. [20, 20]
- B. Compilation fails on the line marked as // 1
- C. Compilation fails on the line marked as // 2
- D. An exception occurs at runtime

2. Given:

```
public class Question_6_2 <T extends Number> {
    T t;
    public static void main(String[] args) {
        Question_6_2 q = new Question_6_2<Integer>(); // 1
        q.t = new Float(1); // 2
        System.out.println(q.t);
    }
}
```

What is the result?

- A. 1.0
- B. Compilation fails on the line marked as // 1
- C. Compilation fails on the line marked as // 2
- D. An exception occurs at runtime

3. Which of the following declarations don't compile?

- A. List<?> l1 = new ArrayList<>();
- B. List<String> l2 = new ArrayList();
- C. List<? super Object> l3 = new ArrayList<String>();
- D. List<? extends Object> l4 = new ArrayList<String>();

4. Given:

```
List<? super Number> list = new ArrayList<Object>(); // 1
list.add(new Integer(2)); // 2
list.add(new Object()); // 3
```

Which line will generate a compile-time error?

- A. Line marked as // 1
- B. Line marked as // 2
- C. Line marked as // 3
- D. No compile-time error is generated

ANSWERS

1. The correct answer is A.

You can modify a list typed as `List<?>`, but when you call `m2()`, inside that method, the list has a different type, `List<T>`, which allows you to change the value of the second element.

2. The correct answer is A.

`<T extends Number>` means that you can assign a `Number` or one of its subclasses to the generic type. This applies to both, the type argument of the class `Question_6_2` and to the variable `t`.

3. The correct answer is C.

Option C doesn't compile because you can only assign a list of type `Object` to `l3`, the lower-bounded wildcard doesn't allow you to assign a subclass (like `String`).

4. The correct answer is C.

`<? super Number>` allows you to assign a `List` of type `Number` or its superclass, which is done in line `// 1`.

The list can contain instances of `Number` or one of its subclasses. Since line `// 3` tries to add an instance of the superclass of `Number` (`Object`), this line generates a compile-time error.

Chapter SEVEN

Collections

Exam Objectives

- Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects.

COLLECTIONS OVERVIEW

A *collection* is a generic term that refers to a container of objects.

The *Java Collections Framework* is a library of classes and interfaces in the `java.util` package that provides collections with different characteristics.

The most important interfaces are:

- **Collection**
This is the base interface of the collection hierarchy and it contains methods like `add()`, `remove()`, `clear()`, and `size()`.
- **Iterable**
Implementing this interface allows an object to be "iterable" with a for-each loop, through an `Iterator`, and with the new `forEach()` method.
- **List**
Interface for collections which, one, store a group of elements that can be accessed using an index, and two, accept duplicates.
- **Set**
Interface for collections which do not allow duplicate elements.
- **Queue**
Interface for collections which store a group of elements in a particular order, commonly in a first-in, first-out order.
- **Map**
Interface for collections whose elements are stored as key/value pairs.

Of the last four, `Map` is the only one that implements neither the `Collection` nor the `Iterable` interface, but still, it's considered a collection, because it contains a group elements.

LIST

List is the most common collection. You use it when you want to allow (or do not care if there are) duplicate elements. You can even insert null elements (not all collections allow it).

Elements have an insertion order, but you can add elements at any position, since this position is based on a zero-based index, like an array.

In fact, the most used implementation, ArrayList, is actually implemented as an Object array under the hood.

The difference with using an array is that a List can grow automatically when elements are added. However, since it does that by copying the elements to a bigger array, adding (and removing) is slower.

Here are some basic List operations:

```
// Creating an ArrayList with an initial capacity of 10
List<String> list = new ArrayList<>(10);

System.out.println(list.isEmpty()); // true
list.add("a");
System.out.println(list.get(0)); // a
list.add(0, "b"); // Inserting b at index 0
list.add("c");
list.add(null);
System.out.println(list); // [b, a, c, null]
list.set(2, "a"); // Replacing element at index 2 with a
System.out.println(list.contains("d")); // false
```

```
// Returning the index of the first match, -1 if not found
System.out.println(list.indexOf("a")); // 1
// Returning the index of the last match, -1 if not found
System.out.println(list.lastIndexOf("a")); // 2

list.remove(1); // Removing by index
list.remove(null); // Removing null
list.remove("a") // Removing the first matching element

System.out.println(list.size()); // 1
```

Another popular implementation is `LinkedList`, a doubly-linked list that also implements the `Deque` interface (more about this interface later).

An easy way to create a `List` is using the `java.util.Arrays.asList` method:

```
String[] arr = {"a", "b", "c", "d"};
List<String> list = Arrays.asList(arr);
```

Or simply:

```
List<String> list = Arrays.asList("a", "b", "c", "d");
```

It returns an implementation of `List` backed by the specified array (but it's not an `ArrayList` and it doesn't implement all methods of `List`) that has fixed size, which means that you can't add elements to it. Also, modifications to the `List` are reflected in the original array.

SET

The main feature of a Set is that it doesn't allow duplicates.

The two most used implementations are HashSet and TreeSet. The difference between them is that TreeSet sorts the elements, while HashSet doesn't guarantee the order or that the order will remain constant over time.

HashSet stores the elements in a hash table (using a HashMap instance). Because of that, elements are not kept in order, but adding and looking up elements is fast.

To retrieve objects and avoid duplicates, the elements have to implement the hashCode() and equals() methods.

Here's an example of HashSet:

```
// Creating a HashSet with an initial capacity of 10
Set<String> set = new HashSet<>(10);
// add() returns true if the element is not already in the set
System.out.println(set.add("b")); // true
System.out.println(set.add("x")); // true
System.out.println(set.add("h")); // true
System.out.println(set.add("b")); // false
System.out.println(set.add(null)); // true
System.out.println(set.add(null)); // false
System.out.println(set); // [null, b, x, h]
```

As you can see, HashSet accepts null values.

TreeSet stores the elements in a red-black tree data structure. That's why it keeps the elements sorted and guarantees $\log(n)$ time cost for adding, removing, looking up an element, and getting the size of the set.

To avoid duplicates, the elements have to implement the `equals()` method. For sorting, elements have to either implement the Comparable interface (the implementation of `compareTo()` has to be consistent with the implementation of the `equals()` method) or pass an implementation of Comparator in the constructor. Otherwise, an exception will be thrown.

Here's an example similar to the previous one implemented with TreeSet:

```
Set<String> set = new TreeSet<>();
System.out.println(set.add("b")); // true
System.out.println(set.add("x")); // true
System.out.println(set.add("h")); // true
System.out.println(set.add("b")); // false
System.out.println(set); // [b, h, x]
```

Since String implements Comparable, and its `compareTo()` method implements lexicographic ordering, a Set is ordered that way.

Notice that this example doesn't add null values. That's because TreeSet doesn't accept them. If you try to add null to a TreeSet, a NullPointerException will be thrown.

This is because when an element is added, it's compared (as a Comparable or with a Comparator) against other values to insert it in the correct order, but it can't do that with a null value.

QUEUE

In a Queue, elements are typically added and removed in a *FIFO* (first-in-first-out) way.

The most used implementation is `ArrayDeque`, which is backed by an array, that has the functionality of adding and removing elements from both the front (as a stack) and back (as a queue) of the queue, and not in any position like in an `ArrayList`. This class doesn't allow inserting of null values.

Besides having the methods of `Collection`, `ArrayDeque` has other methods that are unique to queues. We can classify these methods into two groups:

Methods that throw an exception if something goes wrong:

- `boolean add(E e)`
Adds an element to the end of the queue and returns true if successful or throws an exception otherwise.
- `E remove()`
Removes and returns the first element of the queue or throws an exception if it's empty.
- `E element()`
Returns the next element of the queue or throws an exception if it's empty.

Methods that return `null` if something goes wrong:

- `boolean offer(E e)`
Adds an element to the end of the queue and returns true if successful or false otherwise.
- `E poll()`
Removes and returns the first element of the queue or `null` if it's empty.

- E peek()

Returns the next element of the queue or null if it's empty.

For each operation, there's a version that throws an exception and another that returns false or null. For example, when the queue is empty, the remove() method throws an exception, while the poll() method returns null.

```
Queue<String> queue = new ArrayDeque<>();
System.out.println(queue.offer("a"));    // true    [a]
System.out.println(queue.offer("b"));    // true    [a, b]
System.out.println(queue.peek());        // a      [a, b]
System.out.println(queue.poll());        // a      [b]
System.out.println(queue.peek());        // b      [b]
System.out.println(queue.poll());        // b      []
System.out.println(queue.peek());        // null
```

You can also use this class as a stack, a data structure that order the elements in a LIFO (last-in-first-out), when you use the following methods:

```
void push(E e) // Adds elements to the front of the queue
E pop() // Removes and returns the next element
// or throws an exception if the queue is empty
```

Notice that these methods are not in the Queue interface:

```
ArrayDeque<String> stack = new ArrayDeque<>();
stack.push("a");    // [a]
stack.push("b");    // [b, a]
System.out.println(stack.peek()); // b [b, a]
System.out.println(stack.pop()); // b [a]
System.out.println(stack.peek()); // a [a]
System.out.println(stack.pop()); // a []
System.out.println(stack.peek()); // null
```

MAP

While a List uses an index for accessing its elements, a Map uses a key that can be of any type (usually a String) to obtain a value.

Therefore, a map cannot contain duplicate keys, and a key is associated with one value (which can be any object, even another map, or null).

The two most used implementations are HashMap and TreeMap. The difference between them is that TreeMap sorts the keys, but adds and retrieves keys in $\log(n)$ time while HashMap doesn't guarantee the order but adds and retrieves keys faster.

It is important that the objects used as keys have the methods equals() and hashCode() implemented.

Since Map doesn't implement Collection, its methods are different. Here's an example that shows the most important ones:

```
Map<String, Integer> map = new HashMap<>();

// Adding a key/value pair
System.out.println( map.put("oranges", 7) );           // null
System.out.println( map.put("apples", 5) );            // null
System.out.println( map.put("lemons", 2) );             // null
System.out.println( map.put("bananas", 7) );            // null

// Replacing the value of an existing key. Returns the old one
System.out.println( map.put("apples", 4) );            // 5

System.out.println( map.size() );                      // 4
```

```
// {oranges=7, bananas=7, apples=4, lemons=2}
System.out.println(map);

// Getting a value
System.out.println( map.get("oranges") );           // 7

// Testing if the map contains a key
System.out.println( map.containsKey("apples") );    // true
// Testing if the map contains a value
System.out.println( map.containsValue(5) );          // false

// Removing the key/value pair and returning the value
System.out.println( map.remove("lemons") );         // 2
// Returns null if it can't find the key
System.out.println( map.remove("lemons") );          // null

// Getting the keys as a Set
// (changes are reflected on the map and vice-versa)
Set<String> keys = map.keySet(); // [oranges, bananas, apples]

// Getting the values as a Collection
// (changes are reflected on the map and vice-versa)
Collection<Integer> values = map.values();        // [7, 7, 4]

// Removing all key/value pairs
map.clear();

System.out.println( map.isEmpty() );                 // true
```

If we change the implementation to TreeMap, the map will be stored in a red-black tree structure and sorted just like a TreeSet, either by a Comparator or Comparable, with the natural order of its key by default:

Chapter SEVEN

```
Map<String, Integer> map = new TreeMap<>();  
  
System.out.println( map.put("oranges", 7) ); // null  
System.out.println( map.put("apples", 5) ); // null  
System.out.println( map.put("lemons", 2) ); // null  
System.out.println( map.put("bananas", 7) ); // null  
  
// {apples=5 , bananas=7, lemons=2, oranges=7}  
System.out.println(map);  
  
// [apples, bananas, lemons, oranges]  
Set<String> keys = map.keySet();  
Collection<Integer> values = map.values(); // [5, 7, 2, 7]
```

Notice that because of the way the sort is done (again, just like TreeSet); a TreeMap cannot have a null value as a key:

```
Map<String, Integer> map = new TreeMap<>();  
  
map.put(null, 1); // throws NullPointerException!
```

However, a HashMap can:

```
Map<String, Integer> map = new HashMap<>();  
  
map.put(null, 1); // OK
```

KEY POINTS

- Collection
This is the base interface of the collection hierarchy and it contains methods like `add()`, `remove()`, `clear()`, and `size()`.
- Iterable
Implementing this interface allows an object to be "iterable" with a for-each loop, through an Iterator, and with the new `forEach()` method.
- List
Interface for collections which, one, store a group of elements that can be accessed using an index, and two, accept duplicates.
- Set
Interface for collections which do not allow duplicate elements.
- Queue
Interface for collections which store a group of elements in a particular order, commonly in a first-in, first-out order.
- Map
Interface for collections whose elements are stored as key/value pairs.

The following table compares the collections reviewed in this chapter:

Collection	Interface	Implements Collection?	Allows duplicates?	Allows null values?	Ordered?
ArrayList	List	Yes	Yes	Yes	Yes (Insertion Order)
HashSet	Set	Yes	No	Yes	No
TreeSet	Set	Yes	No	No	Yes (Natural order or by Comparator)
ArrayDeque	Queue Deque	Yes	Yes	No	Yes (FIFO or LIFO)
HashMap	Map	No	Just for values	Yes	No
TreeMap	Map	No	Just for values	No	Yes (Natural order or by Comparator)

SELF TEST

1. Given:

```
public class Question_7_1 {  
    public static void main(String[] args) {  
        ArrayDeque<Integer> deque = new ArrayDeque<Integer>();  
        deque.push(1);  
        deque.push(2);  
        deque.push(3);  
        deque.poll();  
        System.out.println(deque);  
    }  
}
```

What is the result?

- A. [1, 2, 3]
- B. [1, 2]
- C. [2, 1]
- D. An exception occurs at runtime

2. Which of the following options can throw a NullPointerException?

- A. TreeSet<String> s = new TreeSet<>();
 s.add(null);
- B. HashMap<String> m = new HashMap<>();
 m.put(null, null);
- C. ArrayList<String> arr = new ArrayList<>();
 arr.add(null);
- D. HashSet<String> s = new HashSet<String>();
 s.add(null);

3. Given:

```
public class Question_7_3 {  
    public static void main(String[] args) {  
        List<Integer> list = new ArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
        list.remove(1);  
        System.out.println(list);  
    }  
}
```

What is the result?

- A. [2, 3]
- B. [1, 3]
- C. [1, 2, 3]
- D. An exception occurs at runtime

4. Which of the following statements is true?

- A. HashSet is an implementation of Map.
- B. Objects used as values of a TreeMap are required to implement Comparable.
- C. Objects used as values of a TreeMap are required to implement the hashCode() method.
- D. Objects used as keys of a TreeMap are required to implement the hashCode() method.

ANSWERS

1. The correct answer is C.

`push()` inserts the element at the front of the deque. After pushing 1, 2, 3 the queue looks like [3, 2, 1].

`poll()` retrieves and removes the first element of this deque, 3 in this case.

2. The correct answer is A.

`TreeSet` doesn't allow null values because when you add an object, if no `Comparator` is passed to the constructor of the `TreeSet` (like in this case), this class assumes that the object implements `Comparable` and tries to call the `compareTo()` method.

3. The correct answer is B.

The `remove()` method has two versions, one that takes the index of the element to remove, and another that takes the object to remove.

Since we are passing an `int`, the version that takes an index is chosen and the second element is removed. If we want to remove the first element (with value 1), we have to call the `remove()` method like this: `list.remove(new Integer(1))`.

4. The correct answer is D.

`HashSet` is an implementation of `Set`. Objects used as keys of a `TreeMap` are required to implement the `hashCode()` method. Values are not required to implement anything.

Part THREE

Lambda Expressions

Chapter EIGHT

Functional Interfaces

Exam Objectives

- Create and use Lambda expressions.

SIMPLIFYING THINGS

Suppose we have a program that has a list of cars and we need to search for all compact cars.

We can have something like this to do the work:

```
List<Car> findCompactCars(List<Car> cars) {  
    List<Car> compactCars = new ArrayList<Car>();  
    for(Car car : cars) {  
        if(car.getType().equals(CarTypes.COMPACT)) {  
            compactCars.add(car);  
        }  
    }  
    return compactCars;  
}
```

Easy. But the next day, the users realize they also need to search for cars that cost more than 20,000 USD.

So we come up with something like this:

```
List<Car> findCompactCars(List<Car> cars) {  
    List<Car> twentyKCars = new ArrayList<Car>();  
    for(Car car : cars) {  
        if(car.getCostUSD() > 20000) {  
            twentyKCars.add(car);  
        }  
    }  
    return twentyKCars;  
}
```

Chapter EIGHT

Now look at the code. It's practically the same. The only differences are the lines:

```
car.getType().equals(CarTypes.COMPACT)
```

And:

```
car.getCostUSD() > 20000
```

What if we need to filter by another condition in the future?

It's wrong to have duplicate or copy-pasted code, it's not very flexible to change and error-prone.

THE OBJECT-ORIENTED APPROACH

Since Java is an object-oriented let's leverage this by solving the problem using a popular design pattern, Strategy.

This pattern does just what we need, encapsulate the behavior that varies (algorithm) and makes all these behaviors interchangeable.

The recommended way to implement the Strategy pattern in Java is with an interface so we can create implementations with for each algorithm (in our case, the conditions to test)

So let's code an interface that can contain the search condition that varies, for example:

```
interface Searchable {  
    boolean test(Car car);  
}
```

And place the code that varies into implementations of that interface:

```
class CompactCarSearch implements Searchable {  
    public boolean test(Car car) {  
        return car.getType().equals(CarTypes.COMPACT);  
    }  
}
```

Chapter EIGHT

And:

```
class TwentyKCarSearch implements Searchable {  
    public boolean test(Car car) {  
        return car.getCostUSD() > 20000;  
    }  
}
```

That way, we can make the method that does the actual search a little more general:

```
List<Car> findCars(List<Car> cars, Searchable s) {  
    List<Car> searchedCars = new ArrayList<Car>();  
    for(Car car : cars) {  
        if(s.test(car)) {  
            searchedCars.add(car);  
        }  
    }  
    return searchedCars;  
}
```

And call the function in this way:

```
List<Car> compactCars = findCars(cars,  
                                  new CompactCarSearch());  
List<Car> twentyKCars = findCars(cars,  
                                  new TwentyKCarSearch());
```

We don't have duplicate code anymore.

If the user wants another way to search for a car, instead of copy-pasting a search method, now we just implement another class with a condition.

We are now more flexible to change.

But it's still a **COMPLEX** solution. Instead of having two methods we have two classes and one interface.

What if we turn the implementations into anonymous classes?
For example:

```
List<Car> compactCars = findCars(cars, new Searchable() {  
    public boolean test(Car car) {  
        return car.getType().equals(CarTypes.COMPACT);  
    }  
});
```

A little bit better, we can even assign the class to a variable if we want to use it in other parts of our program:

```
Searchable compactCarSearch = new Searchable() {  
    public boolean test(Car car) {  
        return car.getType().equals(CarTypes.COMPACT);  
    }  
};  
List<Car> compactCars = findCars(cars, compactCarSearch);
```

Not bad, but wouldn't it be great to just pass the condition to the method? Like this for example:

```
// Don't even try, it won't work  
List<Car> compactCars = findCars(cars,  
    car.getType().equals(CarTypes.COMPACT));
```

A NEW TYPE OF VALUE

Basically, in Java we work with two types of value, primitive values, and references to objects.

Both can be used as the arguments of a method:

```
method1(3);  
method2(new Object());
```

So if we want to pass some piece of code to a method, we have to wrap it in an object (like in the previous example).

But in Java 8, we can pass that piece of code directly through the use of a lambda expression.

Coincidentally, to use a lambda expression instead of an object in the previous example, we have to start from the same interface:

```
interface Searchable {  
    boolean test(Car car);  
}
```

A functional interface.

A FUNCTIONAL INTERFACE

The starting point to learn about lambda expressions is learning about functional interfaces.

A functional interface is any interface that has exactly **ONE ABSTRACT** method.

This is a tricky definition. Many people think that just having one method makes an interface functional (since interface methods are abstract by default), but it doesn't.

Take for example default methods. Since default methods have an implementation, they are not abstract. So interfaces like the following, are considered functional:

```
interface A {  
    default int defaultMethod() {  
        return 0;  
    }  
    void method();  
}  
  
interface B {  
    default int defaultMethod() {  
        return 0;  
    }  
    default int anotherDefaultMethod() {  
        return 0;  
    }  
    void method();  
}
```

Chapter EIGHT

If an interface declares an abstract method with the signature of one of the methods of `java.lang.Object`, it doesn't count toward the functional interface method count since any implementation of the interface will have an implementation of the method (since all classes extend from `java.lang.Object`).

So an interface like the following is considered functional:

```
interface A {  
    boolean equals(Object o);  
    int hashCode();  
    String toString();  
    void method();  
}
```

A more confusing scenario is when an interface inherits a method that is *equivalent* but not *identical* to another:

```
interface A {  
    void method(List<Double> l);  
}  
interface B extends A {  
    void method(List l);  
}
```

In this case, the method is the same, so it's taken as one method. The class that implements B will have to implement the method that can legally override all the abstract methods:

```
void method(List l);
```

By the way, neither is an empty interface considered functional.

The key here is to have **EXACTLY ONE ABSTRACT** method; that's why these interfaces are also called Single Abstract Method interfaces (SAM Interfaces).

To make things easier, Java 8 also introduced the @FunctionalInterface annotation, which generates a compile-time error when the interface you have annotated is not a valid functional interface.

```
// This won't compile
@FunctionalInterface
interface A {
    void m(int i);
    void m(long l);
}
```

Java interfaces that only have one method declared in their definition are now annotated as functional interfaces. Some examples are:

- java.lang.Runnable
- java.util.Comparator
- java.util.concurrent.Callable
- java.awt.event.ActionListener

But remember, this annotation is just to help you; having it won't make an interface functional.

In the next chapter, we will see how functional interfaces and lambda expressions are related.

KEY POINTS

- A functional interface is any interface that has exactly one abstract method.
- Since default methods have an implementation, they are not abstract so that a functional interface can have any number of them.
- If an interface declares an abstract method with the signature of one of the methods of `java.lang.Object`, it doesn't count toward the functional interface method count.
- A functional interface is valid when it inherits a method that is *equivalent* but not *identical* to another.
- An empty interface is not considered a functional interface.
- A functional interface is valid even if the `@FunctionalInterface` annotation would be omitted.
- Functional interfaces are the basis of lambda expressions.

SELF TEST

1. Given:

```
interface A {  
    default int m1() {  
        return 0;  
    }  
}  
  
@FunctionalInterface  
public interface B extends A {  
    static String m() {  
        return "static";  
    }  
}
```

Which of the following statements are true?

- A. Compilation fails
- B. It compiles successfully
- C. It compiles only if interface B declares a default method
- D. An exception occurs at runtime if this interface is implemented by a class

Chapter EIGHT

2. Given:

```
@FunctionalInterface  
interface C {  
    int m(int i);  
    long m(long i);  
}
```

Which of the following statements are true?

- A. This code compiles successfully
- B. If we remove the annotation, this code will compile
- C. If we remove one method, this code will compile
- D. The @FunctionalInterface annotation makes this interface functional

3. Given:

```
public interface D {  
    int sum(int a, int b);  
    default int subtract(int a, int b) {  
        return a - b;  
    }  
}
```

Which of the following statements are true?

- A. This code compiles successfully
- B. This code doesn't compile
- C. This is an example of a functional interface
- D. Removing the sum method would make this interface functional

4. Which of the following interfaces of the Java API can be considered functional?

- A. `java.util.concurrent.Callable`
- B. `java.util.Map`
- C. `java.util.Iterator`
- D. `java.lang.Comparable`

5. Given:

```
public interface E {  
    static int sum(int a, int b) {  
        return a + b;  
    }  
}
```

Which of the following statements are true?

- A. This code doesn't compile
- B. This code compiles successfully
- C. This is an example of a functional interface
- D. Adding the `@FunctionalInterface` annotation would make this interface functional

ANSWERS

1. The correct answer is A.

A functional interface must declare one abstract method. Since interface B only declares a static method and inherits a default one, it won't compile, not even by adding another default method or by having the `@FunctionalInterface` annotation.

2. The correct answers are B and C.

The code doesn't compile successfully. This is not a functional interface because it has two abstract methods (it doesn't matter they have the same name).

The `@FunctionalInterface` annotation throws an error if the interface it marks is not a functional one. If we remove it, the code compiles although, the interface is still not functional.

If we remove one method, no matter which, the interface becomes functional, making the code compile.

The `@FunctionalInterface` annotation doesn't make an interface functional; its job is just to make the compiler check if the interface is a functional one.

3. The correct answers are A and C.

There's nothing wrong with the code; it compiles successfully.

Functional interfaces can have any number of default methods,

as long as they have only one abstract method. In this case, the code represents a functional interface. If we remove the `sum` method, we would be violating this rule and the interface wouldn't be functional anymore.

4. The correct answers are A and D.

`java.util.concurrent.Callable` is marked with the `@FunctionalInterface` annotation.

Although `java.lang.Comparable` is not marked with the `@FunctionalInterface` annotation, it can be considered one. Remember, this annotation doesn't make an interface functional.

5. The correct answer is B.

There's nothing wrong with this code; it compiles successfully. However, this is not a functional interface since it doesn't have an abstract method. Therefore, all other options are false.

Chapter EIGHT

Chapter NINE

Lambda Expressions

Exam Objectives

- Create and use Lambda expressions

WHAT IS A LAMBDA EXPRESSION?

In the previous chapter, we saw how passing a piece of code instead of a whole object is very useful.

Java 8 has introduced a notation that enables you to write:

```
List<Car> compactCars = findCars(cars,  
        (Car c) -> car.getType().equals(CarTypes.COMPACT)  
);
```

Instead of:

```
List<Car> compactCars = findCars(cars, new Searchable() {  
    public boolean test(Car car) {  
        return car.getType().equals(CarTypes.COMPACT);  
    }  
});
```

These are lambda expressions.

The term lambda expression comes from lambda calculus, written as λ -calculus, where λ is the Greek letter lambda. This form of calculus deals with defining and applying functions.

With lambdas, you won't be able to do things that you couldn't do before them, but they allow you to program in a more simple way with a style called functional programming, a different paradigm than object-oriented programming.

A LAMBDA EXPRESSION

PARAMETERS

(**Object** arg1, **int** arg2) ->

ARROW

BODY

arg1.equals(arg2);

Chapter NINE

A lambda expression has three parts:

A list of parameters

A lambda expression can have zero (represented by empty parentheses), one or more parameters:

```
() -> System.out.println("Hi");  
(String s) -> System.out.println(s);  
(String s1, String s2) -> System.out.println(s1 + s2);
```

The type of the parameters can be declared explicitly, or it can be inferred from the context:

```
(s) -> System.out.println(s);
```

If there is a single parameter, the type is inferred and it is not mandatory to use parentheses:

```
s -> System.out.println(s);
```

If the lambda expression uses a parameter name which is the same as a variable name of the enclosing context, a compile error is generated:

```
String s = "";  
s -> System.out.println(s); // This doesn't compile
```

An arrow

Formed by the characters - and > to separate the parameters and the body.

A body

The body of the lambda expressions can contain one or more statements.

If the body has one statement, curly brackets are not required and the value of the expression (if any) is returned:

```
() -> 4;  
(int a) -> a*6;
```

If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must be returned with a return statement:

```
() -> {  
    System.out.println("Hi");  
    return 4;  
}  
(int a) -> {  
    System.out.println(a);  
    return a*6;  
}
```

If the lambda expression doesn't return a result, a return statement is optional. For example, the following expressions are equivalent:

```
() -> System.out.println("Hi");  
() -> {  
    System.out.println("Hi");  
    return;  
}
```

HOW ARE FUNCTIONAL INTERFACES RELATED TO ALL THIS?

The signature of the abstract method of a functional interface provides the signature of a lambda expression (this signature is called a *functional descriptor*).

This means that to use a lambda expression, you first need a functional interface. For example, using the interface of the previous chapter:

```
interface Searchable {  
    boolean test(Car car);  
}
```

We can create a lambda expression that takes a Car object as argument and returns a boolean:

```
Searchable s = (Car c) -> c.getCostUSD() > 20000;
```

In this case, the compiler inferred that the lambda expression can be assigned to a Searchable interface, just by its signature.

In fact, lambda expressions don't contain information about which functional interface they are implementing. The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.

In the previous example, as the lambda expression is being assigned to a `Searchable` interface, the lambda must take the signature of its abstract method. Otherwise, a compiler error is generated.

If we were using the lambda as an argument to a method, the compiler would use the definition of the method to infer the type of the expression:

```
class Test {  
    public void find() {  
        find(c -> c.getCostUSD() > 20000);  
    }  
  
    private void find(Searchable s) {  
        // Here goes the implementation  
    }  
}
```

Because of this, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature. For example:

```
interface Searchable {  
    boolean test(Car car);  
}  
interface Saleable {  
    boolean approve(Car car);  
}  
//...  
Searchable s1 = c -> c.getCostUSD() > 20000;  
Saleable s2 = c -> c.getCostUSD() > 20000;
```

Chapter NINE

For reference, the contexts where the target type (the functional interface) of a lambda expression can be inferred are:

- A variable declaration
- An assignment
- A return statement
- An array initializer
- Method or constructor arguments
- A ternary conditional expression
- A cast expression

However, if you understand the concept, you don't need to memorize this list.

But what is it all about functional programming, functional descriptors, functional whatever?

A lambda expression is a function. That's a little different than the methods we know because it's actually related to a *mathematical function*.

A mathematical function takes some inputs and produces some outputs, but **HAS NO SIDE EFFECTS**, meaning that as long as we call it with the same arguments, it always returns the same result.

Of course, in Java, you can't always program in a purely functional style (i.e. without any side effect), only in a *functional-style*.

So for practical terms, it may be safe to think of lambda expressions as anonymous methods (or functions), as they don't have a name, just like anonymous classes.

HOW ARE LAMBDA EXPRESSIONS RELATED TO ANONYMOUS CLASSES?

Lambda expressions are an **ALTERNATIVE** to anonymous classes, but they are not the same.

They have some similarities:

- Local variables (variables or parameters defined in a method) can only be used if they are declared final or are effectively final.
- You can access instance or static variables of the enclosing class.
- They must not throw more checked exceptions than specified in the throws clause of the functional interface method. Only the same type or a subtype.

And some significant differences:

- For an anonymous class, this keyword resolves to the anonymous class itself. For a lambda expression, it resolves to the enclosing class where the lambda is written.
- Default methods of a functional interface cannot be accessed from within lambda expressions. Anonymous classes can.
- Anonymous classes are compiled into, well, inner classes. But lambda expressions are converted into private static (in some cases) methods of their enclosing class and, using the `invokedynamic` instruction (added in Java 7), they are bound dynamically. Since there's no need to load another class, lambda expressions are more efficient. This is a very simple explanation, but that's the idea.

By the way, if you really have to know or want to understand why local variables have to be final whereas instance or static variables don't when using them inside an anonymous class or lambda expression, it's because of the way these variables are implemented in Java.

Instance variables are stored on the heap (a region of your computer's memory accessible anywhere in your program), while local variables live on the stack (a special region of your computer's memory that stores temporary variables created by each function).

Variables on the heap are shared by across threads, but variables on the stack are implicitly confined to the thread they're in.

So when you create an instance of an anonymous inner class or a lambda expression, the values of local variables are **COPIED**. In other words, you're not working with the variable, but with a copied value.

First of all, this prevents thread-related problems. Second, since you're working with a copy, if the variable could be modified by the rest of the method, you would be working with an out-of-date variable.

A final (or effectively final) variable removes these possibilities. Since the value can't be changed, you don't need to worry about such changes being visible or causing thread problems.

KEY POINTS

- Lambda expressions have three parts: a list of parameters, and arrow, and a body:

```
(Object o) -> System.out.println(o);
```

- You can think of lambda expressions as anonymous methods (or functions) as they don't have a name.
- A lambda expression can have zero (represented by empty parentheses), one or more parameters.
- The type of the parameters can be declared explicitly, or it can be inferred from the context.
- If there is a single parameter, the type is inferred and is not mandatory to use parentheses.
- If the lambda expression uses as a parameter name which is the same as a variable name of the enclosing context, a compile error is generated.
- If the body has one statement, curly brackets are not required, and the value of the expression (if any) is returned.
- If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must return with a return statement.
- If the lambda expression doesn't return a result, a return statement is optional.
- The signature of the abstract method of a functional interface provides the signature of a lambda expression

(this signature is called a *functional descriptor*).

- This means that to use a lambda expression, you first need a functional interface.
- However, lambda expressions don't contain the information about which functional interface are implementing.
- The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.
- The contexts where the target type of a lambda expression can be inferred include an assignment, method or constructor arguments, and a cast expression.
- Like anonymous classes, lambda expressions can access instance and static variables, but only final or effectively final local variables.
- Also, they cannot throw exceptions that are not defined in the throws clause of the function interface method.
- For a lambda expression, this resolves to the enclosing class where the lambda is written.
- Default methods of a functional interface cannot be accessed from within lambda expressions.

SELF TEST

1. Which of the following are valid lambda expressions?

- A. String a, String b -> System.out.print(a+ b);
- B. () -> return;
- C. (int i) -> i;
- D. (int i) -> i++; return i;

2. Given:

```
interface A {  
    int aMethod(String s);  
}
```

Which of the following are valid statements?

- A. A a = a -> a.length();
- B. A x = y -> {return y;};
- C. A s = "2" -> Integer.parseInt(s);
- D. A b = (String s) -> 1;

3. A lambda expression can be used...

- A. As a method argument
- B. As a conditional expression in an if statement
- C. In a return statement
- D. In a throw statement

4. Given:

```
( ) -> 7 * 12.0;
```

Which of the following interfaces can provide the functional descriptor for the above lambda expression?

A.

```
interface A {  
    default double m() {  
        return 4.5;  
    }  
}
```

B.

```
interface B {  
    Number m();  
}
```

C.

```
interface C {  
    int m();  
}
```

D.

```
interface D {  
    double m(Integer... i);  
}
```

Chapter NINE

5. Given:

```
interface AnInterface {  
    default int aMethod() { return 0; }  
    int anotherMethod();  
}  
  
public class Question_9_5 implements AnInterface {  
    public static void main(String[] args) {  
        AnInterface a = () -> aMethod();  
        System.out.println(a.anotherMethod());  
    }  
    @Override  
    public int anotherMethod() {  
        return 1;  
    }  
}
```

What is the result?

- A. 0
- B. 1
- C. Compilation fails
- D. An exception occurs at runtime

6. Which of the following statements are true?

- A. Curly brackets are required whenever the return keyword is used in a lambda expression
- B. A return keyword is always required in a lambda expression
- C. A return keyword is always optional in a lambda expression
- D. Lambda expressions don't return values

7. How is the `this` keyword handled inside a lambda expression?

- A. You can't use `this` inside a lambda expression
- B. `this` refers to the functional interface of the lambda expression
- C. `this` refers to the lambda expression itself
- D. `this` refers to the enclosing class of the lambda expression

8. Given:

```
interface X {  
    int test(int i);  
}  
public class Question_9_8 {  
    int i = 0;  
    public static void main(String[] args) {  
        X x = i -> i * 2;  
        System.out.println(x.test(3));  
    }  
}
```

What is the result?

- A. 0
- B. 3
- C. 6
- D. Compilation fails

ANSWERS

1. The correct answer is C.

Option A is invalid since the parameter list doesn't have parenthesis.

Option B is invalid since an empty return statement is invalid.

Option C is valid. The body doesn't need to use the return keyword if it only has one statement.

Option D is invalid since the body has two statements and they must be enclosed in curly brackets.

2. The correct answer is D.

Option A is invalid because a is already used as the variable identifier.

Option B is invalid because the lambda expression must return an int value.

Option C is invalid because you can't pass a constant value in the parameter list of the lambda expression.

Option D is valid because it takes a String argument and returns an int value.

3. The correct answers are A and C.

Lambda expressions can be used in:

- A variable declaration
- An assignment
- A return statement
- An array initializer
- As a method or constructor arguments

- A ternary conditional expression
- A cast expression

4. The correct answer is B.

Option A is invalid because the interface is not functional (it doesn't have an abstract method).

Option B is correct because the interface method doesn't take an argument and the type of the lambda expression can be cast to a `java.lang.Number` object.

Option C is invalid because a `double` can't be cast to an `int`.

Option D is invalid because the lambda expression's signature doesn't match the signature of the functional interface method `m(Integer[])`.

5. The correct answer is C.

A lambda expression can access the default methods of its functional interface (`aMethod()` in this case), since default methods are inherited. But it cannot be accessed from `main()` because it's a `static` method and has no `this` reference.

6. The only correct answer is A.

A `return` keyword is not always required (or optional) in a lambda expression. It depends on the signature of the functional interface method.

Chapter NINE

Curly brackets are required whenever the return keyword is used in a lambda expression. Both can be omitted if the lambda expression's body is just one statement:

```
() -> 2 * 3; // Valid  
() -> { return 2 * 3; }; // Valid  
() -> return 2 * 3; // Not Valid!
```

7. The correct answer is D.

For a lambda expression, this resolves to the enclosing class where the lambda is written.

8. The correct answer is C.

Lambda expressions can access instance and static variables, but in this case, the parameter of the lambda expression shadows the instance variable `i`, so the value of 6 is used.

Chapter TEN

Java Built-In Lambda Interfaces

Exam Objectives

- Use the built-in interfaces included in the `java.util.function` package such as `Predicate`, `Consumer`, `Function`, and `Supplier`.
- Develop code that uses primitive versions of functional interfaces.
- Develop code that uses binary versions of functional interfaces.
- Develop code that uses the `UnaryOperator` interface.

WHY BUILT-IN INTERFACES?

A lambda expression must correspond to one functional interface.

We can use any interface as a lambda expression as long as the interface only contains one abstract method.

We also saw that the Java 8 API has many functional interfaces that we can use to construct lambda expressions, like `java.lang.Runnable` or `java.lang.Comparable`.

However, Java 8 contains new functional interfaces to work specifically with lambda expressions, covering the most common scenarios usages.

For example, two common scenarios are to filter things based on a particular condition and test for some condition on the properties of an object.

In the previous chapters, we used:

```
interface Searchable {  
    boolean test(Car c);  
}
```

But the problem is that we have to write an interface like that in each program that uses it (or link a library that contains it).

Luckily, an interface that does the same but accepts any object type already exists in the language.

Chapter TEN

The new functional interfaces are located inside the `java.util.function` package.

There are five of them:

- **Predicate<T>**
- **Consumer<T>**
- **Function<T, R>**
- **Supplier<T>**
- **UnaryOperator<T>**

Where T and R represent generic types (T represents a parameter type and R the return type).

Also, they also have specializations for the cases where the input parameter is a primitive type (actually just for int, long, double, and boolean just in the case of Supplier), for example:

- **IntPredicate**
- **LongConsumer**
- **BooleanSupplier**

Where the name is preceded by the appropriate primitive type.

Plus, four of them have binary versions, which means they take two parameters instead of one:

- **BiPredicate<L, R>**
- **BiConsumer<T, U>**
- **BiFunction<T, U, R>**
- **BinaryOperator<T>**

Where T, U, and R represent generic types (T and U represent parameter types and R the return type).

The tables on the following pages show the complete list of interfaces.

You don't have to memorize them, just try to understand them.

In the following pages, each interface will be explained.

Chapter TEN

Functional Interface	Primitive Versions
Predicate<T>	IntPredicate LongPredicate DoublePredicate
Consumer<T>	IntConsumer LongConsumer DoubleConsumer
Function<T, R>	IntFunction<R> IntToDoubleFunction IntToLongFunction LongFunction<R> LongToDoubleFunction LongToIntFunction DoubleFunction<R> DoubleToIntFunction DoubleToLongFunction ToIntFunction<T> ToDoubleFunction<T> ToLongFunction<T>
Supplier<T>	BooleanSupplier IntSupplier LongSupplier DoubleSupplier
UnaryOperator<T>	IntUnaryOperator LongUnaryOperator DoubleUnaryOperator

Functional Interface	Primitive Versions
BiPredicate<L, R>	
BiConsumer<T, U>	ObjIntConsumer<T> ObjLongConsumer<T> ObjDoubleConsumer<T>
BiFunction<T, U, R>	ToIntBiFunction<T, U> ToLongBiFunction<T, U> ToDoubleBiFunction<T, U>
BinaryOperator<T>	IntBinaryOperator LongBinaryOperator DoubleBinaryOperator

PREDICATE

A predicate is a statement that may be true or false depending on the values of its variables.

This functional interface can be used anywhere you need to evaluate a boolean condition.

This how the interface is defined:

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
  
    // Other default and static methods  
    // ...  
}
```

So the functional descriptor (method signature) is:

T -> boolean

Here's an example using an anonymous class:

```
Predicate<String> startsWithA = new Predicate<String>() {
    @Override
    public boolean test(String t) {
        return t.startsWith("A");
    }
};
boolean result = startsWithA.test("Arthur");
```

And with a lambda expression:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
boolean result = startsWithA.test("Arthur");
```

This interface also has the following default methods:

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
default Predicate<T> negate()
```

These methods return a composed Predicate that represents a short-circuiting logical **AND** and **OR** of this predicate and another and its logical negation.

Short-circuiting means that the other predicate won't be evaluated if the value of the first predicate can predict the result of the operation (if the first predicate returns false in the case of **AND** or if it returns true in the case of **OR**).

Chapter TEN

These methods are useful to combine predicates and make the code more readable, for example:

```
Predicate<String> startsWithA = t -> t.startsWith("A");
Predicate<String> endsWithA = t -> t.endsWith("A");
boolean result = startsWithA.and(endsWithA).test("Hi");
```

Also there's a static method:

```
static <T> Predicate<T> isEqual(Object targetRef)
```

That returns a Predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.

There are also primitive versions for `int`, `long` and `double`. They don't extend from `Predicate<T>`.

For example, here's the definition of `IntPredicate`:

```
@FunctionalInterface
public interface IntPredicate {
    boolean test(int value);

    // And the default methods: and, or, negate
}
```

So instead of using:

```
Predicate<Integer> even = t -> t % 2 == 1;  
boolean result = even.test(5);
```

You can use:

```
IntPredicate even = t -> t % 2 == 1;  
boolean result = even.test(5);
```

Why?

Just to avoid the conversion from Integer to int and work directly with primitive types.

Notice that these primitive versions don't have a generic type. Due to the way generics are implemented, parameters of the functional interfaces can be bound only to object types.

Since the conversion from the wrapper type (Integer) to the primitive type (int) uses more memory and comes with a performance cost, Java provides these versions to avoid autoboxing operations when inputs or outputs are primitives.

CONSUMER

A consumer is an operation that accepts a single input argument and returns no result; it just execute some operations on the argument.

This how the interface is defined:

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
  
    // And a default method  
    // ...  
}
```

So the functional descriptor (method signature) is:

T -> void

Here's an example using an anonymous class:

```
Consumer<String> consumeStr = new Consumer<String>() {
    @Override
    public void accept(String t) {
        System.out.println(t);
    }
};
consumeStr.accept("Hi");
```

And with a lambda expression:

```
Consumer<String> consumeStr = t -> System.out.println(t);
consumeStr.accept("Hi");
```

This interface also has the following default method:

```
default Consumer<T> andThen(Consumer<? super T> after)
```

This method returns a composed Consumer that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

Chapter TEN

These methods are useful to combine Consumers and make the code more readable, for example:

```
Consumer<String> first = t ->
    System.out.println("First:" + t);
Consumer<String> second = t ->
    System.out.println("Second:" + t);
first.andThen(second).accept("Hi");
```

The output is:

```
First: Hi
Second: Hi
```

Look how both Consumers take the same argument and the order of execution.

There are also primitive versions for int, long and double. They don't extend from Consumer<T>.

For example, here's the definition of IntConsumer:

```
@FunctionalInterface
public interface IntConsumer {
    void accept(int value);

    default IntConsumer andThen(IntConsumer after) {
        // ...
    }
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};  
printList(a, t -> System.out.println(t));  
//...  
void printList(int[] a, Consumer<Integer> c) {  
    for(int i : a) {  
        c.accept(i);  
    }  
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};  
printList(a, t -> System.out.println(t));  
//...  
void printList(int[] a, IntConsumer c) {  
    for(int i : a) {  
        c.accept(i);  
    }  
}
```

FUNCTION

A function represents an operation that takes an input argument of a certain type and produces a result of another type.

A common use is to convert or transform from one object to another.

This how the interface is defined:

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T t);  
  
    // Other default and static methods  
    // ...  
}
```

So the functional descriptor (method signature) is:

$T \rightarrow R$

Assuming a method:

```
void round(double d, Function<Double, Long> f) {
    long result = f.apply(d);
    System.out.println(result);
}
```

Here's an example using an anonymous class:

```
round(5.4, new Function<Double, Long>() {
    Long apply(Double d) {
        return Math.round(d);
    }
});
```

And with a lambda expression:

```
round(5.4, d -> Math.round(d));
```

This interface also has the following default methods:

```
default <V> Function<V,R> compose(
    Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(
    Function<? super R,? extends V> after)
```

The difference between these methods is that `compose` applies the function represented by the parameter first, and its result serves as the input to the other function. `andThen` first applies the function that calls the method, and its result acts as the input of the function represented by the parameter.

Chapter TEN

For example:

```
Function<String, String> f1 = s -> {
    return s.toUpperCase();
};

Function<String, String> f2 = s -> {
    return s.toLowerCase();
};

System.out.println(f1.compose(f2).apply("Compose"));
System.out.println(f1.andThen(f2).apply("AndThen"));
```

The output is:

```
COMPOSE
andthen
```

In the first case, f1 is the last function to be applied.

In the second case, f2 is the last function to be applied.

Also there's a static method:

```
static <T> Function<T, T> identity()
```

That returns a function that always returns its input argument.

In the case of primitive versions, they also apply to int, long and double, but there are more combinations than the previous interfaces:

- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, IntFunction<R>:

```
@FunctionalInterface  
public interface IntFunction<R> {  
    R apply(int value);  
}
```

- To indicate that the function returns a primitive type and takes a generic argument, the interface is named **ToXXXFunction**, for example,ToIntFunction<T>:

```
@FunctionalInterface  
public interface ToIntFunction<T> {  
    int applyAsInt(T value);  
}
```

- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where XXX is the argument type and YYY is the return type, for example, IntToDoubleFunction:

```
@FunctionalInterface  
public interface IntToDoubleFunction {  
    double applyAsDouble(int value);  
}
```

Remember that these interfaces are for convenience, to work directly with primitives, for example:

```
DoubleFunction<R> instead of Function<Double, R>  
ToLongFunction<T> instead of Function<T, Long>  
IntToLongFunction instead of Function<Integer, Long>
```

SUPPLIER

A supplier does the opposite of a consumer, it takes no arguments and only returns some value.

This how the interface is defined:

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

So the functional descriptor (method signature) is:

`() -> T`

Here's an example using an anonymous class:

```
String t = "One";
Supplier<String> supplierStr = new Supplier<String>() {
    @Override
    public String get() {
        return t.toUpperCase();
    }
};
System.out.println(supplierStr.get());
```

And with a lambda expression:

```
String t = "One";
Supplier<String> supplierStr = () -> t.toUpperCase();
System.out.println(supplierStr.get());
```

This interface doesn't define default methods.

There are also primitive versions for int, long and double and boolean. They don't extend from Supplier<T>.

For example, here's the definition of BooleanSupplier:

```
@FunctionalInterface
public interface BooleanSupplier {
    boolean getAsBoolean();
}
```

That can be used instead of Supplier<Boolean>.

UNARYOPERATOR

UnaryOperator is just a specialization of the Function interface (in fact, this interface extends from it) for when the argument and the result are of the same type.

This how the interface is defined:

```
@FunctionalInterface  
public interface UnaryOperator<T>  
    extends Function<T, T> {  
    // Just the identity  
    // method is defined  
}
```

So the functional descriptor (method signature) is:

$T \rightarrow T$

Here's an example using an anonymous class:

```
UnaryOperator<String> uOp = new UnaryOperator<String>() {
    @Override
    public String apply(String t) {
        return t.substring(0,2);
    }
};
System.out.println(uOp.apply("Hello"));
```

And with a lambda expression:

```
UnaryOperator<String> uOp = t -> t.substring(0,2);
System.out.println(uOp.apply("Hello"));
```

This interface inherits the default methods of the Function interface:

```
default <V> Function<V,R> compose(
    Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(
    Function<? super R,? extends V> after)
```

And just defines the static method `identity()` for this interface (since static methods are not inherited):

```
static <T> UnaryOperator<T> identity()
```

That returns an `UnaryOperator` that always returns its input argument.

Chapter TEN

There are also primitive versions for `int`, `long` and `double`. They don't extend from `UnaryOperator<T>`.

For example, here's the definition of `IntUnaryOperator`:

```
@FunctionalInterface
public interface IntUnaryOperator {
    int applyAsInt(int value);

    // Definitions for compose, andThen, and identity
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};
int sum = sumNumbers(a, t -> t * 2);
//...
int sumNumbers(int[] a, UnaryOperator<Integer> unary) {
    int sum = 0;
    for(int i : a) {
        sum += unary.apply(i);
    }
    return sum;
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};  
int sum = sumNumbers(a, t -> t * 2);  
//...  
int sumNumbers(int[] a, IntUnaryOperator unary) {  
    int sum = 0;  
    for(int i : a) {  
        sum += unary.applyAsInt(i);  
    }  
    return sum;  
}
```

BiPREDICATE

This interface represents a predicate that takes two arguments.

This how the interface is defined:

```
@FunctionalInterface  
public interface BiPredicate<T, U> {  
    boolean test(T t, U u);  
    // Default methods are defined also  
}
```

So the functional descriptor (method signature) is:

(T, U) -> boolean

Here's an example using an anonymous class:

```
BiPredicate<Integer, Integer> divisible =
        new BiPredicate<Integer, Integer>() {
    @Override
    public boolean test(Integer t, Integer u) {
        return t % u == 0;
    }
};
boolean result = divisible.test(10, 5);
```

And with a lambda expression:

```
BiPredicate<Integer, Integer> divisible =
        (t, u) -> t % u == 0;
boolean result = divisible.test(10, 5);
```

This interface defines the same default methods of the Predicate interface, but with two arguments:

```
default BiPredicate<T, U> and(
        BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> or(
        BiPredicate<? super T, ? super U> other)
default BiPredicate<T, U> negate()
```

This interface **doesn't** have primitive versions.

BiConsumer

This interface represents a consumer that takes two arguments (and don't return a result).

This how the interface is defined:

```
@FunctionalInterface  
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
    // andThen default method is defined  
}
```

So the functional descriptor (method signature) is:

(T, U) -> void

Here's an example using an anonymous class:

```
BiConsumer<String, String> consumeStr =
        new BiConsumer<String, String>() {
    @Override
    public void accept(String t, String u) {
        System.out.println(t + " " + u);
    }
};
consumeStr.accept("Hi", "there");
```

And with a lambda expression:

```
BiConsumer<String> consumeStr =
        (t, u) -> System.out.println(t + " " + u);
consumeStr.accept("Hi", "there");
```

This interface also has the following default method:

```
default BiConsumer<T, U> andThen(
        BiConsumer<? super T, ? super U> after)
```

This method returns a composed BiConsumer that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

Chapter TEN

As in the case of a Consumer, these methods are useful to combine BiConsumers and make the code more readable, for example:

```
BiConsumer<String, String> first = (t, u) ->
    System.out.println(t.toUpperCase() + u.toUpperCase());
BiConsumer<String, String> second = (t, u) ->
    System.out.println(t.toLowerCase() + u.toLowerCase());
first.andThen(second).accept("Again", " and again");
```

The output is:

```
AGAIN AND AGAIN
again and again
```

There are also primitive versions for int, long and double. They don't extend from BiConsumer<T>, and instead of taking two ints, for example, they take one object and a primitive value as a second argument. So the naming convention changes to **ObjXXXConsumer**, where XXX is the primitive type.

For example, here's the definition of ObjIntConsumer:

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
    void accept(T t, int value);
}
```

So instead of using:

```
int[] a = {1,2,3,4,5,6,7,8};  
printList(a, (t, i) -> System.out.println(t + i));  
//...  
void printList(int[] a, BiConsumer<String, Integer> c) {  
    for(int i : a) {  
        c.accept("Number:", i);  
    }  
}
```

You can use:

```
int[] a = {1,2,3,4,5,6,7,8};  
printList(a, (t, i) -> System.out.println(t + i));  
//...  
void printList(int[] a, ObjIntConsumer<String> c) {  
    for(int i : a) {  
        c.accept("Number:", i);  
    }  
}
```

BiFUNCTION

This interface represents a function that takes two arguments of different type and produces a result of another type.

This how the interface is defined:

```
@FunctionalInterface  
public interface BiFunction<T, U, R> {  
    R apply(T t, U u);  
  
    // Other default and static methods  
    // ...  
}
```

So the functional descriptor (method signature) is:

$(T, U) \rightarrow R$

Assuming a method:

```
void round(  
    double d1, double d2, BiFunction<Double, Double, Long> f) {  
    long result = f.apply(d1, d2);  
    System.out.println(result);  
}
```

Here's an example using an anonymous class:

```
round(5.4, 3.8, new BiFunction<Double, Double, Long>() {  
    Long apply(Double d1, Double d2) {  
        return Math.round(d1 + d2);  
    }  
});
```

And with a lambda expression:

```
round(5.4, 3.8, (d1, d2) -> Math.round(d1, d2));
```

This interface, unlike Function, has only one default method:

```
default <V> Function<T, V> andThen(  
    Function<? super R, ? extends V> after)
```

That returns a composed function that first applies the function that calls andThen to its input, to finally apply the function represented by the argument to the result.

This interface also has less primitive versions than Function. It only has the versions that take generic types as arguments and return int, long and double primitive types, with the naming convention **ToXXXBiFunction<T, U>**, where XXX is the primitive type.

For example, here's the definition ofToIntBiFunction:

```
@FunctionalInterface  
public interface ToIntBiFunction<T, U> {  
    int applyAsInt(T t, U u);  
}
```

That replaces BiFunction<T, U, Integer>.

BINARYOPERATOR

This interface is a specialization of the BiFunction interface (in fact, this interface extends from it) for when the arguments and the result are of the same type.

This how the interface is defined:

```
@FunctionalInterface  
public interface BinaryOperator<T>  
    extends BiFunction<T, T, T> {  
    // Two static method are defined  
}
```

So the functional descriptor (method signature) is:

$(T, T) \rightarrow T$

Chapter TEN

Here's an example using an anonymous class:

```
BinaryOperator<String> binOp =  
    new BinaryOperator<String>() {  
        @Override  
        public String apply(String t, String u) {  
            return t.concat(u);  
        }  
    };  
System.out.println(binOp.apply("Hello", " there"));
```

And with a lambda expression:

```
BinaryOperator<String> binOp = (t, u) -> t.concat(u);  
System.out.println(binOp.apply("Hello", " there"));
```

This interface inherits the default method of the BiFunction interface:

```
default <V> Function<T, V> andThen(  
    Function<? super R, ? extends V> after)
```

And defines two new static methods:

```
static <T> BinaryOperator<T> minBy(  
    Comparator<? super T> comparator)  
static <T> BinaryOperator<T> maxBy(  
    Comparator<? super T> comparator)
```

That return a BinaryOperator, which returns the lesser or greater of two elements according to the specified Comparator.

Here's a simple example:

```
BinaryOperator<Integer> biOp =  
    BinaryOperator.maxBy(Comparator.naturalOrder());  
System.out.println(biOp.apply(28, 8));
```

As you can see, these methods are just a wrapper to execute a Comparator.

Comparator.naturalOrder() returns a Comparator that compares Comparable objects in natural order. To execute it, we just call the apply() method with the two arguments needed by the BinaryOperator. Unsurprisingly, the output is:

28

There are also primitive versions for int, long and double, where the two arguments and the return type are of the same primitive type. They don't extend from BinaryOperator or BiFunction.

For example, here's the definition of IntBinaryOperator:

```
@FunctionalInterface  
public interface IntBinaryOperator {  
    int applyAsInt(int left, int right);  
}
```

That you can use instead of BinaryOperator<Integer>.

KEY POINTS

- Java 8 contains new functional interfaces to work with lambda expressions that cover the most common scenarios usages located in the `java.util.function` package.
- They are:
 - `Predicate<T>`
 - `Consumer<T>`
 - `Function<T, R>`
 - `Supplier<T>`
 - `UnaryOperator<T>`
- These interfaces have versions that work with primitive values for `int`, `long`, `double` and `boolean` (only for `Supplier`) just to avoid the cost of converting a wrapper class to its primitive value, for example, `Integer` to `int`.
- These interfaces take one argument (represented by the generic type `T`), but with the exception of `Supplier` (that doesn't take any arguments), they have versions that take two arguments called binary versions.
- A `Predicate` can be used anywhere you need to evaluate a boolean condition. Its function descriptor (method signature) is:
`T -> boolean`
- It has primitive versions for `int`, `long` and `double`, for example, `IntPredicate`.

- A Consumer is an operation that accepts a single input argument and returns no result. Its functional descriptor is:
 $T \rightarrow void$
- It has primitive versions for int, long and double, for example, IntConsumer.
- A Function is an operation that takes an input argument of a certain type and produces a result of another type. Its functional descriptor is:
 $T \rightarrow R$
- It has a lot of primitive versions for int, long and double. We can divide them into three types.
- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, IntFunction<R>.
- To indicate that the function returns a primitive type and takes a generic argument, the interface is named **ToXXXFunction**, for example,ToIntFunction<T>.
- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where XXX is the argument type and YYY is the return type, for example, IntToDoubleFunction.
- A Supplier is an operation that takes no arguments, but it returns some value. Its functional descriptor is:
 $() \rightarrow T$

- It has primitive versions for int, long, double and boolean, for example, IntSupplier.
- UnaryOperator is a specialization of the Function interface (in fact, this interface extends from it) for when the argument and the result are of the same type. So its functional descriptor is:
 $T \rightarrow T$
- It has primitive versions for int, long and double, for example, IntUnaryOperator.
- A BiPredicate represents a predicate that takes two arguments. Its functional descriptor is:
 $(T, U) \rightarrow boolean$
- This interface doesn't have primitive versions.
- A BiConsumer represents a consumer that takes two arguments. Its functional descriptor is:
 $(T, U) \rightarrow void$
- It has primitive versions for int, long and double. They take one object and a primitive value as a second argument. So the naming conventions change to **ObjXXXConsumer**, where XXX is the primitive type, for example, ObjIntConsumer.
- A BiFunction represents a function that takes two arguments of different type and produces a result of another type. Its functional descriptor is:
 $(T, U) \rightarrow R$

- It has primitive versions that take generic types as arguments and return int, long and double primitive types, with the naming convention **ToXXXBiFunction<T, U>**, where XXX is the primitive type.
- A BinaryOperator is a specialization of the BiFunction interface (in fact, this interface extends from it) for when the arguments and the result are of the same type. Its functional descriptor is:
(T, T) -> T
- It defines two static methods:

```
static <T> BinaryOperator<T> minBy(  
        Comparator<? super T> comparator)  
static <T> BinaryOperator<T> maxBy(  
        Comparator<? super T> comparator)
```

- It has primitive versions for int, long and double, for example, IntBinaryOperator.

SELF TEST

1. Given:

```
public class Question_10_1 {  
    public static void main(String[] args) {  
        Predicate<String> p1 = t -> {  
            System.out.print("p1");  
            return t.startsWith(" ");  
        };  
        Predicate<String> p2 = t -> {  
            System.out.print("p2");  
            return t.length() > 5;  
        };  
        p1.and(p2).test("a question");  
    }  
}
```

What is the result?

- A. p1
- B. p2
- C. p1p2
- D. false
- E. Compilation fails

2. Which of the following interfaces is a valid primitive version of BiConsumer<T, U>?

- A. IntBiConsumer<T>
- B. ObjLongConsumer<T>
- C. ToLongBiConsumer<T>
- D. IntToDoubleConsumer<T>

3. Given:

```
public class Question_10_3 {  
    public static void main(String[] args) {  
        IntUnaryOperator u1 = i -> i / 6;  
        IntUnaryOperator u2 = i -> i + 12;  
  
        System.out.println(  
            u1.compose(u2).applyAsInt(12)  
        );  
    }  
}
```

What is the result?

- A. 24
- B. 14
- C. 4
- D. 2
- E. Compilation fails

Chapter TEN

4. Which of the following statements is true?

- A. A Consumer takes a parameter of type T and returns a result of the same type.
- B. UnaryOperator is a specialization of the Operator interface.
- C. The BiFunction interface doesn't have primitive versions.
- D. A Supplier represents an operation that takes no arguments, but it returns some value.

5. Given:

```
public class Question_10_3 {  
    public static void main(String[] args) {  
        Supplier<Boolean> s = () -> {  
            Random generator = new Random();  
            int n = generator.nextInt(1);  
            return n % 2 == 0;  
        };  
        System.out.println(s.getAsBoolean());  
    }  
}
```

What is the result?

- A. true
- B. false
- C. Sometimes true, sometimes false
- D. Compilation fails

6. Which of the following interfaces is a valid primitive version of BiPredicate<T, U>?

- A. IntBiPredicate
- B. ObjBooleanPredicate
- C. ToLongBiPredicate
- D. BiPredicate doesn't have primitive versions

7. Which of the following primitive version of Function returns a generic type while taking a long argument?

- A. ToLongFunction
- B. LongFunction
- C. LongToObjectFunction
- D. There's no primitive version with this characteristic

8. Which of the following statements is true?

- A. The BinaryOperator interface extends from the BiFunction interface.
- B. The BiSupplier interface only takes one generic argument.
- C. The Supplier interface doesn't define any default methods.
- D. minBy and maxBy are two default methods of the BinaryOperator interface.

ANSWERS

1. The correct answer is A.

The `and(Predicate)` method of the `Predicate` interface represents a short-circuiting logical **AND**. This means that the other predicate won't be evaluated if the value of the first predicate can predict the result of the operation (if the first predicate return `false` in the case of **AND**).

In the code, when `p1` is executed, it prints "p1" and then `false` is returned, which stops the execution of the second predicate.

2. The correct answer is B.

`BiConsumer` represents a consumer that takes two arguments and doesn't return a result. This interface takes one object and a primitive value as a second argument, so the naming convention, in this case, is `ObjXXXConsumer`, where `XXX` is the primitive type.

3. The correct answer is C.

The `compose` method (for `Functions` and `UnaryOperators`) first applies the operator passed as a parameter to its input and then applies the other operator to the result.

In this case, `u2` is executed first, and its result (24) is passed to `u1`, resulting in 4.

4. The correct answer is D.

Option A is false. A `Consumer` is an operation that accepts a single input argument and returns no result.

Option B is false. UnaryOperator is a specialization of the Function interface.

Option C is false. The BiFunction interface has primitive versions for int, long, and boolean.

Option D is true. A Supplier represents an operation that takes no arguments, but it returns some value.

5. The correct answer is D.

getAsBoolean() is a method of BooleanSupplier, not Supplier. To compile this program, you either change s to BooleanSupplier or leave it as a Supplier and call the method get().

6. The correct answer is D.

BiPredicate doesn't have primitive versions.

7. The correct answer is B.

Function has three types of primitive versions:

- To indicate that the function returns a generic type and takes a primitive argument, the interface is named **XXXFunction**, for example, LongFunction<R>.
- To indicate that the function returns a primitive type

and takes a generic argument, the interface is named **ToXXXFunction**, for example, `ToLongFunction<T>`.

- To indicate that the function takes a primitive argument and returns another primitive type, the interface is named **XXXToYYYFunction**, where XXX is the argument type and YYY is the return type, for example, `LongToDoubleFunction`.

8. The correct answers are A and C.

Option A is true. The `BinaryOperator` interface extends from the `BiFunction` interface.

Option B is false. There's no `BiSupplier` interface. Since `Supplier` just returns values without taking arguments, it doesn't have any sense to have a binary version.

Option C is true. The `Supplier` interface doesn't define any default methods.

Option D is false. `minBy` and `maxBy` are two static (not default) methods of the `BinaryOperator` interface.

Chapter ELEVEN

Method References

Exam Objectives

- Use method references with Streams.

USING METHODS LIKE OBJECTS

As we know, in Java we can use references to objects, either by creating new objects:

```
List list = new ArrayList();
store(new ArrayList());
```

Or by using existing objects:

```
List list2 = list;
isFull(list2);
```

But what about a reference to a *method*?

If we only use a method of an object in another method, we still have to pass the full object as an argument. Wouldn't be more practical just to pass the method as an argument? Like this for example:

```
isFull(list.size);
```

In Java 8, thanks to lambda expressions, we can do something like that. We can use methods like if they were objects or primitive values.

And that's because a *method reference* is the shorthand syntax for a lambda expression that executes just **ONE** method.

A METHOD REFERENCE

CLASS OR INSTANCE THAT
CONTAINS THE METHOD

Object ::

OPERATOR FOR
METHOD REFERENCES

NAME METHOD TO EXECUTE

(NOTICE THAT IS JUST THE NAME, WITHOUT ARGUMENTS)

methodName

Chapter ELEVEN

We know that we can use lambda expressions instead of using an anonymous class. But sometimes, the lambda expression is really just a call to some method, for example:

```
Consumer<String> c = s -> System.out.println(s);
```

To make the code clearer, you can turn that lambda expression into a method reference:

```
Consumer<String> c = System.out::println;
```

In a method reference, you place the object (or class) that contains the method before the `::` operator and the name of the method after it without arguments.

But you may be thinking:

- How is this clearer?
- What happens to the arguments?
- How can this be a valid expression?
- I don't understand how to construct a valid method reference

First of all, a methods reference can't be used for any method. **They can be used only to replace a single-method lambda expression.**

So to use a method reference you first need a lambda expression with one method. And to use a lambda expression you first need a functional interface, an interface with just one abstract method.

In other words:

Instead of using

AN ANONYMOUS CLASS

you can use

A LAMBDA EXPRESSION

And if this just calls one method, you can use

A METHOD REFERENCE

There are four types of method references:

- A method reference to a *static method*
- A method reference to an *instance method of an object of a particular type*
- A method reference to an *instance method of an existing object*
- A method reference to a *constructor*

Let's begin by explaining the most natural case, a *static method*.

STATIC METHOD

In this case, we have a lambda expression like the following:

```
(args) -> Class.staticMethod(args)
```

That can be turned into the following method reference:

```
Class::staticMethod
```

Notice that between a static method and a static method reference instead of the . operator, we use the :: operator, and that we don't pass arguments to the method reference.

In general, we don't have to pass arguments to method references. However, arguments are treated depending on the type of method reference.

In this case, any arguments (if any) taken by the method are passed automatically behind the curtains.

Wherever we can pass a lambda expression that just calls a static method, we can use a method reference. For example, assuming this class:

```
class Numbers {  
    public static boolean isMoreThanFifty(int n1, int n2) {  
        return (n1 + n2) > 50;  
    }  
}
```

```
public static List<Integer> findNumbers(
    List<Integer> l, BiPredicate<Integer, Integer> p) {
    List<Integer> newList = new ArrayList<>();
    for(Integer i : l) {
        if(p.test(i, i + 10)) {
            newList.add(i);
        }
    }
    return newList;
}
```

We can call the `findNumbers()` method:

```
List<Integer> list = Arrays.asList(12, 5, 45, 18, 33, 24, 40);

// Using an anonymous class
findNumbers(list, new BiPredicate<Integer, Integer>() {
    public boolean test(Integer i1, Integer i2) {
        return Numbers.isMoreThanFifty(i1, i2);
    }
});

// Using a lambda expression
findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));

// Using a method reference
findNumbers(list, Numbers::isMoreThanFifty);
```

INSTANCE METHOD OF AN OBJECT OF A PARTICULAR TYPE

In this case, we have a lambda expression like the following:

```
(obj, args) -> obj.instanceMethod(args)
```

Where an instance of an object is passed, and one of its methods is executed with some optional(s) parameter(s).

That can be turned into the following method reference:

```
ObjectType::instanceMethod
```

This time, the conversion is not that straightforward. First, in the method reference, we don't use the instance itself. We use its type.

Second, the other argument of the lambda expression, if any, it's not used in the method reference, but it's passed behind the curtains like in the static method case.

For example, assuming this class:

```
class Shipment {  
    public double calculateWeight() {  
        double weight = 0;  
        // Calculate weight  
        return weight;  
    }  
}
```

And this method:

```
public List<Double> calculateOnShipments(  
    List<Shipment> l, Function<Shipment, Double> f) {  
    List<Double> results = new ArrayList<>();  
    for(Shipment s : l) {  
        results.add(f.apply(s));  
    }  
    return results;  
}
```

We can call that method using:

```
List<Shipment> l = new ArrayList<Shipment>();  
  
// Using an anonymous class  
calculateOnShipments(l, new Function<Shipment, Double>() {  
    public Double apply(Shipment s) { // The object  
        return s.calculateWeight(); // The method  
    }  
});  
  
// Using a lambda expression  
calculateOnShipments(l, s -> s.calculateWeight());  
  
// Using a method reference  
calculateOnShipments(l, Shipment::calculateWeight);
```

In this example, we don't pass any arguments to the method. The key point here is that an instance of the object is the parameter of the lambda expression, and we form the reference to the instance method with the type of the instance.

Chapter ELEVEN

Here's another example where we pass two arguments to the method reference.

Java has a `Function` interface that takes one parameter, a `BiFunction` that takes two parameters, but there's no `TriFunction` that takes three parameters, so let's make one:

```
interface TriFunction<T, U, V, R> {  
    R apply(T t, U u, V v);  
}
```

Now assume a class with a method that takes two parameters and returns a result, like this:

```
class Sum {  
    Integer doSum(String s1, String s2) {  
        return Integer.parseInt(s1) + Integer.parseInt(s2);  
    }  
}
```

We can wrap the `doSum()` method within a `TriFunction` implementation using an anonymous class:

```
TriFunction<Sum, String, String, Integer> anon =  
    new TriFunction<Sum, String, String, Integer>() {  
        @Override  
        public Integer apply(Sum s, String arg1, String arg2) {  
            return s.doSum(arg1, arg2);  
        }  
    };  
System.out.println(anon.apply(new Sum(), "1", "4"));
```

Or using a lambda expression:

```
TriFunction<Sum, String, String, Integer> lambda =  
    (Sum s, String arg1, String arg2) -> s.doSum(arg1, arg2);  
System.out.println(lambda.apply(new Sum(), "1", "4"));
```

Or just using a method reference:

```
TriFunction<Sum, String, String, Integer> mRef = Sum::doSum;  
System.out.println(mRef.apply(new Sum(), "1", "4"));
```

Here:

- The first type parameter of TriFunction is the object type that contains the method to execute.
- The second type parameter of TriFunction is the type of the first parameter.
- The third type parameter of TriFunction is the type of the second parameter.
- The last type parameter of TriFunction is the return type of the method to execute. Notice how this is omitted (inferred) in the lambda expression and the method reference.

It may seem odd to just see the interface, the class and how they are used with a method reference, but this becomes more evident when you see the anonymous class or even the lambda version.

From:

```
(Sum s, String arg1, String arg2) -> s.doSum(arg1, arg2)
```

To

```
Sum::doSum
```

INSTANCE METHOD OF AN EXISTING OBJECT

In this case, we have a lambda expression like the following:

```
(args) -> obj.instanceMethod(args)
```

That can be turned into the following method reference:

```
obj::instanceMethod
```

This time, an instance defined somewhere else is used, and the arguments (if any) are passed behind the curtains like in the static method case.

For example, assuming these classes:

```
class Car {  
    private int id;  
    private String color;  
    // More properties  
    // And getter and setters  
}  
  
class Mechanic {  
    public void fix(Car c) {  
        System.out.println("Fixing car " + c.getId());  
    }  
}
```

And this method:

```
public static void execute(Car car, Consumer<Car> c) {  
    c.accept(car);  
}
```

We can call the above method using:

```
final Mechanic mechanic = new Mechanic();  
Car car = new Car();  
  
// Using an anonymous class  
execute(car, new Consumer<Car>() {  
    public void accept(Car c) {  
        mechanic.fix(c);  
    }  
});  
  
// Using a lambda expression  
execute(car, c -> mechanic.fix(c));  
  
// Using a method reference  
execute(car, mechanic::fix);
```

The key, in this case, is to use any object visible by an anonymous class/lambda expression and pass some arguments to an instance method of that object.

Here's another quick example using another Consumer:

```
Consumer<String> c = System.out::println;  
c.accept("Hello");
```

CONSTRUCTOR

In this case, we have a lambda expression like the following:

```
(args) -> new ClassName(args)
```

That can be turned into the following method reference:

```
ClassName::new
```

The only thing this lambda expression does is to create a new object, so we just reference a constructor of the class with the keyword new. Like in the other cases, arguments (if any) are not passed in the method reference.

Most of the time, we can use this syntax with two (or three) interfaces of the `java.util.function` package.

If the constructor takes no arguments, a `Supplier` will do the job:

```
// Using an anonymous class
Supplier<List<String>> s = new Supplier() {
    public List<String> get() {
        return new ArrayList<String>();
    }
};
List<String> l = s.get();
```

```
// Using a lambda expression
Supplier<List<String>> s = () -> new ArrayList<String>();
List<String> l = s.get();

// Using a method reference
Supplier<List<String>> s = ArrayList<String>::new;
List<String> l = s.get();
```

If the constructor takes an argument, we can use the Function interface. For example:

```
// Using a anonymous class
Function<String, Integer> f =
    new Function<String, Integer>() {
        public Integer apply(String s) {
            return new Integer(s);
        }
};
Integer i = f.apply("100");

// Using a lambda expression
Function<String, Integer> f = s -> new Integer(s);
Integer i = f.apply("100");

// Using a method reference
Function<String, Integer> f = Integer::new;
Integer i = f.apply("100");
```

Chapter ELEVEN

If the constructor takes two arguments, we use the BiFunction interface:

```
// Using a anonymous class
BiFunction<String, String, Locale> f =
    new BiFunction<String, String, Locale>() {
    public Locale apply(String lang, String country) {
        return new Locale(lang, country);
    }
};

Locale loc = f.apply("en", "UK");

// Using a lambda expression
BiFunction<String, String, Locale> f =
    (lang, country) -> new Locale(lang, country);
Locale loc = f.apply("en", "UK");

// Using a method reference
BiFunction<String, String, Locale> f = Locale::new;
Locale loc = f.apply("en", "UK");
```

If you have a constructor with three or more arguments, you would have to create your own functional interface.

You can see that referencing a constructor is very similar to referencing a static method. The difference is that the constructor "method name" is new.

Many of the examples of this chapter are very simple and probably they don't justify the use of lambda expressions or method references.

As mentioned at the beginning of the chapter, use method reference if they make your code **CLEARER**.

You can avoid the one method restriction by grouping all your code in a static method, for example, and create a reference to that method instead of using a class or a lambda expression with many lines.

But the real power of lambda expressions and method references comes when they are combined with another new feature of Java 8: Streams.

That will be the topic of the next section.

KEY POINTS

- A method reference is the shorthand syntax to a lambda expression that executes just one method.
- The syntax of a method reference is:
`ObjectOrClassName :: methodName`
- In a method reference, you place the object (or class) that contains the method before the `::` operator and the name of the method after it without arguments.
- There are four types of method references:
 - A method reference to a *static method*
 - A method reference to an *instance method of an object of a particular type*
 - A method reference to an *instance method of an existing object*
 - A method reference to a *constructor*
- For static methods, we have a lambda expression like the following:
`(args) -> Class.staticMethod(args)`
- That can be turned into the following method reference:
`Class::staticMethod`
- For instance methods of objects of a particular type, we have a lambda expression like the following:

(obj, args) -> obj.instanceMethod(args)

- Where an instance of an object is passed as an argument and one of its methods is executed with some optional(s) parameter(s).

- And that can be turned into the following method reference:

ObjectType::instanceMethod

- For instance methods of existing objects, we have a lambda expression like the following:

(args) -> obj.instanceMethod(args)

- That can be turned into the following method reference:

obj::instanceMethod

- For creating objects (calling a constructor), we have a lambda expression like the following:

(args) -> **new ClassName(args)**

- That can be turned into the following method reference:

ClassName::new

SELF TEST

1. Given:

```
public class Question_11_1 {  
    public static Runnable print() {  
        return () -> {  
            System.out.println("Hi");  
        };  
    }  
    public static void main(String[] args) {  
        Runnable r = Question_11_1::print;  
        r.run();  
    }  
}
```

What is the result?

- A. Hi
- B. Nothing is printed
- C. Compilation fails
- D. An exception occurs at runtime

2. Given:

```
public class Question_11_2 {  
    void print() {  
        System.out.println("Hi");  
    }  
  
    public static void main(String[] args) {  
        Consumer<Question_11_2> c =  
            Question_11_2::print;  
        c.accept(new Question_11_2());  
    }  
}
```

What is the result?

- A. Hi
- B. Nothing is printed
- C. Compilation fails
- D. An exception occurs at runtime

3. Which of the following statements is true?

- A. You can have a method reference of a constructor.
- B. Method references replace any lambda expression.
- C. When using method references, you don't have to specify the arguments the method receives.
- D. The :: operator can be also used in lambda expressions.

Chapter ELEVEN

4. Given:

```
class Car {  
    public void drive(int speed) {  
        //...  
    }  
}
```

And:

```
Car c = new Car();  
IntConsumer consumer = (int speed) -> c.drive(speed);
```

Which of the following method references can replace the above lambda expression?

- A. Car.drive
- B. c.drive(speed)
- C. Car::drive
- D. c::drive

5. Given:

```
public class Question_11_5 {  
    Question_11_5() {  
        System.out.println(0);  
    }  
  
    public static void main(String[] args) {  
        Runnable r = Question_11_5::new;  
    }  
}
```

What is the result?

- A. 0
- B. Nothing is printed
- C. Compilation fails
- D. An exception occurs at runtime

Chapter ELEVEN

6. Given:

```
class Test {
    Test() {
        System.out.println(0);
    }

    Test(String s) {
        System.out.println(1);
    }
}

public class Question_11_6 {

    public static void main(String[] args) {
        BiFunction<String, String, Test> f = Test::new;
        f.apply(null, null);
    }
}
```

What is the result?

- A. 0
- B. 1
- C. Nothing is printed
- D. Compilation fails
- E. An exception occurs at runtime

7. Given:

```
public class Question_11_7 {  
  
    private Question_11_7() {}  
  
    public static Question_11_7 create() {  
        return new Question_11_7();  
    }  
  
    public static void main(String args[]) {  
        // 1  
    }  
}
```

Which of the following method references can be used to get instances of class Question_11_7 at line // 1?

A.

```
Supplier<Question_11_7> s = Question_11_7::new;
```

B.

```
UnaryOperator<Question_11_7> u = Question_11_7::create;
```

C.

```
Consumer<Question_11_7> c = Question_11_7::create;
```

D.

```
Supplier<Question_11_7> s = Question_11_7::create;
```

ANSWERS

1. The correct answer is B.

```
Runnable r = Question_11_1::print;
```

It's equivalent to:

```
Runnable r = () -> Question_11_1.print();
```

And equivalent to:

```
Runnable r = new Runnable() {  
    public void run() {  
        Question_11_1.print();  
    }  
}
```

The print() method returns another Runnable instance, but as this is not saved or executed, nothing is done.

2. The correct answer is A.

```
Consumer<Question_11_2> c = Question_11_2::print;
```

It's equivalent to:

```
Consumer<Question_11_2> c = (Question_11_2 q) -> q.print();
```

And equivalent to:

```
Consumer<Question_11_2> c = new Consumer<Question_11_2>() {  
    public void accept(Question_11_2 q) {  
        q.print();  
    }  
}
```

That when executed, it just prints "Hi".

3. The correct answers are A and C.

Option A is true. You can have a method reference of a constructor.

Option B is false. Method references only replace one-method lambda expressions.

Option C is true. When using method references, you don't have to specify the arguments the method receives.

Option D is false. The :: operator is just used in method references.

4. The correct answer is D.

Option A and B are not valid method references.

Option C would be right if the method drive(int) were static, but because it's an instance method, option D is the right one.

5. The correct answer is B.

```
Runnable r = Question_11_5::new;
```

It's equivalent to:

```
Runnable r = () -> new Question_11_5();
```

And equivalent to:

Chapter ELEVEN

```
Runnable r = new Runnable() {
    public void run() {
        new Question_11_5();
    }
}
```

However, `r.run()` is never executed, so nothing is printed.

6. The correct answer is D.

Compilation fails because

```
BiFunction<String, String, Test> f = Test::new;
```

It's equivalent to:

```
BiFunction<String, String, Test> f =
    (String s1, String s2) -> new Test(s1, s2);
```

And class `Test` only has constructors that take zero and one arguments.

7. The correct answers are A and D.

Option A is correct because the private constructor can be used inside the `main` method.

Option B is wrong because with an `UnaryOperator`, a parameter of type `Question_11_7` is expected.

Option C is wrong because with a `Consumer` a parameter of type `Question_11_7` is expected, and nothing is returned.

Option D is correct because a `Supplier` returns an instance of `Question_11_7` by calling the static method `create()`.

Part FOUR

Streams and Collections

Chapter TWELVE

Streams

Exam Objectives

- Describe Stream interface and Stream pipeline
- Use method references with Streams

A SIMPLE EXAMPLE

Suppose you have a list of students and the requirements are to extract the students with a score of 90.0 or greater and sort them by score in ascending order.

One way to do it would be:

```
List<Student> studentsScore = new ArrayList<Student>();
for(Student s : students) {
    if(s.getScore() >= 90.0) {
        studentsScore.add(s);
    }
}
Collections.sort(studentsScore, new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return Double.compare(s1.getScore(), s2.getScore());
    }
});
```

Very verbose when we compare it with the Java 8 implementation using streams:

```
List<Student> studentsScore = students
    .stream()
    .filter(s -> s.getScore() >= 90.0)
    .sorted(Comparator.comparing(Student::getScore))
    .collect(Collectors.toList());
```

Don't worry if you don't fully understand the code; we'll see what it means later.

WHAT ARE STREAMS?

First, streams are **NOT** collections.

A simple definition is that streams are **WRAPPERS** for collections and arrays. They wrap an **EXISTING** collection (or another data source) to support operations expressed with **LAMBDAS**, so you specify what you want to do, not how to do it. You already saw it.

These are the characteristics of a stream:

Streams work perfectly with lambdas.

All streams operations take functional interfaces as arguments, so you can simplify the code with lambda expressions (and method references).

Streams don't store its elements.

The elements are stored in a collection or generated on the fly. They are only carried from the source through a pipeline of operations.

Streams are immutable.

Streams don't mutate their underlying source of elements. If, for example, an element is removed from the stream, a new stream with this element removed is created.

Streams are not reusable.

Streams can be traversed only once. After a terminal operation is executed (we'll see what this means in a moment), you have to create another stream from the source to further process it.

Streams don't support indexed access to their elements.

Again, streams are not collections or arrays. The most you can do is get their first element.

Streams are easily parallelizable.

With the call of a method (and following certain rules), you can make a stream execute its operations concurrently, without having to write any multithreading code.

Stream operations are lazy when possible.

Streams defer the execution of their operations either until the results are needed or until it's known how much data is needed.

One thing that allows this laziness is the way their operations are designed. Most of them return a new stream, allowing operations to be chained and form a pipeline that enables this kind of optimizations.

To set up this pipeline you:

1. Create the stream.
2. Apply **zero or more** intermediate operations to transform the initial stream into new streams.
3. Apply a terminal operation to generate a result or a "side-effect".

We're going to explain these steps and finally, we'll talk about more about laziness. In subsequent chapters, we'll go through all the operations supported by streams.

CREATING STREAMS

A stream is represented by the `java.util.stream.Stream<T>` interface. This works with objects only.

There are also specializations to work with primitive types, such as `IntStream`, `LongStream`, and `DoubleStream`.

There are many ways to create a stream. Let's start with the most popular three.

The first one is creating a stream from a `java.util.Collection` implementation using the `stream()` method:

```
List<String> words =  
    Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});  
Stream<String> stream = words.stream();
```

The second one is creating a stream from individual values:

```
Stream<String> stream = Stream.of("hello", "hola", "hallo", "ciao");
```

The third one is creating a stream from an array:

```
String[] words = {"hello", "hola", "hallo", "ciao"};  
Stream<String> stream = Stream.of(words);
```

However, you have to be careful with this last method when working with primitives.

Here's why. Assume an int array:

```
int[] nums = {1, 2, 3, 4, 5};
```

When we create a stream from this array like this:

```
Stream.of(num)
```

We are not creating a stream of Integers (`Stream<Integer>`), but a stream of int arrays (`Stream<int[]>`). This means that instead of having a stream with five elements we have a stream of one element:

```
System.out.println(Stream.of(nums).count()); // It prints 1!
```

The reason is the signatures of the of method:

```
static <T> Stream<T> of(T t) // returns a stream of one element
static <T> Stream<T> of(T... values) // returns a stream whose
                                            // elements are the specified values
```

Since an `int` is not an object, but `int[]` is, the method chosen to create the stream is the first (`Stream.of(T t)`), not the one with the vargs, so a stream of `int[]` is created, but since only one array is passed, the result is a stream of one element.

To solve this, we can force Java to choose the vargs version by creating an array of objects (with `Integer`):

```
Integer[] nums = {1, 2, 3, 4, 5};
System.out.println(Stream.of(nums).count()); // It prints 5!
```

Or use a fourth way to create a stream (that it's in fact used inside `Stream.of(T... values)`):

Chapter TWELVE

```
int[] nums = {1, 2, 3, 4, 5};  
// It also prints 5!  
System.out.println(Arrays.stream(nums).count());
```

Or use the primitive version IntStream:

```
int[] nums = {1, 2, 3, 4, 5};  
// It also prints 5!  
System.out.println(IntStream.of(nums).count());
```

Lesson learned: Don't use Stream<T>.of() when working with primitives.

Here are other ways to create streams.

```
static <T> Stream<T> generate(Supplier<T> s)
```

This method returns an "infinite" stream where each element is generated by the provided Supplier, and it's generally used with the method:

```
Stream<T> limit(long maxSize)
```

That truncates the stream to be no longer than maxSize in length.

For example:

```
Stream<Double> s = Stream.generate(new Supplier<Double>() {  
    public Double get() {  
        return Math.random();  
    }  
}).limit(5);
```

Or:

```
Stream<Double> s = Stream.generate(() -> Math.random()).limit(5);
```

Or just:

```
Stream<Double> s = Stream.generate(Math::random).limit(5);
```

That generates a stream of five random doubles.

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

This method returns an "infinite" stream produced by the iterative application of a function *f* to an initial element *seed*. The first element (*n* = 0) in the stream will be the provided seed. For *n* > 0, the element at position *n* will be the result of applying the function *f* to the element at position *n* - 1. For example:

```
Stream<Integer> s = Stream.iterate(1,
    new UnaryOperator<Integer>() {
        @Override
        public Integer apply(Integer t) {
            return t * 2;
        }
    }).limit(5);
```

Or just:

```
Stream<Integer> s = Stream.iterate(1, t -> t * 2).limit(5);
```

That generates the elements 1, 2, 4, 8, 16.

Chapter TWELVE

There's a Stream.Builder<T> class (that follows the builder design pattern) with methods that add an element to the stream being built:

```
void accept(T t)
Stream.Builder<T> add(T t)
```

For example:

```
Stream.Builder<String> builder =
    Stream.<String>builder()
        .add("h").add("e").add("l").add("l");
builder.accept("o");
Stream<String> s = builder.build();
```

IntStream and LongStream define the methods:

```
static IntStream range(int startInclusive,
                      int endExclusive)
static IntStream rangeClosed(int startInclusive,
                           int endInclusive)

static LongStream range(long startInclusive,
                       long endExclusive)
static LongStream rangeClosed(long startInclusive,
                           long endInclusive)
```

That returns a sequential stream for the range of `int` or `long` elements.

For example:

```
IntStream s = IntStream.range(1, 4); // stream of 1, 2, 3
IntStream s = IntStream.rangeClosed(1,
4); // stream of 1, 2, 3, 4
```

Also, there are new methods in the Java API that generate streams.

For example:

```
IntStream s1 = new Random().ints(5, 1, 10);
```

That returns an `IntStream` of five random `ints` from one (inclusive) to ten (exclusive).

INTERMEDIATE OPERATIONS

You can easily identify intermediate operations; they always return a new stream. This allows the operations to be connected.

For example:

```
Stream<String> s = Stream.of("m", "k", "c", "t")
    .sorted()
    .limit(3)
```

An important feature of intermediate operations is that they don't process the elements until a terminal operation is invoked, in other words, they're lazy.

Intermediate operations are further divided into *stateless* and *stateful* operations.

Stateless operations retain no state from previously elements when processing a new element so each can be processed independently of operations on other elements.

Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

The following table summarizes the methods of the Stream interface that represent intermediate operations.

Method	Type	Description
<code>Stream<T> distinct()</code>	Stateful	Returns a stream consisting of the distinct elements.
<code>Stream<T> filter(Predicate<? super T> predicate)</code>	Stateless	Returns a stream of elements that match the given predicate.
<code><R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)</code>	Stateless	Returns a stream with the content produced by applying the provided mapping function to each element. There are versions for int, long and double also.
<code>Stream<T> limit(long maxSize)</code>	Stateful	Returns a stream truncated to be no longer than maxSize in length.
<code><R> Stream<R> map(Function<? super T, ? extends R> mapper)</code>	Stateless	Returns a stream consisting of the results of applying the given function to the elements of this stream. There are versions for int, long and double also.
<code>Stream<T> peek(Consumer<? super T> action)</code>	Stateless	Returns a stream with the elements of this stream, performing the provided action on each element.
<code>Stream<T> skip(long n)</code>	Stateful	Returns a stream with the remaining elements of this stream after discarding the first n elements.
<code>Stream<T> sorted()</code>	Stateful	Returns a stream sorted according to the natural order of its elements.
<code>Stream<T> sorted(Comparator<? super T> comparator)</code>	Stateful	Returns a stream with the sorted according to the provided Comparator.
<code>Stream<T> parallel()</code>	N/A	Returns an equivalent stream that is parallel.
<code>Stream<T> sequential()</code>	N/A	Returns an equivalent stream that is sequential.
<code>Stream<T> unordered()</code>	N/A	Returns an equivalent stream that is unordered.

TERMINAL OPERATIONS

You can also easily identify terminal operations, they always return something other than a stream.

After the terminal operation is performed, the stream pipeline is *consumed*, and can't be used anymore. For example:

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
IntStream s = IntStream.of(digits);  
long n = s.count();  
System.out.println(s.findFirst()); // An exception is thrown
```

If you need to traverse the same stream again, you must return to the data source to get a new one. For example:

```
int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
long n = IntStream.of(digits).count();  
System.out.println(IntStream.of(digits).findFirst()); // OK
```

The following table summarizes the methods of the Stream interface that represent terminal operations.

Method	Description
<code>boolean allMatch(Predicate<? super T> predicate)</code>	Returns whether all elements of this stream match the provided predicate.
<code>boolean anyMatch(Predicate<? super T> predicate)</code>	Returns whether any elements of this stream match the provided predicate.
<code>boolean noneMatch(Predicate<? super T> predicate)</code>	Returns whether no elements of this stream match the provided predicate.
<code>Optional<T> findAny()</code>	Returns an Optional describing some element of the stream.
<code>Optional<T> findFirst()</code>	Returns an Optional describing the first element of this stream.
<code><R,A> R collect(Collector<? super T,A,R> collector)</code>	Performs a mutable reduction operation on the elements of this stream using a Collector.
<code>long count()</code>	Returns the count of elements in this stream.
<code>void forEach(Consumer<? super T> action)</code>	Performs an action for each element of this stream.
<code>void forEachOrdered(Consumer<? super T> action)</code>	Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
<code>Optional<T> max(Comparator<? super T> comparator)</code>	Returns the maximum element of this stream according to the provided Comparator.
<code>Optional<T> min(Comparator<? super T> comparator)</code>	Returns the minimum element of this stream according to the provided Comparator.
<code>T reduce(T identity, BinaryOperator<T> accumulator)</code>	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<code>Object[] toArray()</code>	Returns an array containing the elements of this stream.
<code><A> A[] toArray(IntFunction<A[]> generator)</code>	Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array.
<code>Iterator<T> iterator()</code>	Returns an iterator for the elements of the stream.
<code>Spliterator<T> spliterator()</code>	Returns a spliterator for the elements of the stream.

LAZY OPERATIONS

Intermediate operations are deferred until a terminal operation is invoked. The reason is that intermediate operations can usually be merged or optimized by a terminal operation.

Let's take for example this stream pipeline:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> str.length())
    .filter(i -> i > 3)
    .limit(2)
    .forEach(System.out::println);
```

Here's what it does:

- It generates a stream of strings,
- Then convert the stream to a stream of ints (representing the length of each string)
- Then it filters the lengths greater than three,
- Then it grabs the first two elements of the stream and
- Finally, prints those two elements.

And you may think the map operation is applied to all seven elements, then the filter operation again to all seven, then it picks the first two, and finally it prints the values.

But this is not how it works. If we modify the lambda expressions of map and filter to print a message:

```
Stream.of("sun", "pool", "beach", "kid", "island", "sea", "sand")
    .map(str -> {
        System.out.println("Mapping: " + str);
        return str.length();
    })
    .filter(i -> {
        System.out.println("Filtering: " + i);
        return i > 3;
    })
    .limit(2)
    .forEach(System.out::println);
```

The order of evaluation will be revealed:

```
Mapping: sun
Filtering: 3
Mapping: pool
Filtering: 4
4
Mapping: beach
Filtering: 5
5
```

From this example, we can see the stream didn't apply all the operations on the pipeline to all elements, only until it finds the elements needed to return a result (due to the `limit(2)` operation). This is called *short-circuiting*.

Short-circuit operations cause intermediate operations to be processed until a result can be produced.

Chapter TWELVE

In such a way, because of lazy and short-circuit operations, streams don't execute all operations on all their elements. Instead, the elements of the stream go through a pipeline of operations until the point a result can be deduced or generated.

You can see short-circuiting as a subclassification. There's only one short-circuit intermediate operation, while the rest are terminal:

INTERMEDIATE

```
Stream<T> limit(long maxSize)
```

(Because it doesn't need to process all the elements of the stream to create a stream of a given size)

TERMINAL

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
Optional<T> findFirst()
Optional<T> findAny()
```

(Because as soon as you find a matching element, there's no need to continuing processing the stream)

In the next chapters, we'll review the rest of the operations of the Stream interface.

KEY POINTS

- Streams can be defined as wrappers for collections and arrays. They wrap an existing collection (or another data source) to support operations expressed with lambdas, so you specify what you want to do, not how to do it.
- These are the characteristics of a stream:
 - Streams work perfectly with lambdas.
 - Streams don't store its elements.
 - Streams are immutable.
 - Streams are not reusable.
 - Streams don't support indexed access to their elements.
 - Streams are easily parallelizable.
 - Stream operations are lazy when possible.
- Stream operations can be chained to form a pipeline. To set up this pipeline you:
 1. Create the stream.
 2. Apply zero or more intermediate operations to transform the initial stream into new streams.
 3. Apply a terminal operation to generate a result or a "side-effect".
- There are many ways to create a stream. The most popular are:

Chapter TWELVE

```
// From an existing collection
List<String> words =
    Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
Stream<String> s1 = words.stream();

// From individual elements
Stream<String> s2 = Stream.of("hello", "hola", "hallo", "ciao");

// From an array
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> s3 = Stream.of(words);
```

- Don't use `Stream<T>.of()` when working with primitives. Instead use `Arrays.stream` or the primitive versions of `Stream`:

```
int[] nums = {1, 2, 3, 4, 5};
IntStream s1 = Arrays.stream(nums);
IntStream s2 = IntStream.of(nums);
```

- Intermediate operations such as `map` or `filter` always return a new stream and are divided into *stateless* and *stateful* operations, and they are lazy, meaning that they are deferred until a terminal operation is invoked.
- Terminal operations such as `count` or `foreach` always return something other than a stream.
- Short-circuit operations cause intermediate operations to be processed until a result can be produced.
- In such a way, because of lazy and short-circuit operations, streams don't execute all operations on all their elements, but until the point a result can be deduced or generated.

SELF TEST

1. Given:

```
public class Question_12_1 {  
    public static void main(String[] args) {  
        IntStream.range(1, 10)  
            .filter(i -> {  
                System.out.print("1");  
                return i % 2 == 0;  
            })  
            .filter(i -> {  
                System.out.print("0");  
                return i > 3;  
            })  
            .limit(1)  
            .forEach(i -> {  
                System.out.print(i);  
            });  
    }  
}
```

What is the result?

- A. 101010104
- B. 11111111100000000004
- C. 11041106
- D. 1101104
- E. An exception is thrown

2. Which of the following are intermediate operations?

- A. limit
- B. peek
- C. anyMatch
- D. skip

3. Which of the following are terminal operations?

- A. sorted
- B. flatMap
- C. max
- D. distinct

4. Which of the following are short-circuit operations?

- A. reduce
- B. parallel
- C. findNone
- D. findFirst

5. Given:

```
public class Question_12_2 {  
    public static void main(String[] args) {  
        IntStream.range(1, 5).count().limit(4);  
    }  
}
```

What is the result?

- A. 5
- B. 4
- C. 1
- D. Compilation error
- E. An exception is thrown

ANSWERS

1. The correct answer is D.

`IntStream.range(1, 10)` produces a stream of ints from 1 to 9. Since `limit` is a short-circuit operation, when there's an element that fulfills both `filter` conditions, the stream will terminate. When 1 is evaluated, 1 is printed in the first filter, but since `false` is returned, 2 is processed. This element passes the first filter, but not the second one, so 10 is printed. When 3 is processed, 1 from the first filter is printed and then 4 is processed. This element passes both conditions so 10 is printed and the stream finishes, printing 4 at the end because of the `forEach` operation.

2. The correct answers are A, B, and D.

`limit`, `peek`, and `skip` are intermediate operations. `anyMatch` is a terminal operation.

3. The correct answer is C.

`max` is the only terminal operation. `sorted`, `flatMap`, and `distinct` are intermediate operations.

4. The correct answer is D.

`findFirst` is the only short-circuit operation. `reduce` is a terminal operation. `parallel` is an intermediate operation. `findNone` doesn't exist.

5. The correct answer is D.

Since `count` is a terminal operation, it doesn't return a reference to a stream, and it cannot be chained, so a compile-time error is generated.

Chapter THIRTEEN

Iterating and Filtering Collections

Exam Objectives

- Collections Streams and Filters.
- Iterate using forEach methods of Streams and List.
- Filter a collection by using lambda expressions.

ITERATION

Usually, when you have a list, you'd want to iterate over its elements. A common way is to use a `for` block.

Either with an index:

```
List<String> words = ...  
for(int i = 0; i < words.size(); i++) {  
    System.out.println(words.get(i));  
}
```

Or with an iterator:

```
List<String> words = ...  
for(Iterator<String> it = words.iterator(); it.hasNext();) {  
    System.out.println(it.next());  
}
```

Or with the so-called `for-each` loop:

```
List<String> words = ...  
for(String w : words) {  
    System.out.println(w);  
}
```

Besides looking ugly, the first two add points where an error can happen (the index and iterator variables). The recommended way is to use the `for-each` loop whenever you can.

Chapter THIRTEEN

Therefore, taking advantage of functional interfaces, Java 8 adds another option to iterate lists based on the for-each loop, the `foreach` method:

```
default void forEach(Consumer<? super T> action)
```

Since this method is defined in the `Iterable` interface, is not only available to lists, but to all the implementations of this interface, such as Queues, Sets, Deques, and even some SQL-related exceptions, like `SQLException`.

Also, notice this is a default method, which means that there's a default implementation that implementing classes can override (and many do, mostly to deal with concurrent modifications).

This is the default implementation:

```
for (T t : this) {  
    action.accept(t);  
}
```

So basically, it's a for-each loop using the new functional features of Java 8.

To use it, we can start with an anonymous class:

```
List<String> words = ...  
words.forEach(new Consumer<String>() {  
    public void accept(String t) {  
        System.out.println(t);  
    }  
});
```

Remember that the Consumer interface represents an operation that takes one parameter but doesn't return any result.

That anonymous class can be transformed into a lambda expression:

```
words.forEach(t -> System.out.println(t));
```

Or in this particular example, a method reference:

```
words.forEach(System.out::println);
```

Just remember the rules about using final or effectively final variables inside anonymous classes or lambda expressions. Code like the following is not valid:

```
int max = 0;
words.forEach(t -> {
    // The following line won't compile, you can't modify max
    max = Math.max(max, t.length());
    System.out.println(t);
});
```

If you want to do things like this (get the max length of all the strings in a list), it's better to use streams to iterate the collection and apply other operations (for this particular example, we'll see how to calculate the max length with the reduce method in a later chapter).

Chapter THIRTEEN

The Stream interface provides a corresponding forEach method:

```
void forEach(Consumer<? super T> action)
```

Since this method doesn't return a stream, it represents a terminal operation.

Using it is not different than the Iterable version:

```
Stream<String> words = ...  
// As an anonymous class  
words.forEach((new Consumer<String>() {  
    public void accept(String t) {  
        System.out.println(t);  
    }  
});  
// As a lambda expression  
words.forEach(t -> System.out.println(t));  
// As a method reference  
words.forEach(System.out::println);
```

Of course, the advantage of using streams is that you can chain operations, for example:

```
words.sorted()  
.limit(2)  
.forEach(System.out::println);
```

As a terminal operation, you can't do things like this:

```
words.forEach(t -> System.out.println(t.length()));  
words.forEach(System.out::println);
```

If you want to do something like this, either create a new stream each time:

```
Stream.of(wordList).forEach(t -> System.out.println(t.length()));
Stream.of(wordList).forEach(System.out::println);
```

Or wrap the two calls to `System.out.println` into one lambda:

```
Consumer<String> print = t -> {
    System.out.println(t.length());
    System.out.println(t);
};

words.forEach(print);
```

You can't use `return`, `break` or `continue` to terminate an iteration either. `break` and `continue` will generate a compilation error since they cannot be used outside of a loop and `return` doesn't make sense when we see that the `foreach` method is implemented basically as:

```
for (T t : this) {
    action.accept(t); // Inside accept, return has no effect
}
```

As a side note (since it's not covered in the exam), Java 8 also added a `forEach` method to the `Map` interface. However, since a map has a key and value, this new method takes a `BiConsumer`:

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

With a default implementation is equivalent to:

```
for (Map.Entry<K, V> entry : map.entrySet()) {
    action.accept(entry.getKey(), entry.getValue());
}
```

FILTERING

Another common requirement is to filter (or remove) elements from a collection that don't match a particular condition.

You normally do this either by copying the matching elements to another collection:

```
List<String> words = ...  
List<String> nonEmptyWords = new ArrayList<String>();  
for(String w : words) {  
    if(w != null && !w.isEmpty()) {  
        nonEmptyWords.add(w);  
    }  
}
```

Or by removing the non-matching elements in the collection itself with an iterator (only if the collection supports removal):

```
List<String> words = new ArrayList<String>();  
// ... (add some strings)  
for (Iterator<String> it = words.iterator(); it.hasNext();) {  
    String w = it.next();  
    if (w == null || w.isEmpty()) {  
        it.remove();  
    }  
}
```

Now in Java 8, there's a new method on the Collection interface:

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

That removes all of the elements of the collection that satisfy the given predicate (the default implementation uses the iterator version).

This makes the code simpler by using either lambda expressions or method references:

```
// Using an anonymous class
words.removeIf(new Predicate<String>() {
    public boolean test(String t) {
        return t == null || t.isEmpty();
    }
});
// Using a lambda expression
words.removeIf(t -> t == null || t.isEmpty());
```

For the case where you copy the matching elements to another collection, you have the `filter` method of the `Stream` interface:

```
Stream<T> filter(Predicate<? super T> predicate)
```

That returns a new stream consisting of the elements that satisfy the given predicate.

Since this method returns a stream, it represents an intermediate operation, which basically means that you can chain any number of filters or other intermediate operations:

```
List<String> words = Arrays.asList("hello", null, "");
words.stream()
    .filter(t -> t != null) // ["hello", ""]
    .filter(t -> !t.isEmpty()) // ["hello"]
    .forEach(System.out::println);
```

Chapter THIRTEEN

Of course, the result of executing this code is:

```
hello
```

You can also create a method (or use an existing one) on some class to do it with a method reference, for the sole purpose of clarity:

```
class StringUtils {  
    public static boolean isNotNullOrEmpty(String s) {  
        return s != null && !s.isEmpty();  
    }  
}  
  
// ...  
  
List<String> words = Arrays.asList("hello", null, "");  
// Using an anonymous class  
words.stream()  
    .filter(new Predicate<String> () {  
        public boolean test(String t) {  
            return StringUtils.isNotNullOrEmpty(t);  
        }  
    })  
    .forEach(System.out::println);  
// Using a lambda expression  
words.stream()  
    .filter(t -> StringUtils.isNotNullOrEmpty(t))  
    .forEach(System.out::println);  
// Using a lambda expression  
words.stream()  
    .filter(StringUtils::isNotNullOrEmpty)  
    .forEach(System.out::println);
```

The `Stream` interface also has the `distinct` method to filter duplicate elements, according to the `Object.equals(Object)` method.

```
Stream<T> distinct()
```

Again, since it returns a new stream, this is an intermediate operation. Because it has to know the values of the elements to find out which ones are duplicates, this operation is also stateful.

Here's an example:

```
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
    .filter(t -> t != null) // ["hello", "hello"]
    .distinct() // ["hello"]
    .forEach(System.out::println);
```

Of course, the result is:

```
hello
```

KEY POINTS

- Java 8 adds the following method to the Iterable interface as another option to iterate the implementations of this interface (like lists):

```
default void forEach(Consumer<? super T> action)
```

- For example:

```
List<String> words = Arrays.asList("hello", "world");
words.forEach(t -> System.out.println(t));
```

- The Stream interface also has this method:

```
void forEach(Consumer<? super T> action)
```

- This is a terminal operation. Here's an example:

```
Stream<String> words = Stream.of("hello", "world");
words.forEach(t -> System.out.println(t));
```

- Of course, the advantage of using streams is that you can chain operations, for example:

```
words.sorted()
    .limit(2)
    .forEach(System.out::println);
```

- But as a terminal operation, you can't do things like this:

```
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

- For filtering, on the side of collections, we have a new method:

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

- That removes all of the elements of the collection that

satisfy the given predicate.

- On the Stream interface, we have:

```
Stream<T> filter(Predicate<? super T> predicate)
```

- That returns a new stream consisting of the elements that satisfy the given predicate.
- Since this method returns a stream, it represents an intermediate operation, which means that you can chain any number of filters or other intermediate operations:

```
List<String> words = Arrays.asList("hello", null, "");
words.stream()
    .filter(t -> t != null) // ["hello", ""]
    .filter(t -> !t.isEmpty()) // ["hello"]
    .forEach(System.out::println);
```

- The Stream interface also has the distinct method to filter duplicate elements, according to the Object.equals(Object) method.

```
Stream<T> distinct()
```

- This is an intermediate operation, and because it has to know the values of the elements to find out which ones are duplicates, this operation is also stateful. Here's an example:

```
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
    .filter(t -> t != null) // ["hello", "hello"]
    .distinct() // ["hello"]
    .forEach(System.out::println);
```

SELF TEST

1. Given:

```
public class Question_13_1 {  
    public static void main(String[] args) {  
        List<Integer> l = Arrays.asList(1,2,3,4,5,6);  
        Stream.of(l)  
            .forEach(i -> System.out.print(i-1));  
    }  
}
```

What is the result?

- A. 123456
- B. 012345
- C. 543210
- D. Compilation error
- E. An exception is thrown

2. Which of the following statements is true?

- A. filter is a terminal operation.
- B. filter is a stateful operation.
- C. Stream.forEach takes an implementation of the Consumer functional interface as an argument.
- D. You can chain more than one forEach operation in a stream pipeline.

3. Given:

```
public class Question_13_2 {  
    public static void main(String[] args) {  
        Arrays.asList(1,2,3,4,5,6).stream()  
            .filter(i -> i%2 == 0).filter(i -> i > 3)  
            .forEach(System.out::print);  
    }  
}
```

What is the result?

- A. 246
- B. 46
- C. 1
- D. 5
- E. Compilation error

4. Given:

```
public class Question_13_3 {  
    public static void main(String[] args) {  
        Arrays.asList(1,1,1,1,1,1).stream()  
            .filter(i -> i > 1).distinct()  
            .forEach(System.out::print);  
    }  
}
```

What is the result?

- A. 1
- B. 0
- C. Nothing is printed
- D. Compilation error
- E. An exception is thrown

ANSWERS

1. The correct answer is D.

`Stream.of(T t)` creates a stream of a single element, in this case of type `List<Integer>`. So `i` in the lambda expression has this type. Therefore, `i - 1` is an invalid expression that generates a compiler error.

To create a stream from this list, you have to use `l.stream()`.

2. The correct answer is C.

Option A is false. `filter` is an intermediate operation.

Option B is false. `filter` is a stateless operation since the predicate of the filter is applied to each element of the stream independently.

Option C is true. `Stream.forEach` takes an implementation of the `Consumer` functional interface as an argument.

Option D is false. `forEach` is a terminal operation, so you can't chain two `forEach` operations.

3. The correct answer is B.

The `filter` operation generates a stream with only even values (2, 4, 6). A second `filter` generates a stream with values greater than three (4, 6), that are finally printed.

4. The correct answer is C.

The `filter` returns an empty stream (since none of the elements match the condition), so nothing else is filtered or printed.

Chapter FOURTEEN

Optional Class

Exam Objectives

- Develop code that uses the Optional class.

THE PROBLEM WITH NULL

Most programming languages have a data type to represent the absence of a value, and it is known by many names:

NULL, nil, None, Nothing

The null type was introduced in ALGOL W by Tony Hoare in 1965, and it's considered one of the worst mistakes of computer science. In Tony Hoare's own words:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language ((ALGOL W)). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

– Tony Hoare

Still, some may be wondering, what is the problem with null?

Well, if you're a little worried by the problems this code might cause, you know the answer already:

```
String summary =  
    book.getChapter(10).getSummary().toUpperCase();
```

Chapter FOURTEEN

The problem with that code is that if any of those methods returns a null reference (for example, if the book doesn't have a tenth chapter), a `NullPointerException` (the most common exception in Java) will be thrown at runtime stopping the program.

What can we do to avoid this exception?

Perhaps, the easiest way is to check for `null`. Here's one way to it:

```
String summary = "";
if(book != null) {
    Chapter chapter = book.getChapter(10);
    if(chapter != null) {
        if(chapter.getSummary() != null) {
            summary = chapter.getSummary().toUpperCase();
        }
    }
}
```

You don't know if any object in this hierarchy can be `null`, so you check every object for this value. Obviously, this is not the best solution; it's not very practical and damage readability.

There may be another issue. Is checking for `null` really desirable? I mean, what if those objects should never be `null`? By checking for `null`, we will be hiding the error and not be dealing with it.

Of course, this is also a design issue. For example, if a chapter has no summary yet, what would be better to use as a default value? An empty string or `null`?

To address this problem, Java 8 introduced the class `java.util.Optional<T>`.

THE OPTIONAL CLASS

The job of this class is to **ENCAPSULATE** an optional value, an object that can be null.

Using the previous example, if we know that not all chapters have a summary, instead of modeling the class like this:

```
class Chapter {  
    private String summary;  
    // Other attributes and methods  
}
```

We can use the Optional class:

```
class Chapter {  
    private Optional<String> summary;  
    // Other attributes and methods  
}
```

So if there's a value, the Optional class just wraps it. Otherwise, an empty value is represented by the method `Optional.empty()`, that returns a singleton instance of `Optional`.

By using this class instead of `null`, first, we explicitly declare that the `summary` attribute is optional. Then, we can avoid `NullPointerExceptions` while having at our disposal the useful methods of `Optional` that we'll review up next.

First, let's see how to create an instance of this class.

Chapter FOURTEEN

To get an empty Optional object, use:

```
Optional<String> summary = Optional.empty();
```

If you are sure that an object is not null, you can wrap it in an Optional object this way:

```
Optional<String> summary = Optional.of("A summary");
```

An NullPointerException will be thrown if the object is null. However, you can use:

```
Optional<String> summary = Optional.ofNullable("A summary");
```

That returns an Optional instance with the specified value if it is non-null. Otherwise, it returns an empty Optional.

If you want to know if an Optional contains a value, you can do it like this:

```
if( summary.isPresent() ) {  
    // Do something  
}
```

Or in a more functional style:

```
summary.ifPresent(s -> System.out.println(s));  
// Or summary.ifPresent(System.out::println);
```

The ifPresent() method takes a Consumer<T> as an argument that is executed only if the Optional contains a value.

To get the value of an Optional use:

```
String s = summary.get();
```

However, this method will throw a `java.util.NoSuchElementException` if the Optional doesn't contain a value, so it's better to use the `ifPresent()` method.

Alternatively, if we want to return something when the Optional doesn't contain a value, there are three other methods we can use:

```
String summaryOrDefault = summary.orElse("Default summary");
```

The `orElse()` method returns the argument (that must be of type T, in this case a String) when the Optional is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrDefault =
    summary.orElseGet( () -> "Default summary" );
```

The `orElseGet()` method takes a `Supplier<? extends T>` as an argument that returns a value when the Optional is empty. Otherwise, it returns the encapsulated value.

```
String summaryOrElse =
    summary.orElseThrow( () -> new Exception() );
```

The `orElseThrow()` method takes a `Supplier<? extends X>`, where X is the type of the exception to throw when the Optional is empty. Otherwise, it returns the encapsulated value.

Chapter FOURTEEN

Like streams, there are versions of the `Optional` class to work with primitives, `OptionalInt`, `OptionalLong`, and `OptionalDouble`, so you can use `OptionalInt` instead of `Optional<Integer>`:

```
OptionalInt optionalInt = OptionalInt.of(1);
int i = optionalInt.getAsInt();
```

However, the use of these primitive versions are not encouraged, especially because they lack three useful methods of `Optional`: `filter()`, `map()`, and `flatMap()`. And since `Optional` just contains one value, the overhead of boxing/unboxing a primitive is not significant.

The `filter()` method returns the `Optional` if a value is present and matches the given predicated. Otherwise, an empty `Optional` is returned.

```
String summaryStr =
summary.filter(s -> s.length() > 10).orElse("Short summary");
```

The `map()` method is generally used to transform from one type to another. If the value is present, it applies the provided `Function<? super T, ? extends U>` to it. For example:

```
int summaryLength = summary.map(s -> s.length()).orElse(0);
```

The `flatMap()` method is similar to `map()`, but it takes an argument of type `Function<? super T, Optional<U>>` and if the value is present, it returns the `Optional` that results from applying the provided function. Otherwise, it returns an empty `Optional`.

In Chapter 17, we'll review in more detail the methods `map()` and `flatMap()` and how they are used with streams.

KEY POINTS

- The `java.util.Optional<T>` class **ENCAPSULATES** an optional value, i.e. an object that can be null.
- An empty value is represented by the method `Optional.empty()`.
- You can wrap an object in an `Optional` with the method `of()`, however, a `NullPointerException` will be thrown if the object is null.
- The method `ofNullable()` returns an `Optional` instance with the specified value if it is non-null. Otherwise, it returns an empty `Optional`.
- To get the value of an `Optional` use the method `get()`, but it will throw a `java.util.NoSuchElementException` if the `Optional` doesn't contain a value, so it's better to use the `ifPresent()` method that takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.
- The `orElse()` method returns the argument when the `Optional` is empty, otherwise, it returns the encapsulated value.
- The `orElseGet()` method takes a `Supplier` that returns a value when the `Optional` is empty. Otherwise, it returns the encapsulated value.
- The `orElseThrow()` method takes a `Supplier` that throws an exception when the `Optional` is empty. Otherwise, it returns the encapsulated value.

SELF TEST

1. Given:

```
public class Question_14_1 {  
    public static void main(String[] args) {  
        Optional<String> opt = Optional.of("1");  
        String s =  
            opt.orElseGet( () -> new RuntimeException() );  
        System.out.println(s);  
    }  
}
```

What is the result?

- A. 1
- B. Nothing is printed
- C. Compilation fails
- D. An exception occurs at runtime

2. Which of the following statements is true?

- A. The method `Optional.isPresent()` takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.
- B. The method `Optional.of()` can create an empty `Optional`.
- C. The method `Optional.of()` can throw a `NullPointerException`.
- D. The method `Optional.ifPresent()` takes a `Function<T,U>` as an argument.

3. Given:

```
public class Question_14_3 {  
    public static void main(String[] args) {  
        System.out.printlnToInt("a").get();  
    }  
    private static Optional<Integer> ToInt(String s) {  
        try {  
            return Optional.of(Integer.parseInt(s));  
        } catch(Exception e) {  
            return Optional.empty();  
        }  
    }  
}
```

What is the result?

- A. a
- B. Optional.empty
- C. Compilation fails
- D. An exception occurs at runtime

4. Given:

```
public class Question_14_4 {  
    public static void main(String[] args) {  
        System.out.println(Optional.of(0).orElse(1));  
    }  
}
```

What is the result?

- A. 0
- B. 1
- C. Compilation fails
- D. An exception occurs at runtime

ANSWERS

1. The correct answer is C.

The method `orElseGet()` returns a value of the type of the `Optional`, it doesn't return an exception, no matter if it's not executed. For exceptions, we have to use `orElseThrow()`.

2. The correct answer is C.

Option A is false. The method `Optional.ifPresent()` is the one that takes a `Consumer<T>` as an argument that is executed only if the `Optional` contains a value.

Option B is false. The method `Optional.of()` cannot create an empty `Optional`.

Option C is true. The method `Optional.of()` can throw a `NullPointerException` if its argument is `null`.

Option D is false. The method `Optional.ifPresent()` takes a `Consumer<T>` as an argument.

3. The correct answer is D.

The program is passing an invalid number to the method, so an exception is thrown by `Integer.parseInt()` and `Optional.empty` is returned. The method `get()` throws an exception if the `Optional` has an empty value, and this is exactly what happens.

4. The correct answer is A.

The `orElse()` method returns the argument when the `Optional` is empty, otherwise, it returns the encapsulated value of the `Optional`, in this case, 0.

Chapter FiFTEEN

Data Search

Exam Objectives

- Search for data by using search methods of the Stream classes including `findFirst`, `findAny`, `anyMatch`, `allMatch`, `noneMatch`.

FINDING AND MATCHING

Searching is a common operation when you have a set of data.

The Stream API has two types of operation for searching.

Methods starting with *Find*:

```
Optional<T> findAny()  
Optional<T> findFirst()
```

That search for an element in a stream. Since there's a possibility that an element couldn't be found (if the stream is empty, for example), the return type of this methods is an **Optional**.

And method ending with *Match*:

```
boolean allMatch(Predicate<? super T> predicate)  
boolean anyMatch(Predicate<? super T> predicate)  
boolean noneMatch(Predicate<? super T> predicate)
```

That indicate if a certain element matches the given predicate, that's why they return a **boolean**.

Since all these methods return a type different than a stream, they are considered **TERMINAL** operations.

FINDANY() AND FINDFIRST()

`findAny()` and `findFirst()` practically do the same, they return the first element they find in a stream:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream.findFirst().ifPresent(System.out::println); // 1
```

```
IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream2.findAny().ifPresent(System.out::println); // 1
```

If the stream is empty, they return an empty `Optional`:

```
Stream<String> stream = Stream.empty();
System.out.println(stream.findAny().isPresent()); // false
```

Of course, you can combine these methods with other stream operations:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream
    .filter(i -> i > 4)
    .findFirst().ifPresent(System.out::println); // 5
```

When to use `findAny()` and when to use `findFirst()`?

When working with parallel streams, it's harder to find the first element. In this case, it's better to use `findAny()` if you don't mind which element is returned.

ANYMATCH(), ALLMATCH(), AND NONEMATCH()

`anyMatch()` returns `true` if any of the elements in a stream matches the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream.anyMatch(i -> i%3 == 0)); // true
```

If the stream is empty or if there's no matching element, this method returns `false`:

```
IntStream stream = IntStream.empty();
System.out.println(stream.anyMatch(i -> i%3 == 0)); // false
```

```
IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream2.anyMatch(i -> i%10 == 0)); // false
```

`allMatch()` returns `true` only if **ALL** elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream.allMatch(i -> i > 0)); // true
```

```
IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream2.allMatch(i -> i%3 == 0)); // false
```

If the stream is empty, this method returns **TRUE** without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(stream.allMatch(i -> i%3 == 0)); // true
```

Chapter FIFTEEN

`noneMatch()` is the opposite of `allMatch()`, it returns true if **NONE** of the elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream.noneMatch(i -> i > 0)); // false
```

```
IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream2.noneMatch(i -> i%3 == 0)); // false
```

```
IntStream stream3 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(stream3.noneMatch(i -> i > 10)); // true
```

If the stream is empty, this method returns also **TRUE** without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(stream.noneMatch(i -> i%3 == 0)); // true
```

SHORT-CIRCUITING

All of these operations use something similar to the short-circuiting of `&&` and `||` operators.

Short-circuiting means that the evaluation stops once a result is found.

In the case of the `find*` operations, it's obvious that they stop at the first found element.

But in the case of the `*Match` operations, think about it, why would you evaluate all the elements of a stream when by evaluating the third element (for example) you can know if all or none (for example) of the elements will match?

Consider this code:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
boolean b = stream.filter(i -> {
    System.out.println("Filter:" + i);
    return i % 2 == 0;
})
.allMatch(i -> {
    System.out.println("AllMatch:" + i);
    return i < 3;
});
System.out.println(b);
```

What would you think the output will be?

Chapter FIFTEEN

The output:

```
Filter:1  
Filter:2  
AllMatch:2  
Filter:3  
Filter:4  
AllMatch:4  
false
```

As you can see, first of all, operations on a stream are not evaluated sequentially (in this case, first filter all the elements and then evaluate if all elements match the predicate of `allMatch()`).

Second, we can see that as soon as an element passes the filter predicate (like 2) the predicate of `allMatch()` is evaluated.

Finally, we can see short-circuiting in action. As soon as the predicate of `allMatch()` finds an element that doesn't evaluate to true (like 4), the two stream operations are canceled, no further elements are processed and the result is returned.

Just remember:

- With some operations, the whole stream doesn't need to be processed.
- Stream operations are not performed sequentially.

KEY POINTS

- The Stream API has two types of operation for searching. Methods starting with *Find*:

`Optional<T> findAny()`

`Optional<T> findFirst()`

- And method ending with *Match*:

`boolean allMatch(Predicate<? super T> predicate)`

`boolean anyMatch(Predicate<? super T> predicate)`

`boolean noneMatch(Predicate<? super T> predicate)`

- Both types are considered **TERMINAL** operations.
- `findAny()` and `findFirst()` practically do the same, they return the first element they find in a stream. If the stream is empty, they return an empty `Optional`.
- When working with parallel streams, it's harder to find the first element, so in this case, it's better to use `findAny()` if you don't really mind which element is returned.
- `anyMatch()` returns true if any of the elements in a stream matches the given predicate. If the stream is empty or if there's no matching element, it returns false.
- `allMatch()` returns true only if **ALL** elements in the stream match the given predicate.
- `noneMatch()` returns true if **NONE** of the elements in the stream match the given predicate.
- Both `allMatch()` and `noneMatch()` return true if the stream is empty.
- All of these operations are short-circuiting, meaning that the evaluation stops once a result is found.

SELF TEST

1. Given:

```
public class Question_15_1 {  
    public static void main(String[] args) {  
        Stream<Integer> s = Stream.of(100, 45, 98, 33);  
        s.anyMatch(i -> i > 50)  
            .findAny()  
            .ifPresent(System.out::println);  
    }  
}
```

What is the result?

- A. 100
- B. 98
- C. Nothing is printed
- D. Compilation fails

2. Which of the following methods of the Stream interface returns an Optional type?

- A. filter()
- B. findMatch()
- C. findAny()
- D. anyMatch()

3.

```
public class Question_15_3 {  
    public static void main(String[] args) {  
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);  
        stream.allMatch(i -> {  
            System.out.print(i);  
            return i % 3 == 0;  
        });  
    }  
}
```

What is the result?

- A. 1234567
- B. 36
- C. 1
- D. Compilation fails

4. Given:

```
public class Question_15_4 {  
    public static void main(String[] args) {  
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);  
        stream.filter(i -> {  
            return i > 3;  
        }).anyMatch(i -> {  
            System.out.print(i);  
            return i % 2 == 1;  
        });  
    }  
}
```

What is the result?

- A. 45
- B. 5
- C. 4567
- D. Compilation fails

ANSWERS

1. The correct answer is D.

`anyMatch()` is a terminal operation, it cannot be linked together with another operation.

2. The correct answer is C.

`findAny()` returns `Optional<T>`. `filter()` is an intermediate operation that returns a new stream. `findMatch()` doesn't exist and `anyMatch()` returns a boolean.

3. The correct answer is C.

`allMatch()` short-circuits at the first element that doesn't match the given predicate, 1.

4. The correct answer is A.

`filter()` returns a stream of elements starting with 4. So first, 4 is passed to `anyMatch()`, it's printed and since the condition fails, the next element generated by `filter()`, 5, it's passed, which is printed and, since the condition is true, the whole process is stopped.

Chapter SIXTEEN

Stream Operations on Collections

Exam Objectives

- Develop code that uses Stream data methods and calculation methods.
- Use `java.util.Comparator` and `java.lang.Comparable` interfaces.
- Sort a collection using Stream API.

COMPARATOR AND COMPARABLE

To sort arrays or collections, Java provides two very similar interfaces:

- **java.util.Comparator**

```
public interface Comparator<T> {  
    int compare(T obj1, T obj2);  
  
    // Other default and static methods ...  
}
```

- **java.lang.Comparable**

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```

The difference is that `java.util.Comparator` is implemented by a class you use to sort **ANOTHER** class' objects while `java.lang.Comparable` is implemented by the **SAME** object you want to sort.

With `Comparator`, you make an object to *compare* two objects of another type to sort them, that's why you take two parameters and the method is called *compare* (those two objects).

With `Comparable`, you make an object *comparable* to another object of the same type to sort them; that's why you only take **ONE** parameter, and the method is called *compareTo* (that other object). Since this interface is easier to grasp, let's start with it.

JAVA.LANG.COMPARABLE

The method to implement is:

```
int compareTo(T obj);
```

As you can see, it returns an `int`. Here are its rules:

- When **ZERO** is returned, it means that the object is **EQUAL** to the argument.
- When a number **GREATER** than zero is returned, it means that the object is **GREATER** than the argument.
- When a number **LESS** than zero is returned, it means that the object is **LESS** than the argument.

Many classes of Java (like `BigDecimal`, `BigInteger`, wrappers like `Integer`, `String`, etc.) implement this interface with a natural order (like 1, 2, 3, 4 or A, B, C, a, b, c).

Since this method can be used to test if an object is equal to another one, it's recommended that the implementation is consistent with the `equals(Object)` method (if the `compareTo` method returns 0, the `equals` method must return `true`).

Once a object implements this interface, it can be sorted by `Collections.sort()` or `Arrays.sort()`. Also, it can be used as key in a sorted map (like `TreeMap`) or in a sorted set (like `TreeSet`).

The following is an example of how an object can implement Comparable.

```
public class Computer implements Comparable<Computer> {
    private String brand;
    private int id;

    public Computer(String brand, int id) {
        this.brand = brand;
        this.id = id;
    }

    // Let's compare first by brand and then by id
    public int compareTo(Computer other) {
        // Reusing the implementation of String
        int result = this.brand.compareTo(other.brand);

        // If the objects are equal, compare by id
        if(result == 0) {
            // Let's do the comparison "manually"
            // instead of using Integer.compare()
            if(this.id > other.id) result = 1;
            else if( this.id < other.id) result = -1;
            // else result = 0;
        }
        return result;
    }

    // equals and compareTo must be consistent
    // to avoid errors in some cases
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Computer)) return false;
        return this.brand.equals(((Computer) other).brand)
               && this.id == ((Computer) other).id;
    }
}
```

Chapter SIXTEEN

```
public static void main(String[] args) {
    Computer c1 = new Computer("Lenovo", 1);
    Computer c2 = new Computer("Apple", 2);
    Computer c3 = new Computer("Dell", 3);
    Computer c4 = new Computer("Lenovo", 2);

    // Some comparisons
    System.out.println(c1.compareTo(c1)); // c1 == c1
    System.out.println(c1.compareTo(c2)); // c1 > c2
    System.out.println(c2.compareTo(c1)); // c2 < c1
    System.out.println(c1.compareTo(c4)); // c1 < c2
    System.out.println(c1.equals(c4)); // c1 != c2

    // Creating a list and sorting it
    List<Computer> list = Arrays.asList(c1, c2, c3, c4);
    Collections.sort(list);
    list.forEach(
        c -> System.out.format("%s-%d\n", c.brand, c.id)
    );
}
```

When you execute this program, this is the output:

```
0
11
-11
-1
false
Apple-2
Dell-3
Lenovo-1
Lenovo-2
```

JAVA.UTIL.COMPARATOR

The method to implement is:

```
int compare(T o1, T o2);
```

The rules of the returned value are similar than Comparable's:

- When **ZERO** is returned, it means that the **FIRST** argument is **EQUAL** to the **SECOND** argument.
- When a number **GREATER** than zero is returned, it means that the **FIRST** argument is **GREATER** than the **SECOND** argument.
- When a number **LESS** than zero is returned, it means that the **FIRST** argument is **LESS** than the **SECOND** argument.

One advantage of using a Comparator instead of Comparable is that you can have many Comparators to sort the same object in different ways.

For instance, we can take the Computer class of the previous example to create a Comparator that sorts first by id and then by brand, and since the rules of the returned value are practically the same as Comparable's, we can use the compareTo method:

```
Comparator<Computer> sortById = new Comparator<Computer>() {
    public int compare(Computer c1, Computer c2) {
        int result = Integer.compare(c1.id, c2.id);
        return result == 0
            ? c1.brand.compareTo(c2.brand) : result;
    }
};
```

Also, `Integer.compare(x, y)` is equivalent to:

```
Integer.valueOf(x).compareTo(Integer.valueOf(y))
```

Chapter SIXTEEN

Luckily, Comparator is a functional interface, so we can use a lambda expression instead of an inner class:

```
Comparator<Computer> sortById = (c1, c2) -> {
    int result = Integer.compare(c1.id, c2.id);
    return result == 0
        ? c1.brand.compareTo(c2.brand) : result;
}
```

So, when we use it in the list of the previous example:

```
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list, sortById);
list.forEach(
    c -> System.out.format("%d-%s\n", c.id, c.brand)
);
```

The output is:

```
1-Lenovo
2-Apple
2-Lenovo
3-Dell
```

In case you're wondering, Comparable is also considered a functional interface, but since Comparable is expected to be implemented by the object being compared, you'll almost never use it as a lambda expression.

In Java 8, with the introduction of default and static methods in interfaces, we have some useful methods on Comparator to simplify our code like:

```

Comparator<T>
    Comparator.comparing(Function<? super T, ? extends U>)
Comparator<T>
    Comparator.comparingInt(ToIntFunction<? super T>)
Comparator<T>
    Comparator.comparingLong(ToLongFunction<? super T>)
Comparator<T>
    Comparator.comparingDouble(ToDoubleFunction<? super T>)

```

That takes a Function (a lambda expression) that returns the value of a property of the object that will be used to create a Comparator using the value returned by comparedTo (also notice the versions when you're working with primitives).

For example:

```

Comparator<Computer> sortById =
    Comparator.comparing(c -> c.id);

```

Or:

```

Comparator<Computer> sortById =
    Comparator.comparingInt(c -> c.id);

```

They are equivalent to:

```

Comparator<Computer> sortById = new Comparator<Computer>() {
    public int compare(Computer c1, Computer c2) {
        return Integer.valueOf(c1.id)
            .compareTo(Integer.valueOf(c2.id));
    }
};

```

Chapter SIXTEEN

Another useful method is `thenComparing` that chains two Comparators (notice that this is not a static method):

```
Comparator<T>
    thenComparing(Comparator<? super T>)
Comparator<T>
    thenComparing(Function<? super T, ? extends U>)
Comparator<T>
    thenComparingIntToIntFunction<? super T>)
Comparator<T>
    thenComparingLongToLongFunction<? super T>)
Comparator<T>
    thenComparingDouble.ToDoubleFunction<? super T>)
```

This way, we can simplify the code to create a Comparator to sort by id and then by brand by using:

```
Comparator<Computer> sortByIdThenByBrand =
    Comparator.comparing((Computer c) -> c.id)
        .thenComparing(c -> c.brand);
```

Finally, the default method `reverse()` will create a Comparator that reverses the order of the original Comparator:

```
List<Computer> list = Arrays.asList(c1, c2, c3, c4);
Collections.sort(list,
    Comparator.comparing((Computer c) -> c.id).reversed());
list.forEach(c -> System.out.format("%d-%s\n", c.id, c.brand));
```

The output:

```
3-Dell
2-Apple
2-Lenovo
1-Lenovo
```

SORTING A STREAM

Sorting a stream is simple. The method

```
Stream<T> sorted()
```

Returns a stream with the elements sorted according to their natural order. For example:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);  
list.stream().sorted().forEach(System.out::println);
```

Will print:

```
2  
37  
38  
54  
57
```

The only requirement is that the elements of the stream implement `java.lang.Comparable` (that way, they are sorted in natural order). Otherwise, a `ClassCastException` may be thrown.

If we want to sort using a different order, there's a version of this method that takes a `java.util.Comparator` (this version is not available for primitive stream like `IntStream`):

```
Stream<T> sorted(Comparator<? super T> comparator)
```

Chapter SIXTEEN

For example:

```
List<String> strings =  
    Arrays.asList("Stream", "Operations", "on", "Collections");  
strings.stream()  
    .sorted( (s1, s2) -> s2.length() - s1.length() )  
    .forEach(System.out::println);
```

Or:

```
List<String> strings =  
    Arrays.asList("Stream", "Operations", "on", "Collections");  
strings.stream()  
    .sorted( Comparator.comparing(  
        String s) -> s.length()).reversed()  
    .forEach(System.out::println);
```

Both will print:

```
Collections  
Operations  
Stream  
on
```

The first snippet of code will return a positive value if the first string length is less than the second's, and a negative value otherwise, to sort the string in descending order.

The second snippet of code will create a Comparator with the length of the string in natural order (ascending order) and then reverse that order.

DATA AND CALCULATION METHODS

The Stream interface provides the following data and calculation methods:

```
long count()
Optional<T> max(Comparator<? super T> comparator)
Optional<T> min(Comparator<? super T> comparator)
```

And in the case of the primitive versions of the Stream interface:

IntStream

```
OptionalDouble average()
long count()
OptionalInt max()
OptionalInt min()
int sum()
```

LongStream

```
OptionalDouble average()
long count()
OptionalLong max()
OptionalLong min()
long sum()
```

DoubleStream

```
OptionalDouble average()
long count()
OptionalDouble max()
OptionalDouble min()
double sum()
```

Chapter SIXTEEN

`count()` returns the number of elements in the stream or zero if the stream is empty:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
System.out.println(list.stream().count()); // 5
```

`min()` returns the minimum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.

`max()` returns the maximum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.

When we talk about primitives, is easy to know which the minimum or maximum value is. But when we are talking about objects (of any kind), Java needs to know how to compare them to know which one is the maximum and the minimum. That's why the `Stream` interface needs a `Comparator` for `max()` and `min()`:

```
List<String> strings =
    Arrays.asList("Stream", "Operations", "on", "Collections");
strings.stream()
    .min( Comparator.comparing(
        String s) -> s.length())
    .ifPresent(System.out::println); // on
```

`sum()` returns the sum of the elements in the stream or zero if the stream is empty:

```
System.out.println(IntStream.of(28, 4, 91, 30).sum()); // 153
```

`average()` returns the average of the elements in the stream wrapped in an `OptionalDouble` or an empty one if the stream is empty:

```
System.out.println(IntStream.of(28, 4, 91, 30).average()); // 38.25
```

KEY POINTS

- `java.util.Comparator` is implemented by a class you use to sort **ANOTHER** class' objects. `java.lang.Comparable` is implemented by the **SAME** object you want to sort.
- The main methods of both interfaces return an `int`. Their rules are very similar:
 - When **ZERO** is returned, it means that the object (or first argument) is **EQUAL** to the (second) argument.
 - When a number **GREATERTHAN** than zero is returned, it means that the object (or first argument) is **GREATERTHAN** than the (second) argument.
 - When a number **LESSTHAN** than zero is returned, it means that the object (or first argument) is **LESSTHAN** than the (second) argument.
- `comparing()`, `thenComparing()`, and `reverse()` are helper methods of the `Comparator` interface added in Java 8.
- The `sorted()` method of the `Stream` interface returns a stream with the elements sorted according to its natural order. You can also pass a `Comparator` as an argument.
- `count()` returns the number of elements in the stream or zero if the stream is empty.
- `min()` returns the minimum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.
- `max()` returns the maximum value in the stream wrapped in an `Optional` or an empty one if the stream is empty.
- `sum()` returns the sum of the elements in the stream or zero if the stream is empty.
- `average()` returns the average of the elements in the stream wrapped in a `OptionalDouble` or an empty one if the stream is empty.

SELF TEST

1. Given:

```
public class Question_16_1 {  
    public static void main(String[] args) {  
        List<String> strings = Arrays.asList(  
            "Stream", "Operations", "on", "Collections");  
        Collections.sort(strings, String::compareTo);  
        System.out.println(strings.get(0));  
    }  
}
```

What is the result?

- A. Collections
- B. on
- C. Compilation fails
- D. An exception occurs at runtime

2. Which of the following statements returns a valid Comparator?

- A. (String s) -> s.length();
- B. Comparator.reversed();
- C. Comparator.thenComparing((String s) -> s.length());
- D. Comparator.comparing((String s) -> s.length() * -1);

3.

Given:

```
public class Question_16_3 {
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(30, 5, 8);
        list.stream().max().get();
    }
}
```

What is the result?

- A. 5
- B. 30
- C. Compilation fails
- D. An exception occurs at runtime

4. Given:

```
public class Question_16_4 {
    public static void main(String[] args) {
        List<String> strings = Arrays.asList(
            "Stream", "Operations", "on", "Collections");
        strings.stream().sorted(
            Comparator.comparing(
                (String s1, String s2) ->
                    s1.length() - s2.length()))
            .forEach(System.out::print);
    }
}
```

What is the result?

- A. CollectionsOperationsStreamOn
- B. onStreamOperationsCollections
- C. Compilation fails
- D. An exception occurs at runtime

ANSWERS

1. The correct answer is A.

`String::compareTo` is a valid method reference. Since `compareTo` sorts in natural order (A, B, C, etc.), "Collections" becomes the first element of the List.

2. The correct answer is D.

Option A is invalid. `Comparator` takes two arguments.

Option B is invalid. `reversed()` is not a static method.

Option C is invalid. `thenComparing` is not a static method.

Option D is valid. The static method `Comparator.comparing` takes a lambda expression with one argument that represents a Function that returns the property on which a Comparator is created.

3. The correct answer is C.

The `max` method of the `Stream` interface takes a `Comparator` as an argument. It's not like the `max` method of the primitive streams that know how to calculate the maximum element.

4. The correct answer is C.

The static method `Comparator.comparing` takes a lambda expression with one argument that represents a Function. Taking two arguments would require a `BiFunction`.

Chapter SEVENTEEN

Peeking, Mapping, Reducing and Collecting

Exam Objectives

- Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method.
- Save results to a collection using the collect method and group/partition data using the Collectors class.
- Use flatMap() methods of the Stream API.

PEEK()

peek() is a simple method:

```
Stream<T> peek(Consumer<? super T> action)
```

It just executes the provided Consumer and returns a new stream with the same elements of the original one.

Most of the time, this method is used with System.out.println() for debugging purposes (to see what's on the stream):

```
System.out.format("\n%d", IntStream.of(1, 2, 3, 4, 5, 6)
    .limit(3)
    .peek( i ->
        System.out.format("%d ", i) )
    .sum() );
```

The output:

```
1 2 3
6
```

Notice peek() is an intermediate operation. In the example, we can't use something like forEach() to print the values returned by limit() because forEach() is a terminal operation (and we couldn't call sum() anymore).

It's important to emphasize that peek() is intended to see the elements of a stream in a particular point of the pipeline, it's considered bad practice to change the stream in any way. If you want to do that, use the following method.

MAP()

map() is used to transform the value or the type of the elements of a stream:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

```
IntStream mapToInt(ToIntFunction<? super T> mapper)
```

```
LongStream mapToLong(ToLongFunction<? super T> mapper)
```

```
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

As you can see, map() takes a Function to convert the elements of a stream of type T to type R, returning a stream of that type R:

```
Stream.of('a', 'b', 'c', 'd', 'e')
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

The output:

```
97 98 99 100 101
```

There are versions for transforming to primitive types, for example:

```
IntStream.of(100, 110, 120, 130 ,140)
    .mapToDouble(i -> i/3.0)
    .forEach(i -> System.out.format("%.2f ", i));
```

Will output:

```
33.33 36.67 40.00 43.33 46.67
```

FLATMAP()

flatMap() is used to "flatten" (or combine) the elements of a stream into one (new) stream:

```
<R> Stream<R> flatMap(Function<? super T,
    ? extends Stream<? extends R>> mapper)

DoubleStream flatMapToDouble(Function<? super T,
    ? extends DoubleStream> mapper)

IntStream flatMapToInt(Function<? super T,
    ? extends IntStream> mapper)

LongStream flatMapToLong(Function<? super T,
    ? extends LongStream> mapper)
```

If the Function used by flatMap() returns null, an empty stream is returned instead.

Let's see how this work. If we have a stream of lists of characters:

```
List<Character> aToD = Arrays.asList('a', 'b', 'c', 'd');
List<Character> eToG = Arrays.asList('e', 'f', 'g');
Stream<List<Character>> stream = Stream.of(aToD, eToG);
```

And we want to convert all the characters to their int representation, what we need to do is to get the elements of the lists into one stream and then convert each character to an int. Fortunately, the “combining” part is exactly what flatMap() does:

Chapter SEVENTEEN

```
stream
    .flatMap(l -> l.stream())
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

So this code can output:

```
97 98 99 100 101 102 103
```

This way, with `flatMap()` you can convert a `Stream<List<T>>` to `Stream<T>`.

Using `peek()` after `flatMap()` may clarify how the elements are processed:

```
stream
    .flatMap(l -> l.stream())
    .peek(System.out::print)
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

As you can see from the output, the stream returned from `flatMap()` is passed through the pipeline, as if we were working with a stream of single elements and not with a stream of lists of elements:

```
a97 b98 c99 d100 e101 f102 g103
```

In both cases, `map()` and `flatMap()` return a stream. `map()` returns `Stream<Integer>` and `flatMap()` returns `Stream<Character>`.

In both cases, `map()` and `flatMap()` take a `Function` as its argument, but each `Function` has different parameters. `Function<Character, Integer>` and `Function<List<Character>, Stream<? extends Character>>`, respectively.

REDUCTION

A reduction is an operation that takes many elements and combines them (or *reduce* them) into a single value or object, and it is done by applying an operation multiple times.

Some examples of reductions are summing N elements, finding the maximum element of N numbers, or counting elements.

Like in the following example, where using a `for` loop, we reduce an array of numbers to their sum:

```
int[] numbers = {1, 2, 3, 4, 5, 6};  
int sum = 0;  
for(int n : numbers) {  
    sum += n;  
}
```

Of course, making reductions with streams instead of loops has more benefits, like easier parallelization and improved readability.

The `Stream` interface has two methods for reduction:

- `reduce()`
- `collect()`

We can implement reductions with both methods, but `collect()` helps us to implement a type of reduction called *mutable reduction*, where a container (like a `Collection`) is used to accumulate the result of the operation.

REDUCE()

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity,
         BinaryOperator<T> accumulator)

<U> U reduce(U identity,
               BiFunction<U,? super T,U> accumulator,
               BinaryOperator<U> combiner)

BinaryOperator<T>BiFunction<T,           T,           T>boolean
elementsFound=false;
T result = null;
for (T element : stream) {
    if (!elementsFound) {
        elementsFound = true;
        result = element;
    } else {
        result = accumulator.apply(result, element);
    }
return elementsFound ? Optional.of(result)
                     : Optional.empty();
```

This code just applies a function for each element, accumulating the result and returning an `Optional` wrapping that result, or an empty `Optional` if there were no elements.

Let's see a concrete example. We just see how a sum is a reduce operation:

```
int[] numbers = {1, 2, 3, 4, 5, 6};  
int sum = 0;  
for(int n : numbers) {  
    sum += n;  
}
```

Here, the accumulator operation is:

```
sum += n;
```

Or:

```
sum = sum + n;
```

Which translate to:

```
OptionalInt total = IntStream.of(1, 2, 3, 4, 5, 6)  
    .reduce( (sum, n) -> sum + n );
```

(Notice how the primitive version of stream uses the primitive version of `Optional`).

This is what happens step by step:

Chapter SEVENTEEN

1. An internal variable that accumulates the result is set to the first element of a stream (1).
2. This accumulator and the second element of the stream (2) are passed as arguments to the `BinaryOperator` represented by the lambda expression `(sum, n) -> sum + n`.
3. The result (3) is assigned to the accumulator.
4. The accumulator (3) and the third element of the stream (3) are passed as arguments to the `BinaryOperator`.
5. The result (6) is assigned to the accumulator.
6. Steps 4 and 5 are repeated for the next elements of the stream until there are no more elements.

However, what if you need to have an initial value? For cases like that, we have the version that takes two arguments:

```
T reduce(T identity,  
        BinaryOperator<T> accumulator)
```

The first argument is the initial value, and it is called *identity* because strictly speaking, this value must be an identity for the accumulator function, in other words, for each value *v*, `accumulator.apply(identity, v)` must be equal to *v*.

This version of `reduce()` is equivalent to:

```
T result = identity;  
for (T element : stream) {  
    result = accumulator.apply(result, element);  
}  
return result;
```

Notice that this version does not return an `Optional` object because if the stream empty, the identity value is returned.

For example, the sum example can be rewritten as:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
        .reduce( 0, (sum, n) -> sum + n ); // 21
```

Or using a different initial value:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
        .reduce( 4, (sum, n) -> sum + n ); // 25
```

However, notice that in the last example, the first value cannot be considered an identity (as in the first example) since, for instance, $4 + 1$ is not equal to 4.

This can bring some problems when working with parallel streams, which we'll review in the next chapter.

Now, notice that with these versions, you take elements of type T and return a reduced value of type T also.

However, if you want to return a reduced value of a different type, you have to use the three arguments version of reduce():

```
<U> U reduce(U identity,
            BiFunction<U, ? super T, U> accumulator,
            BinaryOperator<U> combiner)
```

(Notice the use of types T and U)

This version is equivalent to:

Chapter SEVENTEEN

```
U result = identity;
for (T element : stream) {
    result = accumulator.apply(result, element)
}
return result;
```

Consider for example that we want to get the sum of the length of all strings of a stream, so we are getting strings (type T), and we want an integer result (type U).

In that case, we use `reduce()` like this:

```
int length = Stream.of("Parallel", "streams", "are", "great")
    .reduce(0,
        (accumInt, str) -> accumInt + str.length(), //accumulator
        (accumInt1, accumInt2) -> accumInt1 + accumInt2); //combiner
```

We can make it clearer by adding the argument types:

```
int length = Stream.of("Parallel", "streams", "are", "great")
    .reduce(0,
        (Integer accumInt, String str) ->
            accumInt + str.length(), //accumulator
        (Integer accumInt1, Integer accumInt2) ->
            accumInt1 + accumInt2); //combiner
```

As the accumulator function adds a mapping (transformation) step to the accumulator function, this version of the `reduce()` can be written as a combination of `map()` and the other versions of the `reduce()` method (you may know this as the *map-reduce* pattern):

```
int length = Stream.of("Parallel", "streams", "are", "great")
    .mapToInt(s -> s.length())
    .reduce(0, (sum, strLength) -> sum + strLength);
```

Or simply:

```
int length = Stream.of("Parallel", "streams", "are", "great")
    .mapToInt(s -> s.length())
    .sum();
```

Because, in fact, the calculation operations that we learned about in the last chapter are implemented as reduce operations under the hood:

- average
- count
- max
- min
- sum

Also, notice that if return a value of the same type, the combiner function is no longer necessary (it turns out that this function is the same as the accumulator function) so, in this case, it's better to use the two argument version.

It's recommended to use the three version `reduce()` method when:

- Working with parallel streams (more of this in the next chapter)
- Having one function as a mapper and accumulator is more efficient than having separate mapping and reduction functions.

COLLECT()

This method has two versions:

```
<R,A> R collect(Collector<? super T,A,R> collector)
```

```
<R> R collect(Supplier<R> supplier,
                 BiConsumer<R,? super T> accumulator,
                 BiConsumer<R,R> combiner)
```

The first version uses predefined collectors from the `Collectors` class while the second one allows you to create your own collectors. Primitive streams (like `IntStream`), only have this last version of `collect()`.

Remember that `collect()` performs a mutable reduction on the elements of a stream, which means that it uses a mutable object for accumulating, like a `Collection` or a `StringBuilder`. In contrast, `reduce()` combines two elements to produce a new one and represents an immutable reduction.

However, let's start with the version that takes three arguments, as it's similar to the `reduce()` version that also takes three arguments.

As you can see from its signature, first, it takes a `Supplier` that returns the object that will be used as a container (accumulator) and returned at the end.

The second parameter is an accumulator function, which takes the container and the element to be added to it.

The third parameter is the combiner function, which merges the intermediate results into the final one (useful when working with parallel streams).

This version of collect() is equivalent to:

```
R result = supplier.get();
for (T element : stream) {
    accumulator.accept(result, element);
}
return result;
```

For example, if we want to "reduce" or "collect" all the elements of a stream into a List, we can do it this way:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
            () -> new ArrayList<>(), // Creating the container
            (l, i) -> l.add(i), // Adding an element
            (l1, l2) -> l1.addAll(l2) // Combining elements
        );
```

We can make it clearer by adding the argument types:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
            () -> new ArrayList<>(),
            (List<Integer> l, Integer i) -> l.add(i),
            (List<Integer> l1, List<Integer> l2) -> l1.addAll(l2)
        );
```

Or we can also use method references:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
            ArrayList::new,
            ArrayList::add,
            ArrayList::addAll
        );
```

COLLECTORS

The previous version of `collect()` is useful to learn how collectors work, but in practice, it's better to use the other version.

Some common collectors of the `Collectors` class are:

Method	Returned value from <code>collect()</code>	Description
<code>toList</code>	<code>List</code>	Accumulates elements into a <code>List</code> .
<code>toSet</code>	<code>Set</code>	Accumulates elements into a <code>Set</code> .
<code>toCollection</code>	<code>Collection</code>	Accumulates elements into a <code>Collection</code> implementation.
<code>toMap</code>	<code>Map</code>	Accumulates elements into a <code>Map</code> .
<code>joining</code>	<code>String</code>	Concatenates elements into a <code>String</code> .
<code>groupingBy</code>	<code>Map<K, List<T>></code>	Groups elements of type <code>T</code> in lists according to a classification function, into a map with keys of type <code>K</code> .
<code>partitioningBy</code>	<code>Map<Boolean, List<T>></code>	Partitions elements of type <code>T</code> in lists according to a predicate, into a map.

Since calculation methods can be implemented as reductions, the `Collectors` class also provides them as collectors:

Method	Returned value from <code>collect()</code>	Description
<code>averagingInt</code> <code>averagingLong</code> <code>averagingDouble</code>	<code>Double</code>	Returns the average of the input elements.
<code>counting</code>	<code>Long</code>	Counts the elements of input elements.
<code>maxBy</code>	<code>Optional<T></code>	Returns the maximum element according to a given <code>Comparator</code> .
<code>minBy</code>	<code>Optional<T></code>	Returns the minimum element according to a given <code>Comparator</code> .
<code>summingInt</code> <code>summingLong</code> <code>summingDouble</code>	<code>Integer</code> <code>Long</code> <code>Double</code>	Returns the sum of the input elements.

Chapter SEVENTEEN

This way, we can rewrite our previous example:

```
List<Integer> list =  
    Stream.of(1, 2, 3, 4, 5)  
        .collect(ArrayList::new,  
            ArrayList::add,  
            ArrayList::addAll  
    );
```

As:

```
List<Integer> list =  
    Stream.of(1, 2, 3, 4, 5)  
        .collect(Collectors.toList()); // [1, 2, 3, 4, 5]
```

Since all these methods are static, we can use static imports:

```
import static java.util.stream.Collectors.*;  
...  
List<Integer> list =  
    Stream.of(1, 2, 3, 4, 5)  
        .collect(toList()); // [1, 2, 3, 4, 5]
```

If we want to collect the elements into a Set:

```
Set<Integer> set =  
    Stream.of(1, 1, 2, 2, 2)  
        .collect(toSet()); // [1, 2]
```

If we want to create another Collection implementation:

```
Deque<Integer> deque =  
    Stream.of(1, 2, 3)  
        .collect(toCollection(ArrayDeque::new)); // [1, 2, 3]
```

Map, Reduce and Collect

If we are working with streams of Strings, we can join all the elements into one String with:

```
String s = Stream.of("a", "simple", "string")
    .collect(joining()); // "asimplestring"
```

We can also pass a separator:

```
String s = Stream.of("a", "simple", "string")
    .collect(joining(" ")); // " a simple string"
```

And a prefix and a suffix:

```
String s = Stream.of("a", "simple", "string")
    .collect(joining(" ", "This is ", "."));
    // "This is a simple string."
```

In the case of maps, things get a little more complicated, because depending on our needs, we have three options.

In the first option, `toMap()` takes two arguments (I'm not showing the return types because they are hard to read and don't provide much value anyway):

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper)
```

Both Functions take an element of the stream as an argument and return the key or the value of an entry of the Map, for example:

```
Map<Integer, Integer> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(i -> i,           // Key
                      i -> i * 2 // Value
                    )
    );
```

Chapter SEVENTEEN

Here, we're using the element (like 1) as the key, and the element multiplied by two as the value (like 2).

We can also write `i -> i` as `Function.identity()`:

```
Map<Integer, Integer> map =  
    Stream.of(1, 2, 3, 4, 5, 6)  
        .collect(toMap(Function.identity(), // Key  
                        i -> i * 2           // Value  
                    )  
    );
```

`java.util.function.Function.identity()` returns a function that always returns its input argument, in other words, it's equivalent to `t -> t`.

But what happens if more than one element is mapped to the same key, like in:

```
Map<Integer, Integer> map =  
    Stream.of(1, 2, 3, 4, 5, 6)  
        .collect(toMap(i -> i % 2,      // Key  
                        i -> i           // Value  
                    )  
    );
```

Java won't know what to do, so an exception will be thrown:

```
Exception in thread "main" java.lang.IllegalStateException: Duplicate key 1  
at java.util.stream.Collectors.lambda$throwingMerger$113(Collectors.java:133)  
at java.util.stream.Collectors$$Lambda$3/3035633356.apply(Unknown Source)  
at java.util.HashMap.merge(HashMap.java:1245)  
at java.util.stream.Collectors.lambda$toMap$171(Collectors.java:1320)
```

For those cases, we use the version that takes three arguments:

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper,
      BinaryOperator<U> mergeFunction)
```

The third argument is a function that defines what to do when there's a duplicate key. For example, we can create a List to append values:

```
Map<Integer, List<Integer>> map =
    Stream.of(1, 2, 3, 4, 5, 6)
        .collect(toMap(
            i -> i % 2,
            i -> new ArrayList<Integer>(Arrays.asList(i)),
            (list1, list2) -> {
                list1.addAll(list2);
                return list1;
            }
        ));
    }
```

This will return the following map:

```
{0=[2, 4, 6], 1=[1, 3, 5]}
```

The third version of `toMap()` takes all these arguments plus one that returns a new, empty Map into which the results will be inserted:

```
toMap(Function<? super T,? extends K> keyMapper,
      Function<? super T,? extends U> valueMapper,
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapSupplier)
```

Chapter SEVENTEEN

So we can change the default implementation (HashMap) to ConcurrentHashMap for example:

```
Map<Integer, List<Integer>> map =  
    Stream.of(1, 2, 3, 4, 5, 6)  
        .collect(toMap(  
            i -> i % 2,  
            i -> new ArrayList<Integer>(Arrays.asList(i)),  
            (list1, list2) -> {  
                list1.addAll(list2);  
                return list1;  
            },  
            ConcurrentHashMap::new  
        )  
    );
```

About the calculation methods, they are easy to use. Except for counting(), they either take a Function to produce a value to apply the operation or (in the case of maxBy and minBy) they take a Comparator to produce the result:

```
double avg = Stream.of(1, 2, 3)  
    .collect(averagingInt(i -> i * 2)); // 4.0  
  
long count = Stream.of(1, 2, 3)  
    .collect(counting()); // 3  
  
Stream.of(1, 2, 3)  
    .collect(maxBy(Comparator.naturalOrder()))  
    .ifPresent(System.out::println); // 3  
  
Integer sum = Stream.of(1, 2, 3)  
    .collect(summingInt(i -> i)); // 6
```

GROUPINGBY()

The `Collectors` class provides two functions to group the elements of a stream into a list, in a kind of an SQL `GROUP BY` style.

The first method is `groupingBy()` and it has three versions. This is the first one:

```
groupingBy(Function<? super T, ? extends K> classifier)
```

It takes a `Function` that classifies elements of type `T`, groups them into a list and returns the result in a `Map` where the keys (of type `K`) are the `Function` returned values.

For example, if we want to group a stream of numbers by the range they belong (tens, twenties, etc.), we can do it with something like this:

```
Map<Integer, List<Integer>> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
        .collect( groupingBy (i -> i/10 * 10) );
```

The moment you compare this code with the traditional way to it (with a `for` loop), it's when you realize the power of streams:

```
List<Integer> stream =
    Arrays.asList(2, 34, 54, 23, 33, 20, 59, 11, 19, 37);
Map<Integer, List<Integer>> map = new HashMap<>();

for(Integer i : stream) {
    int key = i/10 * 10;
    List<Integer> list = map.get(key);
    if(list == null) {
```

Chapter SEVENTEEN

```
        list = new ArrayList<>();
        map.put(key, list);
    }
    list.add(i);
}
```

Either way, those will return the following map:

```
{0=[2], 50=[54,59], 20=[23,20], 10=[11,19], 30=[34,33,37]}
```

The second version takes a *downstream collector* as an additional argument:

```
groupingBy(Function<? super T,? extends K> classifier,
           Collector<? super T,A,D> downstream)
```

A *downstream collector* is a collector that is applied to the results of another collector.

We can use any collector here, for instance, to count the elements in each group of the previous example:

```
Map<Integer, Long> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
        .collect(
            groupingBy( i -> i/10 * 10,
                       counting()
            )
        );

```

(Notice how the type of the values of the Map change to reflect the type returned by the downstream collector, counting())

This will return the following map:

```
{0=1, 50=2, 20=2, 10=2, 30=3}
```

We can even use another `groupingBy()` to classify the elements in a second level. For instance, instead of counting, we can further classify the elements in even or odd:

```
Map<Integer, Map<String, List<Integer>>> map =
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
        .collect(groupingBy(i -> i/10 * 10,
                           groupingBy(i -> i%2 == 0 ? "EVEN" : "ODD")
        )
    );
```

This will return the following map (with a little formatting):

```
{
    0  = {EVEN=[2]},
    50 = {EVEN=[54], ODD=[59]},
    20 = {EVEN=[20], ODD=[23]},
    10 = {ODD=[11, 19]},
    30 = {EVEN=[34], ODD=[33, 37]}
}
```

The key of the high-level map is an Integer because the first `groupingBy()` returns an Integer.

The type of the values of the high-level map changed (again) to reflect the type returned by the downstream collector, `groupingBy()`.

In this case, a String is returned so this will be the type of the keys of the second-level map, and since we are working with a stream of Integers, the values have a type of `List<Integer>`.

Chapter SEVENTEEN

Seeing the output of these examples, you may be wondering, is there a way to have the result ordered?

Well, `TreeMap` is the only implementation of `Map` that is ordered. Fortunately, the third version of `groupingBy()` add a `Supplier` argument that let us choose the type of the resulting `Map`:

```
groupingBy(Function<? super T, ? extends K> classifier,  
          Supplier<M> mapFactory,  
          Collector<? super T,A,D> downstream)
```

This way, if we pass an instance of `TreeMap`:

```
Map<Integer, Map<String, List<Integer>>> map =  
    Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)  
        .collect( groupingBy(i -> i/10 * 10,  
                           TreeMap::new,  
                           groupingBy(i -> i%2 == 0 ? "EVEN" : "ODD"))  
    );
```

This will return the following map:

```
{  
    0  = {EVEN=[2]},  
    10 = {ODD=[11, 19]},  
    20 = {EVEN=[20], ODD=[23]},  
    30 = {EVEN=[34], ODD=[33, 37]},  
    50 = {EVEN=[54], ODD=[59]}  
}
```

PARTITIONINGBY()

The second method for grouping is `partitioningBy()`.

The difference with `groupingBy()` is that `partitioningBy()` will return a Map with a Boolean as the key type, which means there are only two groups, one for `true` and one for `false`.

There are two versions of this method. The first one is:

```
partitioningBy(Predicate<? super T> predicate)
```

It partitions the elements according to a Predicate and organizes them into a `Map<Boolean, List<T>>`.

For example, if we want to partition a stream of numbers by the ones that are less than 50 and the ones that don't, we can do it this way:

```
Map<Boolean, List<Integer>> map =  
    Stream.of(45, 9, 65, 77, 12, 89, 31)  
        .collect(partitioningBy(i -> i < 50));
```

This will return the following map:

```
{false=[65, 77, 89], true=[45, 9, 12, 31, 12]}
```

As you can see, because of the Predicate, the map will always have two elements.

And like `groupingBy()`, this method has a second version that takes a

Chapter SEVENTEEN

downstream collector.

For example, if we want to remove duplicates, we just have to collect the elements into a Set like this:

```
Map<Boolean, Set<Integer>> map =  
    Stream.of(45, 9, 65, 77, 12, 89, 31, 12)  
        .collect(  
            partitioningBy(i -> i < 50,  
                           toSet()  
            )  
        );
```

This will produce the following Map:

```
{false=[65, 89, 77], true=[9, 12, 45, 31]}
```

However, unlike groupingBy(), there's no version that allows us to change the type of the Map returned. But it doesn't matter, you only have two keys that you can get with:

```
Set<Integer> lessThan50 = map.get(true);  
Set<Integer> moreThan50 = map.get(false);
```

KEY POINTS

- `peek()` executes the provided `Consumer` and returns a new stream with the same elements of the original one. Most of the time, this method is used for debugging purposes.
- `map()` is used to transform the value or the type of the elements of a stream through a provided `Function`.
- `flatMap()` is used to “flatten” or combine the contents of a stream into another (new) stream. In contrast to `map()`, `flatMap()` takes a `Function` on a `Stream` object, per se; whereas `map()` takes a `Function` on the objects within its stream. Both `map()` and `flatMap()` return some kind of `Stream`.
- A reduction is an operation that takes many elements and combines them (or *reduce* them) into a single value or object.
- `reduce()` performs a reduction on the elements of a stream using an accumulation function, an optional identity, and an also optional combiner function.
- `collect()` implements a type of reduction called *mutable reduction*, where a container (like a `Collection`) is used to accumulate the result of the operation.
- The `Collectors` class provides static methods such as `toList()` and `toMap()` to create a collection or a map from a stream and some calculation methods like `averagingInt()`.
- `Collectors.groupingBy()` groups the elements of a stream using a given `Function` as a classifier. It can also receive a *downstream collector* to create another level of classification.
- You can also group (or partition) the elements in a stream based on a condition (`Predicate`) using the `Collectors.partitioningBy()` method.

SELF TEST

1. Given:

```
public class Question_17_1 {  
    public static void main(String[] args) {  
        Map<Boolean, List<Integer>> map =  
            Stream.of(1, 2, 3, 4)  
                .collect(partitioningBy(i -> i < 5));  
        System.out.println(map);  
    }  
}
```

What is the result?

- A. {true=[1,2, 3, 4]}
- B. {false=[], true=[1, 2, 3, 4]}
- C. {false=[1,2, 3, 4]}
- D. {false=[1, 2, 3, 4], true=[]}

2. Given:

```
groupingBy(i -> i%3, toList())
```

Which of the following is equivalent?

- A. partitioningBy(i -> i%3 == 0, toList())
- B. partitioningBy(i -> i%3, toList())
- C. groupingBy(i -> i%3 == 0)
- D. groupingBy(i -> i%3)

3.

Given:

```
public class Question_17_3 {
    public static void main(String[] args) {
        Stream.of("aaaaa", "bbb", "ccc")
            .map(s -> s.split(""))
            .limit(1)
            .forEach(System.out::print);
    }
}
```

What is the result?

- A. aaaaa
- B. abc
- C. a
- D. None of the above

4. Given:

```
public class Question_17_4 {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("a", "b", "c")
                .flatMap(s -> Stream.of(s, s, s))
                .collect(Collectors.toList())
        );
    }
}
```

What is the result?

- A. [a, a, a, b, b, b, c, c, c]
- B. [a, a, a]
- C. [a, b, c]
- D. Compilation fails

Chapter SEVENTEEN

5. Which of the following is the right way to implement `OptionalInt min()` with a reduce operation?

- A. `reduce((a,b) -> a > b)`
- B. `reduce(Math::min)`
- C. `reduce(Integer.MIN_VALUE, Math::min)`
- D. `collect(Collectors.minBy())`

6. Which of the following is a correct overload of the `reduce()` method?

- A. `T reduce(BinaryOperator<T> accumulator)`
- B. `Optional<T> reduce(T identity,
BinaryOperator<T> accumulator)`
- C. `<U> U reduce(BinaryOperator<T> accumulator,
BinaryOperator<U> combiner)`
- D. `<U> U reduce(U identity,
BiFunction<U,? super T,U> accumulator,
BinaryOperator<U> combiner)`

7. Given:

```
public class Question_17_7 {
    public static void main(String[] args) {
        Map<Integer, Map<Boolean, List<Integer>>> map =
            Stream.of(56, 54, 1, 31, 98, 98, 16)
                .collect(groupingBy((Integer i) -> i%10,
                    TreeMap::new,
                    partitioningBy((Integer i) -> i > 5)
                )
            );
        System.out.println(map);
    }
}
```

What is the result?

- A. {


```
6={false=[], true=[56, 16]},
4={false=[], true=[54]},
1={false=[1], true=[31]},
8={false=[], true=[98]}
```

}
- B. {


```
1={false=[1], true=[31]},
4={false=[], true=[54]},
6={false=[], true=[56, 16]},
8={false=[], true=[98]}
```

}
- C. {


```
1={false=[1], true=[31]},
4={false=[], true=[54]},
6={false=[], true=[56, 16]},
8={false=[], true=[98, 98]}
```

}
- D. {


```
1={false=[1], true=[31]},
4={false=[], true=[54]}
```

}

ANSWERS

1. The correct answer is B.

`partitioningBy()` always returns a map with two keys, even if one key contains an empty list. Since the predicate returns true for all the elements of the stream, they are placed on the list of the true key.

2. The correct answer is D.

Option A is not equivalent because `partitioningBy()` returns a map with a Boolean as a key.

Option B doesn't compile, `partitioningBy()` takes a predicate as an argument (a lambda expression that must return a boolean).

Option C is not equivalent because it will create a map with keys of type Boolean.

Option D is equivalent because by default, `groupingBy()` creates a map whose values are of type `List<T>`.

3. The correct answer is D.

The program prints something like this:

```
[Ljava.lang.String;@87aac27
```

The mapping function converts each element of the stream to an array of strings.

Then, `limit(1)` returns the first element of the stream, an array containing `["a", "a", "a", "a", "a"]`, which is passed to `System.out.print()`.

4. The correct answer is A.

For each element of the stream, flatMap() returns a stream with the element three times repeated. The streams are merged into one and this is converted to a List, which is printed.

5. The correct answer is B.

Option A doesn't compile. reduce() takes a BinaryOperator, not a Predicate.

Option B is the right way to implement OptionalInt min() with a reduce operation.

Option C is incorrect because it doesn't return an Optional.

Option D doesn't compile. minBy() takes a Comparator.

6. The correct answer is D.

Option A is incorrect, it must return an Optional.

Option B is incorrect, it must return just T.

Option C is incorrect, it's missing the first argument, U identity.

7. The correct answer is C.

The type of the resulting map is:

`TreeMap<Integer, Map<Boolean, List<Integer>>>`

Elements are collected in a TreeMap, which has a natural order by default so we can discard Option A.

Keys of the map will be the remainder after dividing by 10 each element: 6, 4, 1, 1, 8, 8, 6. Values with the same key are appended to a List so we can discard Option B because it is missing a 98.

A downstream collector, partitioningBy(), partitions each value of the map into two groups, depending on whether the individual elements are greater than 5 or not, so Option B is also discarded.

Chapter SEVENTEEN

Chapter EIGHTEEN

Parallel Streams

Exam Objectives

- Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

WHAT IS A PARALLEL STREAM?

Until now, all the examples of this section have used sequential streams, where each element are processed one by one.

In contrast, parallel streams split the stream into multiple parts. Each part is processed by a different thread at the same time (in parallel).

Under the hood, parallel streams use the Fork/Join Framework (which we'll review in a later chapter).

This means that by default, the number of threads available to process parallel streams equals the number of available cores of the processor of your machine.

The advantage of use parallel streams over the Fork/Join Framework is that they are easier to use.

To create a parallel stream just use the `parallel()` method:

```
Stream<String> parallelStream =  
    Stream.of("a", "b", "c").parallel();
```

To create a parallel stream from a Collection use the `parallelStream()` method:

```
List<String> list = Arrays.asList("a", "b", "c");  
Stream<String> parStream = list.parallelStream();
```

How PARALLEL STREAMS WORK?

Let's start with the simplest example:

```
Stream.of("a", "b", "c", "d", "e")
    .forEach(System.out::print);
```

Printing a list of elements with a sequential stream will output the expected result:

abcde

However, when using a parallel stream:

```
Stream.of("a", "b", "c", "d", "e")
    .parallel()
    .forEach(System.out::print);
```

The output can be different for each execution:

```
cbade // One execution
cebad // Another execution
cbdea // Yet another execution
```

The reason is the decomposing of the stream on parts and their processing by different threads we talked about before.

So parallel streams are more appropriate for operations where the order of processing doesn't matter and that don't need to keep a state (they are stateless and independent).

An example to see this difference is the use of `findFirst()` vs. `findAny()`.

In a previous chapter, we mentioned that `findFirst()` method returns the first element of a stream. But when using parallel streams and this is decomposed in multiple parts, this method has to "know" which element is the first one:

```
long start = System.nanoTime();
String first = Stream.of("a", "b", "c", "d", "e")
    .parallel().findFirst().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    first + " found in " + duration + " milliseconds");
```

The output:

```
a found in 2.436155 milliseconds
```

Because of that, if the order doesn't matter, it's better to use `findAny()` with parallel streams:

```
long start = System.nanoTime();
String any = Stream.of("a", "b", "c", "d", "e")
    .parallel().findAny().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    any + " found in " + duration + " milliseconds");
```

The output:

```
c found in 0.063169 milliseconds
```

Chapter EIGHTEEN

As a parallel stream is processed, well, in parallel, it is reasonable to believe that it will be processed faster than a sequential stream. But as you can see with `findFirst()`, this is not always the case.

Stateful operations, like :

```
Stream<T> distinct()  
Stream<T> sorted()  
Stream<T> sorted(Comparator<? super T> comparator)  
Stream<T> limit(long maxSize)  
Stream<T> skip(long n)
```

Incorporate state from previously processed elements and may need to go through the entire stream to produce a result, so they are not a good fit for parallel streams.

By the way, you can turn a parallel stream into a sequential one with the `sequential()` method:

```
stream.parallel().filter(...).sequential().forEach(...);
```

Check if a stream is parallel with `isParallel()`:

```
stream.parallel().isParallel(); // true
```

And turn an ordered stream into a unordered one (or ensure that the stream is unordered) with `unordered()`;

```
stream.parallel().unordered().collect(...);
```

But don't believe that by first executing the stateful operations and then turning the stream into a parallel one, the performance will be better in all cases, or worse, the entire operation may run in parallel, like the following example:

```
double start = System.nanoTime();
Stream.of("b", "d", "a", "c", "e")
    .sorted()
    .filter(s -> {
        System.out.println("Filter:" + s);
        return !"d".equals(s);
    })
    .parallel()
    .map(s -> {
        System.out.println("Map:" + s);
        return s += s;
    })
    .forEach(System.out::println);
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println(duration + " milliseconds");
```

One might think that the stream is sorted and filtered sequentially, but the output shows something else:

```
Filter:c
Map:c
cc
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:d
Filter:e
Map:e
ee
79.470779 milliseconds
```

Compare this with the output of the sequential version (just comment `parallel()`):

Chapter EIGHTEEN

```
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:c
Map:c
cc
Filter:d
Filter:e
Map:e
ee
1.554562 milliseconds
```

In this case, the sequential version performed better.

But if we have an independent or stateless operation, where the order doesn't matter, let's say, counting the number of odd numbers in a large range, the parallel version will perform better:

```
double start = System.nanoTime();
long c = IntStream.rangeClosed(0, 1_000_000_000)
    .parallel()
    .filter(i -> i % 2 == 0)
    .count();
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Got " + c + " in " + duration + " milliseconds");
```

The parallel version output:

```
Got 500000001 in 738.678448 milliseconds
```

The sequential version output:

```
Got 500000001 in 1275.271882 milliseconds
```

In summary, parallel streams don't always perform better than sequential streams.

This, the fact that parallel streams process results independently and that the order cannot be guaranteed are the most important things you need to know.

But in practice, how do you know when to use sequential or parallel streams for better performance?

Here are some rules:

- For a small set of data, sequential streams are almost always the best choice due to the overhead of the parallelism.
- When using parallel streams, avoid stateful (like `sorted()`) and order-based (like `findFirst()`) operations.
- Operations that are computationally expensive (considering all the operation in the pipeline), generally have a better performance using a parallel stream.
- When in doubt, check the performance with an appropriate benchmark.

REDUCING PARALLEL STREAMS

In concurrent environments, assignments are bad.

This is because if you mutate the state of variables (especially if they are shared by more than one thread), you may run into many troubles to avoid invalid states.

Consider this example, which implements the factorial of 10 in a very particular way:

```
class Total {  
    public long total = 1;  
    public void multiply(long n) { total *= n; }  
}  
...  
Total t = new Total();  
LongStream.rangeClosed(1, 10)  
    .forEach(t::multiply);  
System.out.println(t.total);
```

Here, we are using a variable to gather the result of the factorial. The output of executing this snippet of code is:

3628800

However, when we turn the stream into a parallel one:

```
LongStream.rangeClosed(1, 10)  
    .parallel()  
    .forEach(t::multiply);
```

Sometimes we get the correct result and other times we don't.

The problem is caused by the multiple threads accessing the variable total concurrently. Yes, we can synchronize the access to this variable (as we'll see in a later chapter), but that kind of defeats the purpose of parallelism (I told you assignments are bad in concurrent environments).

Here's where `reduce()` comes in handy.

If you remember from the previous chapter, `reduce()` combines the elements of a stream into a single one.

With parallel streams, this method creates intermediate values and then combines them, avoiding the "ordering" problem while still allowing streams to be processed in parallel by eliminating the shared state and keep it inside the reduction process.

The only requirement is that the applied reducing operation must be *associative*.

This means that the operation `op` must follow this equality:

$$(a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c)$$

Or:

$$a \text{ op } b \text{ op } c \text{ op } d == (a \text{ op } b) \text{ op } (c \text{ op } d)$$

So we can evaluate `(a op b)` and `(c op d)` in parallel.

Returning to our example, we can implement it using `parallel()` and `reduce()` in this way:

Chapter EIGHTEEN

```
long tot = LongStream.rangeClosed(1, 10)
    .parallel()
    .reduce(1, (a,b) -> a*b);
System.out.println(tot);
```

When we execute this snippet of code, it produces the correct result every time (3628800).

And if we time the execution of the first snippet (47.216181 milliseconds) and this last one (3.094717 milliseconds), we can see a great improvement in performance. Of course, these values (and the other ones presented in this chapter) depend on the power of the machine, but you should get similar results.

We can also apply reduction to the example presented at the beginning of this chapter to process the string in parallel while maintaining the order:

```
String s = Stream.of("a","b","c","d","e")
    .parallel()
    .reduce("", (s1, s2) -> s1 + s2);
System.out.println(s);
```

The output:

abcde

These are simple examples, but reduction is hard to get it right sometimes, so it's best to avoid shared mutable state, use stateless and associative operations, and follow the rules to ensure parallel streams produce the best results.

For instance, remember this example from the last chapter?

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
    .reduce( 4, (sum, n) -> sum + n );
```

In this example, the first parameter is not really an identity since $4+n$ is not equal to n .

When I made the stream parallel, instead of getting the expect result (25), I got 45:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6).parallel()
    .reduce( 4, (sum, n) -> sum + n );
```

Why?

Because the stream is divided into parts, and the accumulator function is applied to each one independently, which means that 4 is added not only to the first element, but to the first element of each part.

Also, as we also saw in the last chapter, the version of `reduce()` that takes three arguments is particularly useful in parallel streams.

For instance, if we take the example from the last chapter that sums the length of some strings and make it parallel:

```
int length = Stream.of("Parallel", "streams", "are", "great")
    .parallel()
    .reduce(0,
        (accumInt, str) -> accumInt + str.length(), //accumulator
        (accumInt1, accumInt2) -> accumInt1 + accumInt2); //combiner
```

We can see what happens to get the result (23):

Chapter EIGHTEEN

1. The accumulator function is applied to each element in no particular order. For example:

```
0 + 5    // great
0 + 3    // are
0 + 8    // Parallel
0 + 7    // streams
```

2. The results of the accumulator function are combined:

```
5 + 3    // 8
8 + 7    // 15
8 + 15   // 23
```

When we are returning a reduced value of type U from elements of type T, and in a parallel stream the elements are divided into N intermediate results of type U, it makes sense to have a function that knows how to combine those values of type U into a single result.

Because of that, if we are not using different types, the accumulator function **IS** the same as the combiner function.

Finally, just like reduce(), we can safely use collect() with parallel streams if we follow the same requirements of associativity and identity, like for example, for any partially accumulated result, combining it with an empty result container must produce an equivalent result.

Or, if we are grouping with the Collectors class and ordering is not important, we can use the method groupingByConcurrent(), the concurrent version of groupingBy().

KEY POINTS

- Parallel streams split the stream into multiple parts. Each part is processed by a different thread at the same time (in parallel).
- To create a parallel stream from another stream, use the `parallel()` method.
- To create a parallel stream from a Collection use the `parallelStream()` method.
- Parallel streams are more appropriate for operations where the order of processing doesn't matter and that don't need to keep a state (they are stateless and independent).
- You can turn a parallel stream into a sequential one with the `sequential()` method.
- You can check if a stream is parallel with `isParallel()`.
- You can turn an ordered stream into a unordered one (or ensure that the stream is unordered) with `unordered()`.
- Parallel streams don't always perform better than sequential streams.
- With parallel streams, `reduce()` creates intermediate values and then combines them, avoiding the "ordering" problem while still allowing streams to be processed in parallel by eliminating shared (mutating) state and keeping it inside the reduction process.
- The only requirement is that the applied reducing operation must be *associative*: $(a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c)$.

SELF TEST

1. Given:

```
public class Question_18_1 {  
    public static void main(String[] args) {  
        OptionalInt sum = IntStream.rangeClosed(1, 10)  
            .parallel()  
            .unordered()  
            .reduce(Integer::sum);  
        System.out.println(sum.orElse(0));  
    }  
}
```

What is the result?

- A. 0
- B. 55 is printed all the time
- C. Sometimes 55 is printed
- D. An exception occurs at runtime

2. Which of the following statements is true?

- A. You can call the method `parallel()` in a Collection to create a parallel stream.
- B. Operations that are computationally expensive, generally have a better performance using a sequential stream.
- C. `filter()` is a stateless method.
- D. Parallel streams always perform better than sequential streams.

3.

Given:

```
public class Question_18_3 {  
    public static void main(String[] args) {  
        OptionalDouble avg = IntStream.rangeClosed(1, 10)  
            .parallel()  
            .average();  
        System.out.println(avg.getAsDouble());  
    }  
}
```

What is the result?

- A. 5.5 is printed all the time
- B. Sometimes 5.5 is printed
- C. Compilation fails
- D. An exception occurs at runtime

4. Given:

```
public class Question_18_4 {  
    public static void main(String[] args) {  
        IntStream.of(1, 1, 3, 3, 7, 6, 7)  
            .distinct()  
            .parallel()  
            .map(i -> i*2)  
            .sequential()  
            .forEach(System.out::print)  
    }  
}
```

What is the result?

- A. It can print 142612 sometimes
- B. 1133767
- C. It can print 3131677 sometimes
- D. 261412

ANSWERS

1. The correct answer is B.

Sums are associative operations, therefore, the `reduce()` method ensures that the stream can be processed in parallel while preserving the order of its elements.

2. The correct answer is C.

Option A is false. Collections use the `parallelStream()` method. Option B is false. Operations that are computationally expensive, generally have a better performance using a parallel stream.

Option C is true. `filter()` is considered stateless.

Option D is false. Parallel streams don't always perform better than sequential streams.

3. The correct answer is A.

If you remember from the previous chapter, the `average()`, `sum()`, `max()`, and `min()` methods are implemented as a reduce operation with the `collect()` method, so the stream can be safely processed in parallel.

4. The correct answer is D.

The call of `sequential()` turns the whole stream pipeline into a sequential one. In general, we can say that the last call to `parallel()` or `sequential()` is the one that affects the whole stream pipeline.

Part FIVE

Exceptions and Assertions

Chapter NINETEEN

Exceptions

Exam Objectives

- Use try-catch and throw statements
- Use catch, multi-catch, and finally clauses
- Use Autoclose resources with a try-with-resources statement
- Create custom exceptions and Auto-closeable resources

EXCEPTIONS

Errors can (and will) happen in any program. In Java, errors are represented by exceptions.

Basically, in Java there are three types of exception:

java.lang.Exception

Extends from `java.lang.Throwable` and represents errors that are expected. In some cases, the program can recover itself from them. Some examples are:
`IOException`, `ParseException`, `SQLException`

java.lang.RuntimeException

Extends from `java.lang.Exception` and represents unexpected errors generated at runtime. In most cases, the program cannot recover itself from them. Some examples are:

`ArithmetricException`, `ClassCastException`, `NullPointerException`

java.lang.Error

Extends from `java.lang.Throwable` and represents serious problems or abnormal conditions that a program should not deal with. Some examples are:

`AssertionError`, `IOError`, `LinkageError`, `VirtualMachineError`

`RuntimeException` and its subclasses are not required to be caught since they're not expected all the time. They're also called unchecked.

`Exception` and its subclasses (except for `RuntimeException`) are known as checked exceptions because the compiler has to check if they are caught at some point by a `try-catch` statement.

TRY-CATCH BLOCK

```
try {  
    // Code that may throw  
    // an exception  
  
} catch(Exception e) {  
    // Do something with  
    // the exception using  
    // reference e  
  
}
```

THERE'S ONLY
ONE TRY
BLOCK

THERE CAN BE MORE THAN ONE CATCH
BLOCK (ONE FOR EACH EXCEPTION TO
CATCH)

TRY-CATCH BLOCK

A try block is used to enclose code that might throw an exception, it doesn't matter if it's a checked or an unchecked one.

A catch block is used to handle an exception. It defines the type of the exception and a reference to it.

Let's see an example:

```
class Test {  
    public static void main(String[] args) {  
        int[] arr = new int[3];  
        for(int i = 0; i <= arr.length; i++) {  
            arr[i] = i * 2;  
        }  
        System.out.println("Done");  
    }  
}
```

There's an error in the above program, can you see it?

In the last iteration of the loop, `i` will be 3, and since arrays have zero-based indexes, an exception will be thrown at runtime:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3  
at com.example.Test.main(TestException.java:8)
```

If an exception is not handled, the JVM provides a default exception handler that performs the following tasks:

1. It prints out exception description.
2. It prints the stack trace (hierarchy of methods where the exception occurred).
3. It causes the program to terminate.

However, if the exception is handled by in a try-catch block, the normal flow of the application is maintained and rest of the code is executed.

```
class Test {  
    public static void main(String[] args) {  
        try {  
            int[] arr = new int[3];  
            for(int i = 0; i <= arr.length; i++) {  
                arr[i] = i * 2;  
            }  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception caught");  
        }  
        System.out.println("Done");  
    }  
}
```

The output:

```
Exception caught  
Done
```

This is an example of an unchecked exception. Again, they don't have to be caught, but catching them is certainly useful.

Chapter NINETEEN

On the other hand, we have checked exceptions, which need to be surrounded by a `try` block if you don't want the compiler to complain. So this piece of code:

```
SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
Date date = sdf.parse("01-10"); // Compile-time error
System.out.println(date);
```

Becomes this:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.err.println("ParseException caught");
}
```

Since, according to its signature, the `parse` method throws a `java.text.ParseException` (that extends directly from `java.lang.Exception`):

```
public Date parse(String source) throws ParseException
```

The `throws` keyword indicates the exceptions that a method can throw. Only checked exceptions are required to be declared this way.

Now, remember not to confuse `throws` with `throw`. The latter will actually throw an exception:

```
public void myMethod() throws SQLException {
    // Exceptions are created with the new operator like any Java class
    throw new SQLException();
}
```

We can also catch the superclass directly:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (Exception e) {
    System.err.println("Exception caught");
}
```

Although this is not recommended, since the above catch block will catch every exception (checked or unchecked) that could be possibly thrown by the code.

So it's better to catch both in this way:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (ParseException e) {
    System.err.println("ParseException caught");
} catch (Exception e) {
    System.err.println("Exception caught");
}
```

If an exception can be caught in more than one block, the exception will be caught in the first block defined.

However, we have to respect the hierarchy of the classes, if a superclass is defined before a subclass, a compile-time error is generated:

Chapter NINETEEN

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (Exception e) {
    System.err.println("Exception caught");
} catch (ParseException e) {
    System.err.println("ParseException caught");
}
```

An error is also generated if a catch block is defined for an exception that couldn't be thrown by the code in the try block:

```
try {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");
    Date date = sdf.parse("01-10");
    System.out.println(date);
} catch (SQLException e) { // Compile-time error
    System.err.println("ParseException caught");
}
```

The reason of these two errors is that the code of both catch blocks will never be executed (it's unreachable, as the compiler says).

In one case, the catch block with the superclass will be executed for all exceptions that belong to that type and in the other case, the exception can never be possible thrown and the catch block can never be possible executed.

Finally, if the code that throws a checked exception is not inside a try-catch block, the method that contains that code must declare the exception in the throws clause.

In this case, the caller of the method must either catch the exception or also declare it in the throws clause and so on until the main method of the program is reached:

```
public class Test {  
    public static void main(String[] args) throws ParseException {  
        m1();  
    }  
  
    private static void m1() throws ParseException {  
        m2();  
    }  
  
    private static void m2() throws ParseException {  
        m3();  
    }  
  
    private static void m3() throws ParseException {  
        m4();  
    }  
  
    private static void m4() throws ParseException {  
        SimpleDateFormat sdf = new SimpleDateFormat("MM/dd");  
        Date date = sdf.parse("01-10");  
        System.out.println(date);  
    }  
}
```

MULTI-CATCH

```
try {  
  
    // Code that may throw one or  
    // two exceptions  
  
} catch(Exception1 | Exception2 e) {  
  
    // Do something with the caught  
    // exception using reference e  
  
}
```

CATCH EITHER EXCEPTION1 OR
EXCEPTION2

FINALLY

```
try {  
  
    // Code that may throw an  
    // exception  
  
} finally {  
  
    // Block that is always executed  
  
}
```

THE CATCH BLOCK IS OPTIONAL. YOU CAN HAVE BOTH OR EITHER A CATCH BLOCK OR A FINALLY BLOCK

THE FINALLY BLOCK IS ALWAYS EXECUTED, NO MATTER IF AN EXCEPTION IS THROWN IN THE TRY BLOCK, RE-THROWN INSIDE THE CATCH BLOCK, OR NOT CAUGHT AT ALL

MULTI-CATCH AND FINALLY

Consider something like the following code:

```
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0)
        ? 10 / arr[1]
        : 10 * arr[2];
} catch (ArithmetricException e) {
    e.printStackTrace();
    return res;
} catch (IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
return res;
```

Isn't that ugly? I mean to have two catch blocks with the same code. Fortunately, the multi-catch block allows us to catch two or more exception with a single catch block:

```
try {
    ...
} catch (ArithmetricException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
```

Think of the pipe character as an **OR** operator. Also, notice there's only one variable at the end of the catch clause for all the exceptions declared. If you want to differentiate between exceptions, you can use the instanceof operator:

```
try {
    ...
} catch (ArithmetricException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmetricException) {
        // Do something else if the exception type
        // is ArithmetricException
    }
    e.printStackTrace();
    return res;
}
```

Also, the variable is treated as final, which means that you can't reassign (why would you want anyway?):

```
try {
    ...
} catch (ArithmetricException | IndexOutOfBoundsException e) {
    if(e instanceof ArithmetricException) {
        // Compile-time error
        e = new ArithmetricException("My Exception");
    }
} catch(Exception e) {
    e = new Exception("My Exception"); // It compiles!
    throw e;
}
```

One last rule. You cannot combine subclasses and their superclasses in the same multi-catch block:

Chapter NINETEEN

```
try {
    ...
} catch (ArithmetricException | RuntimeException e) {
    // The above line generates a compile-time error
    // because ArithmetricException is a subclass of
    // RuntimeException
    e.printStackTrace();
    return res;
}
```

This is similar to the case when a superclass is declared in a catch block before the subclass. The code is redundant, the superclass will always catch the exception.

Back to this piece of code:

```
int res = 0;
try {
    int[] arr = new int[2];
    res = (arr[1] != 0)
        ? 10 / arr[1]
        : 10 * arr[2];
} catch (ArithmetricException | IndexOutOfBoundsException e) {
    e.printStackTrace();
    return res;
}
return res;
```

Since the value of res is always returned, we can use a finally block:

```
try {
    ...
} catch (ArithmetricException | IndexOutOfBoundsException e) {
    e.printStackTrace();
} finally {
    return res;
}
```

The `finally` block is **ALWAYS** executed, even when an exception is caught or when either the `try` or `catch` block contains a `return` statement. For that reason, it's commonly used to close resources like database connections or file handles.

There's only one exception to this rule. If you call `System.exit()`, the program will terminate abnormally without executing the `finally` block. However, as it's considered bad practice to call `System.exit()`, this rarely happens.

TRY-WITH-RESOURCES

```
try (AutoCloseableResource r =  
     // Code that may throw  
     // an exception  
) catch(Exception e) {  
    // Handle exception  
} finally {  
    // Always executes  
}
```

RESOURCE THAT IS CLOSED
AUTOMATICALLY

```
new AutoCloseableResource( ) ) {
```

RESOURCE IS CLOSED AFTER THE TRY
BLOCK FINISHES

CATCH AND FINALLY BLOCKS ARE BOTH
OPTIONAL IN A TRY-WITH-RESOURCES

TRY-WITH-RESOURCES

As we said before, the finally block is generally used to close resources. Since Java 7, we have the try-with-resources block, in which the try block, one or more resources are declared so they can be closed without doing it explicitly in a finally block:

```
try (BufferedReader br
      = new BufferedReader(new FileReader("/file.txt"))) {
    int value = 0;
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

In the example, the BufferedReader is closed after the try block finishes its execution. This would be equivalent to:

```
BufferedReader br = null;
try {
    int value = 0;
    br = new BufferedReader(new FileReader("/file.txt"));
    while((value = br.read()) != -1) {
        System.out.println((char)value);
    }
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
```

```

        if (br != null) br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

If you declare more than one resource, they have to be separated by a semicolon:

```

try (FileReader fr = new FileReader("/file.txt");
     BufferedReader br = new BufferedReader(fr)) {
    ...
}

```

Also, resources declared inside a try-with-resources cannot be used outside this block (first reason, they're out of scope, second reason, they're closed after the try block ends):

```

try (BufferedReader br
      = new BufferedReader(new FileReader("/file.txt"))) {
    ...
}
String line = br.readLine(); // Compile-time error

```

Now, don't think any class will work in a try-with-resources.

```

class MyResource {
    void useResource() { }
}

try (MyResource r = new MyResource()) { // Compile-time error
    r.useResource()
}

```

Chapter NINETEEN

The class(es) used in a try-with-resources block must implement one of the following interfaces:

- `java.lang.AutoCloseable`
- `java.io.Closeable`

They both declare a `close()` method, and the only practical difference between these two interfaces is that the `close` method of the `Closeable` interface only throws exceptions of type `IOException`:

```
void close() throws IOException;
```

While the `close()` method of the `AutoCloseable` interface throws exceptions of type `Exception` (in other words, it can throw almost any kind of exception):

```
void close() throws Exception;
```

So the `close()` method is called automatically, and if this method actually throws and exception, we can catch it in the catch block.

```
class MyResource implements AutoCloseable {  
    public void close() throws Exception {  
        int x = 0;  
        //...  
        if(x == 1) throw new Exception("Close Exception");  
    }  
    void useResource() {}  
}  
...  
try (MyResource r = new MyResource()) { // Problem gone!  
    r.useResource();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

But what happens if the `try` block also throws an exception?

Well, the result is that the exception from the `try` block "wins" and the exceptions from the `close` method are "suppressed".

In fact, you can retrieve these suppressed exceptions by calling the `Throwable[] java.lang.Throwable.getSuppressed()` method from the exception thrown by the `try` block.

```
try (MyResource r = new MYResource()) {  
    r.useResource();  
    throw new Exception("Exception inside try");  
} catch (Exception e) {  
    System.err.println(e.getMessage());  
    Stream.of(e.getSuppressed())  
        .forEach(t -> System.err.println(t.getMessage()));  
}
```

The output (assuming the `close()` method throws an exception):

```
Exception inside try  
Close Exception
```

CUSTOM EXCEPTIONS

Since exceptions are classes, we can just extend any exception of the language to create our own exceptions.

If you want to force the catching of your exception, extend from `Exception` or one of its subclasses. If you don't want to force it, extend from `RuntimeException` or one of its subclasses.

```
class TooHardException extends Exception {
    public TooHardException(Exception e) {
        super(e);
    }
}
class TooEasyException extends RuntimeException {}
```

As you can see, it's a convention to add `Exception` to your classes' name. The `Error` and `Throwable` classes are not actually used for custom exceptions.

The main members of the `Exception` class that you'd want to know are:

Default constructor	<code>Exception()</code>
Constructor that takes a message	<code>Exception(String)</code>
Constructor that takes another exception (that represents the cause)	<code>Exception(Throwable)</code>
Returns exception's message	<code>String getMessage()</code>
Returns (if any) the exception's cause	<code>Throwable getCause()</code>
Returns the list of suppressed exceptions	<code>Throwable[] getSuppressed()</code>
Prints the stack trace (cause and suppressed exceptions included)	<code>void printStackTrace()</code>

KEY POINTS

- In Java, there are three types of exception
 - `java.lang.Exception`
 - `java.lang.RuntimeException`
 - `java.lang.Error`
- `RuntimeException` and its subclasses are not required to be caught since they're not expected all the time. They're also called unchecked.
- `Exception` and its subclasses (except for `RuntimeException`) are known as checked exceptions because the compiler has to check if they are caught at some point by a try-catch statement.
- If an exception can be caught in more than one block, the exception will be caught in the first block defined.
- However, we have to respect the hierarchy of the classes, if a superclass is defined before a subclass, a compile-time error is generated.
- If the code that throws a checked exception is not inside a try-catch block, the method that contains that code must declare the exception in the throws clause.
- In this case, the caller of the method must either catch the exception or also declare it in the throws clause and so on until the main method of the program is reached.
- The multi-catch block allows us to catch two or more unrelated exceptions with a single catch block:

Chapter NINETEEN

```
try {
    // ...
} catch(Exception1 | Exception2 e) {
    // ...
}
```

- The finally block is **ALWAYS** executed, even when an exception is caught or when either the try or catch block contains a return statement.
- In a try-with-resources block, one or more resources are declared so they can be closed automatically after the try block ends just by implementing `java.lang.AutoCloseable` or `java.io.Closeable`:

```
try (Resource r = new Resource()) {
    //...
} catch(Exception e) { }
```

- When using a try-with-resources block, catch and finally are optional.
- You can create your own exceptions just by extending from `java.lang.Exception` (for checked exceptions) or `java.lang.RuntimeException` (for unchecked exceptions).

SELF TEST

1. Given:

```
public class Question_19_1 {

    protected static int m1() {
        try {
            throw new RuntimeException();
        } catch(RuntimeException e) {
            return 1;
        } finally {
            return 2;
        }
    }

    public static void main(String[] args) {
        System.out.println(m1());
    }
}
```

What is the result?

- A. 1
- B. 2
- C. Compilation fails
- D. An exception occurs at runtime

Chapter NINETEEN

2. Given:

```
public class Question_19_2 {  
    public static void main(String[] args) {  
        try {  
            // Do nothing  
        } finally {  
            // Do nothing  
        }  
    }  
}
```

What of the following is true?

- A. The code doesn't compile correctly
- B. The code would compile correctly if we add a catch block
- C. The code would compile correctly if we remove the finally block
- D. The code compiles correctly as it is

3. Which of the following statements are true?

- A. In a try-with-resources, the catch block is required.
- B. The throws keyword is used to throw an exception.
- C. In a try-with-resources block, if you declare more than one resource, they have to be separated by a semicolon.
- D. If a catch block is defined for an exception that couldn't be thrown by the code in the try block, a compile-time error is generated.

4. Given:

```
class Connection implements java.io.Closeable {  
    public void close() throws IOException {  
        throw new IOException("Close Exception");  
    }  
}  
  
public class Question_19_4 {  
    public static void main(String[] args) {  
        try (Connection c = new Connection()) {  
            throw new  
                RuntimeException("RuntimeException");  
        } catch (IOException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

What is the result?

- A. Close Exception
- B. RuntimeException
- C. RuntimeException and then CloseException
- D. Compilation fails
- E. The stack trace of an uncaught exception is printed

5. Which of the following exceptions are direct subclasses of RuntimeException?

- A. java.io.FileNotFoundException
- B. java.lang.ArithmetricException
- C. java.lang.ClassCastException
- D. java.lang.InterruptedException

ANSWERS

1. The correct answer is B.

The finally block is always executed before the returning of a method, even if the try or the catch blocks also have a return statement.

2. The correct answer is D.

The code correctly compiles as it is. In a normal try block, the catch and the finally block are optional but either one of the must be present.

3. The correct answers are C and D.

Option A is false. In a try-with-resources, the catch block is not required.

Option B is false. The throws keyword is used to declare which exceptions a method could throw.

Option C is true. In a try-with-resources block, if you declare more than one resource, they have to be separated by a semicolon.

Option D is true. If a catch block is defined for an exception that couldn't be thrown by the code in the try block, a compile-time error is generated.

4. The correct answer is E.

This is the output of the program:

```
Exception in thread "main" java.lang  
RuntimeException: RuntimeException  
        at chapter.twenty.Question_19_4.main(Question_19_4.java:13)  
        Suppressed: java.io.IOException: Close Exception  
                at chapter.twenty.Connection.close(Question_19_4.java:7)  
                at chapter.twenty.Question_19_4.main(Question_19_4.java:15)
```

Catching the IOException thrown by the close method of the class

Connection is required for the program to compile.

However, the `RuntimeException` thrown inside the `try` block is not caught, so the default exception handler takes care of it.

5. The correct answers are B and C.

`java.lang.ArithmetricException` and `java.lang.ClassCastException` are subclasses of `java.lang.RuntimeException`.

Chapter NINETEEN

Chapter TWENTY

Assertions

Exam Objectives

- Test invariants by using assertions.

USING ASSERTIONS

An assertion is a statement used to check if something is true and helps you to detect errors in a program.

An assert statement has the following syntax:

```
assert booleanExpression;
```

If `booleanExpression` evaluates to **FALSE**, an exception of type `java.lang.AssertionError` (a subclass of `Error`) is thrown.

In other words, the above statement is equivalent to:

```
if(booleanExpressionIsFalse) {  
    throw new AssertionException();  
}
```

This is because, as said before, an assertion has to be true. If it's not, an error has to be thrown.

An assert statement can also take a `String` as a message:

```
assert booleanExpression: "Message about the error";
```

Which is equivalent to:

```
if(booleanExpressionIsFalse) {  
    throw new AssertionException("Message about the error");  
}
```

Chapter TWENTY

The thing is, assertions are **NOT** enabled by default.

You can have assertion all over you code, but if they are disabled, Java will skip them all.

In summary:

- If assertions are enabled and the boolean expression is true, nothing happens.
- If assertions are enabled and the boolean expression is false, an `AssertionError` is thrown.
- If assertions are disabled, no matter the outcome of the boolean expression, assertions are ignored.

Assertions are enabled in the command-line with either:

```
java -ea MainClass
```

Or

```
java --enableassertions MainClass
```

This would enable assertions in all the classes of our program except for the classes of Java (or system classes).

For example, if we have this class:

```
public TestAssertion {  
    public static void main(String[] args) {  
        // Parentheses are optional  
        assert (args.length > 0) :  
            "At least one argument is required";  
        System.out.println(args[0]);  
    }  
}
```

And we run it this way:

```
java -ea TestAssertion
```

The output will be:

```
Exception in thread "main" java.lang.AssertionError: At least  
one argument is required  
at com.example.TestAssertion.main(TestAssertion.java:7)
```

If we run it this way:

```
java -ea TestAssertion Hi
```

The output will be:

Hi

If we run it this way:

```
java TestAssertion
```

The output will be:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  
at com.example.TestAssertion.main(TestAssertion.java:8)
```

As you can see, assertions can help us catch some errors derived from assumptions we make in our program. Just be aware that they are not enabled by default. They are not designed to be run in production, for that, it's better to use regular exceptions.

Chapter TWENTY

Assertions are better used to validate parameters, properties, preconditions, postconditions, application flow of control (for points of the code that should never be reached), and in general, error checking that otherwise, you would have to comment or disabled when the code is ready for production.

Assertions can also be enabled for one class:

```
java -ea:ClassName MainClass
```

Or for all classes in a package:

```
java -ea:com.example MainClass
```

For enabling assertions in the unnamed package (when you don't use a package statement):

```
java -ea:... MainClass
```

Or in the rare case you'd want to enable assertions for the system classes:

```
java -esa MainClass
```

Or

```
java --enableassertions MainClass
```

There's also a command to disable assertions:

```
java -da MainClass
```

Or

```
java -disableassertions MainClass
```

With the same options that -ea. To disable for one class:

```
java -da:ClassName MainClass
```

For disabling assertions in a package:

```
java -da:com.example MainClass
```

For enabling assertion in the unnamed package:

```
java -ea:... MainClass
```

For the system classes:

```
java -dsa MainClass
```

Or

```
java -disablesystemassertions MainClass
```

This option exists because there will be times when you want to use commands like:

```
java -ea -da:ClassOne MainClass
```

To enable assertions in all the program classes except for Class1.

KEY POINTS

- An assertion is a statement used to check if something is true and helps you to detect errors in a program.

- An assert statement has the following syntax:

```
assert booleanExpression;  
assert booleanExpression: "Message about the error";
```

- If the boolean expression of the assert statement evaluates to false, the program will throw a `java.lang.AssertionError` and terminate.
- By default, assertions are disabled. You can use the command-line arguments `-ea` (for enabling assertions) and `-da` (for disabling assertions) with other options for classes, packages, and system classes when running the program.

SELF TEST

1. Given:

```
public class Question_20_1 {  
    public static void main(String[] args) {  
        int x = 0;  
        assert ++x > 0 : x;  
    }  
}
```

What is the result when this program is executed with assertions enabled?

- A. 0
- B. 1
- C. Nothing is printed
- D. An exception is thrown at runtime

2. Which of the following statements is true?

- A. You can disable assertion at the command-line.
- B. Assertions are enabled by default in system classes.
- C. Even if assertions are disabled, if the boolean expression evaluates to false, a runtime error is thrown.
- D. Depending on the syntax, assertions can throw either a checked or a runtime exception.

Chapter TWENTY

3. Given:

```
java -esa -ea:com.example MainClass
```

Which of the following statements is true?

- A. This command enables assertions in all classes of our program.
- B. This command enables assertions in system classes and classes in the com.example package.
- C. This command enables assertions in system classes in just MainClass.
- D. This command enables assertions in system classes of the package com.example.

4. Given:

```
public class Question_20_4 {  
    public static void main(String[] args) {  
        assert isValid();  
    }  
    public static boolean isValid() {  
        return false;  
    }  
}
```

What is the result when this program is executed with assertions enabled?

- A. false
- B. Nothing is printed
- C. A compile time error
- D. An AssertionError is thrown at runtime

5. When assertions are enabled, which of the following statements behaves similar way to:

assert false : "Assertion is false";

- A. throw new Assertion("Asssertion is false");
- B. throw new AssertionError("Asssertion is false");
- C. if(false) throw new AssertionError("Asssertion is false");
- D. if(false) throw new RuntimeException("Asssertion is false");

ANSWERS

1. The correct answer is C.

The `++` operator increments the value of `x` to 1 before the condition is evaluated. Note that changing values of variables or objects in assertions is discouraged since there's no guarantee that assertions would be enabled when the program runs.

2. The correct answer is A.

You can disable assertion at the command-line. The other statements are false.

3. The correct answer is B.

The command:

```
java -esa -ea:com.example MainClass
```

Enables assertions in system classes and classes in the `com.example` package.

4. The correct answer is D.

There's nothing wrong with the program, the method `isValid()` returns `false`, which makes the program to throw an `AssertionError` at runtime.

5. The correct answer is B.

```
assert false : "Assertion is false"
```

Behaves in a similar way to:

```
throw new AssertionError("Assertion is false");
```

When assertions are enabled.

Part SIX

Date/Time API

Chapter TWENTY-ONE

Core Date/Time Classes

Exam Objectives

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

A NEW DATE/TIME API

Since the beginning of Java, `java.util.Date` and `java.util.Calendar` (introduced later) have been the classes to use when working with dates and times.

However, these classes are far from perfect, some of their problems are:

- `java.util.Date` represents time with "only" milliseconds precision (which may not be enough in some applications).
- Years start from 1900 and months start at 0.
- The time zone of the date is the JVM's default time zone.
- Both `java.util.Date` and `java.util.Calendar` are mutable classes, meaning that when they change, they don't create another instance with the new values (which is not ideal now that you can program in a functional style with Java).

For those reasons, Java 8 introduced a new Date/Time API based on the popular date/time library Joda-Time and contained in the also new `java.time` package.

This chapter will deal with the core classes of the new API, that don't provide time zone information. The classes that do provide time zone information will be the topic of the next chapter.

Let's start with a high-level overview of the core classes.

All these classes are immutable, thread-safe, and with the exception of `Instant`, they don't store or represent a time-zone.

Chapter TWENTY-ONE

On the one hand, we have:

LocalDate

Represents a date with the year, month, and day of the month information. For example, `2015-08-25`.

LocalTime

Represents a time with hour, minutes, seconds, and nanoseconds information. For example, `13:21.05.123456789`.

LocalDateTime

A combination of the above. For example, `2015-08-25 13:21.05.12345`.

On the other hand:

Instant

Represents a single point in time in seconds and nanoseconds. For example `923,456,789 seconds and 186,054,812 nanoseconds`.

Period

Represents an amount of time in terms of years, months and days. For example, `5 years, 2 months and 9 days`.

Duration

Represents an amount of time in terms of seconds and nanoseconds. For example, `12.87656 seconds`.

`LocalDate`, `LocalTime`, `LocalDateTime` and `Instant` implement the interface `java.time.temporal.Temporal`, so they all have similar methods.

While `Period` and `Duration` implement the interface `java.time.temporal.TemporalAmount`, which also makes them very similar.

LOCALDATE CLASS

The key to learning how to use this class is to have in mind that it holds the year, month, day and derived information of a date. All of its methods use this information or have a version to work with each of them.

The following are the most important (most used) methods of this class.

To create an instance, we can use the static method of:

```
// With year(-999999999 to 999999999),
// month (1 to 12), day of the month (1 - 31)
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);
// This version uses the enum java.time.Month
LocalDate newYear2002 = LocalDate.of(2002, Month.JANUARY, 1);
```

Notice that unlike `java.util.Date`, months start from one. If you try to create a date with invalid values (like February, 29), an exception will be thrown. For today's date use `now()`:

```
LocalDate today = LocalDate.now();
```

Once we have an instance of `LocalDate`, we can get the year, the month, and the day with methods like the following:

```
int year = today.getYear();
int month = today.getMonthValue();
Month monthAsEnum = today.getMonth(); // as an enum
int dayYear = today.getDayOfYear();
int dayMonth = today.getDayOfMonth();
DayOfWeek dayWeekEnum = today.getDayOfWeek(); //as an enum
```

Chapter TWENTY-ONE

We can also use the get method:

```
int get(java.time.temporal.TemporalField field) // value as int  
long getLong(java.time.temporal.TemporalField field) // As long
```

Which takes an implementation of the interface `java.time.temporal.TemporalField` to access a specific field of a date. `java.time.temporal.ChronoField` is an enumeration that implements this interface, so we can have for example:

```
int year = today.get(ChronoField.YEAR);  
int month = today.get(ChronoField.MONTH_OF_YEAR);  
int dayYear = today.get(ChronoField.DAY_OF_YEAR);  
int dayMonth = today.get(ChronoField.DAY_OF_MONTH);  
int dayWeek = today.get(ChronoField.DAY_OF_WEEK);  
long dayEpoch = today.getLong(ChronoField.EPOCH_DAY);
```

The supported values for `ChronoField` are:

- `DAY_OF_WEEK`
- `ALIGNED_DAY_OF_WEEK_IN_MONTH`
- `ALIGNED_DAY_OF_WEEK_IN_YEAR`
- `DAY_OF_MONTH`
- `DAY_OF_YEAR`
- `EPOCH_DAY`
- `ALIGNED_WEEK_OF_MONTH`
- `ALIGNED_WEEK_OF_YEAR`
- `MONTH_OF_YEAR`
- `PROLEPTIC_MONTH`
- `YEAR_OF_ERA`
- `YEAR`
- `ERA`

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a `LocalDate` against another instance, we have three methods plus another one for leap years:

```
boolean after = newYear2001.isAfter(newYear2002); // false
boolean before = newYear2001.isBefore(newYear2002); // true
boolean equal = newYear2001.equals(newYear2002); // false
boolean leapYear = newYear2001.isLeapYear(); // false
```

Once an instance of this class is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the `with()` method and its versions:

```
LocalDate newYear2003 = newYear2001.with(ChronoField.YEAR, 2003);
LocalDate newYear2004 = newYear2001.withYear(2004);
LocalDate december2001 = newYear2001.withMonth(12);
LocalDate february2001 = newYear2001.withDayOfYear(32);
// Since these methods return a new instance, we can chain them!
LocalDate xmas2001 = newYear2001.withMonth(12).withDayOfMonth(25);
```

Another way is by adding or subtracting guess what? Years, months, days, or even weeks:

```
// Adding
LocalDate newYear2005 = newYear2001.plusYears(4);
LocalDate march2001 = newYear2001.plusMonths(2);
LocalDate january15_2001 = newYear2001.plusDays(14);
LocalDate lastWeekJanuary2001 = newYear2001.plusWeeks(3);
LocalDate newYear2006 = newYear2001.plus(5, ChronoUnit.YEARS);
```

Chapter TWENTY-ONE

```
// Subtracting
LocalDate newYear2000 = newYear2001.minusYears(1);
LocalDate nov2000 = newYear2001.minusMonths(2);
LocalDate dec30_2000 = newYear2001.minusDays(2);
LocalDate lastWeekDec2001 = newYear2001.minusWeeks(1);
LocalDate newYear1999 = newYear2001.minus(2, ChronoUnit.YEARS);
```

Notice that the plus and minus versions take a `java.time.temporal.ChronoUnit` enumeration, different than `java.time.temporal.ChronoField`. The supported values are:

- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Finally, the method `toString()` returns the date in the format `uuuu-MM-dd`:

```
System.out.println(newYear2001.toString()); // Prints 2001-01-01
```

LOCALTIME CLASS

The key to learning how to use this class is to have in mind that it holds the hour, minutes, seconds, and nanoseconds. All of its methods use this information or have a version to work with each of them.

The following are the most important (most used) methods of this class. As you can see, they are the same (or very similar) methods of LocalDate, adapted to work with time instead of date.

To create an instance, we can use the static method of:

```
// With hour (0-23) and minutes (0-59)
LocalTime fiveThirty = LocalTime.of(5, 30);
// With hour, minutes, and seconds (0-59)
LocalTime noon = LocalTime.of(12, 0, 0);
// With hour, minutes, seconds, and nanoseconds (0-999,999,999)
LocalTime almostMidnight = LocalTime.of(23, 59, 59, 999999);
```

If you try to create a time with an invalid value (like LocalTime.of(24, 0)), an exception will be thrown. To get the current time use now():

```
LocalTime now = LocalTime.now();
```

Once we have an instance of LocalTime, we can get the hour, the minutes, and other information with methods like the following:

```
int hour = now.getHour();
int minute = now.getMinute();
int second = now.getSecond();
int nanosecond = now.getNano();
```

Chapter TWENTY-ONE

We can also use the get() method:

```
int get(java.time.temporal.TemporalField field) // value as int  
long getLong(java.time.temporal.TemporalField field) // As long
```

Just like in the case of LocalDate, we can have, for example:

```
int hourAMPM = now.get(ChronoField.HOUR_OF_AMPM); // 0 - 11  
int hourDay = now.get(ChronoField.HOUR_OF_DAY); // 0 - 23  
int minuteDay = now.get(ChronoField.MINUTE_OF_DAY); // 0 - 1,439  
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR); // 0 - 59  
int secondDay = now.get(ChronoField.SECOND_OF_DAY); // 0 - 86,399  
int secondMinute = now.get(ChronoField.SECOND_OF_MINUTE); // 0 - 59  
long nanoDay = now.getLong(ChronoField.NANO_OF_DAY); // 0-86399999999  
int nanoSecond = now.get(ChronoField.NANO_OF_SECOND); // 0-999999999
```

The supported values for ChronoField are:

- NANO_OF_SECOND
- NANO_OF_DAY
- MICRO_OF_SECOND
- MICRO_OF_DAY
- MILLI_OF_SECOND
- MILLI_OF_DAY
- SECOND_OF_MINUTE
- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a time object against another one, we have three methods:

```
boolean after = fiveThirty.isAfter(noon); // false
boolean before = fiveThirty.isBefore(noon); // true
boolean equal = noon.equals(almostMidnight); // false
```

Like `LocalDate`, once an instance of `LocalTime` is created we cannot modify it, but we can create another instance from an existing one.

One way is through the `with` method and its versions:

```
LocalTime ten = noon.with(ChronoField.HOUR_OF_DAY, 10);
LocalTime eight = noon.withHour(8);
LocalTime twelveThirty = noon.withMinute(30);
LocalTime thirtyTwoSeconds = noon.withSecond(32);
// Since these methods return a new instance, we can chain them!
LocalTime secondsNano = noon.withSecond(20).withNano(999999);
```

Of course, another way is by adding or subtracting hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalTime sixThirty = fiveThirty.plusHours(1);
LocalTime fiveForty = fiveThirty.plusMinutes(10);
LocalTime plusSeconds = fiveThirty.plusSeconds(14);
LocalTime plusNanos = fiveThirty.plusNanos(99999999);
LocalTime sevenThirty = fiveThirty.plus(2, ChronoUnit.HOURS);
```

Chapter TWENTY-ONE

```
// Subtracting
LocalTime fourThirty = fiveThirty.minusHours(1);
LocalTime fiveTen = fiveThirty.minusMinutes(20);
LocalTime minusSeconds = fiveThirty.minusSeconds(2);
LocalTime minusNanos = fiveThirty.minusNanos(1);
LocalTime fiveTwenty = fiveThirty.minus(10, ChronoUnit.MINUTES);
```

Notice that the plus and minus versions take a `java.time.temporal.ChronoUnit` enumeration, different than `java.time.temporal.ChronoField`. The supported values are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS

Finally, the method `toString()` returns the time in the format `HH:mm:ss.SSSSSSSS`, omitting the parts with value zero (for example, just returning `HH:mm` if it has zero seconds/nanoseconds):

```
System.out.println(fiveThirty.toString()); // Prints 05:30
```

LOCALDATETIME CLASS

The key to learning how to use this class is to remember that it combines LocalDate and LocalTime classes.

It represents both a date and a time, with information like year, month, day, hours, minutes, seconds, and nanoseconds. Other fields, such as day of the year, day of the week, and week of year can also be accessed.

To create an instance, we can use either the static method of() or from a LocalDate or LocalTime instance:

```
// Setting seconds and nanoseconds to zero
LocalDateTime dt1 = LocalDateTime.of(2014, 9, 19, 14, 05);
// Setting nanoseconds to zero
LocalDateTime dt2 = LocalDateTime.of(2014, 9, 19, 14, 05, 20);
// Setting all fields
LocalDateTime dt3 = LocalDateTime.of(2014, 9, 19, 14, 05, 20, 9);
// Asuming this date
LocalDate date = LocalDate.now();
// And this time
LocalTime time = LocalTime.now();
// Combine the above date with the given time like this
LocalDateTime dt4 = date.atTime(14, 30, 59, 999999);
// Or this
LocalDateTime dt5 = date.atTime(time);
// Combine this time with the given date. Notice that LocalTime
// only has this method to be combined with a LocalDate
LocalDateTime dt6 = time.atDate(date);
```

Chapter TWENTY-ONE

If you try to create an instance with an invalid value or date, an exception will be thrown. To get the current date/time use `now()`:

```
LocalDateTime now = LocalDateTime.now();
```

Once we have an instance of `LocalDateTime`, we can get the information with the methods we know from `LocalDate` and `LocalTime`, such as:

```
int year = now.getYear();
int dayYear = now.getDayOfYear();
int hour = now.getHour();
int minute = now.getMinute();
```

We can also use the `get()` method:

```
int get(java.time.temporal.TemporalField field)
long getLong(java.time.temporal.TemporalField field)
```

For example:

```
int month = now.get(ChronoField.MONTH_OF_YEAR);
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR);
```

The supported values for `ChronoField` are:

- `NANO_OF_SECOND`
- `NANO_OF_DAY`
- `MICRO_OF_SECOND`
- `MICRO_OF_DAY`
- `ILLI_OF_SECOND`
- `ILLI_OF_DAY`
- `SECOND_OF_MINUTE`

- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY
- DAY_OF_WEEK
- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH
- YEAR_OF_ERA
- YEAR
- ERA

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a `LocalDateTime` object against another one, we have three methods:

```
boolean after = now.isAfter(dt1); // true
boolean before = now.isBefore(dt1); // false
boolean equal = now.equals(dt1); // false
```

Chapter TWENTY-ONE

Once an instance of `LocalTime` is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the `with` method and its versions:

```
LocalDateTime dt7 = now.with(ChronoField.HOUR_OF_DAY, 10);
LocalDateTime dt8 = now.withMonth(8);
// Since these methods return a new instance, we can chain them!
LocalDateTime dt9 = now.withYear(2013).withMinute(0);
```

Another way is by adding or subtracting years, months, days, weeks, hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalDateTime dt10 = now.plusYears(4);
LocalDateTime dt11 = now.plusWeeks(3);
LocalDateTime dt12 = newYear2001.plus(2, ChronoUnit.HOURS);

// Subtracting
LocalDateTime dt13 = now.minusMonths(2);
LocalDateTime dt14 = now.minusNanos(1);
LocalDateTime dt15 = now.minus(10, ChronoUnit.SECONDS);
```

In this case, the supported values for `ChronoUnit` are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Finally, the method `toString()` returns the date/time in the format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSS`, omitting the parts with value zero, for example:

```
System.out.println(dt1.toString()); // Prints 2014-09-19T14:05
```

INSTANT CLASS

Although in practical terms, a `LocalDateTime` instance represents an instant in the time line, there is another class that may be more appropriate.

In Java 8, the `java.time.Instant` class represents an instant in the number of seconds that has passed since the *epoch*, a convention used in UNIX/POSIX systems and set at midnight of January 1, 1970 UTC time.

From that date, time is measured in 86,400 seconds per day. This information is stored as a `long`. The class also supports a nanosecond precision, stored as an `int`.

You can create an instance of this class with the following methods:

```
// Setting seconds
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);
// Setting seconds and nanoseconds (can be negative)
Instant sixSecTwoNanBeforeEpoch = Instant.ofEpochSecond(-6, -2);
// Setting milliseconds after (can be before also) epoch
Instant fiftyMillisecondsAfterEpoch = Instant.ofEpochMilli(50);
```

For the current instance of the system clock use:

```
Instant now = Instant.now();
```

Once we have an instance of `Instance`, we can get the information with the following methods:

Chapter TWENTY-ONE

```
long seconds = now.getEpochSecond(); // Gets the seconds
int nanos1 = now.getNano(); // Gets the nanoseconds
// Gets the value as an int
int milis = now.get(ChronoField.MILLI_OF_SECOND);
// Gets the value as a long
long nanos2 = now.getLong(ChronoField.NANO_OF_SECOND);
```

The supported ChronoField values are:

- NANO_OF_SECOND
- MICRO_OF_SECOND
- MILLI_OF_SECOND
- INSTANT_SECONDS

Using any other value will throw an exception. The same is true when getting a value that doesn't fit into an int with get(TemporalField).

To check an Instant object against another one, we have three methods:

```
boolean after = now.isAfter(fiveSecondsAfterEpoch); // true
boolean before = now.isBefore(fiveSecondsAfterEpoch); // false
boolean equal = now.equals(fiveSecondsAfterEpoch); // false
```

Once an instance of this object is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method:

```
Instant i1 = now.with(ChronoField.NANO_OF_SECOND, 10);
```

Another way is by adding or subtracting seconds, milliseconds, or nanoseconds:

```

// Adding
Instant dt10 = now.plusSeconds(400);
Instant dt11 = now.plusMillis(98622200);
Instant dt12 = now.plusNanos(3000138900);
Instant dt13 = newYear2001.plus(2, ChronoUnit.MINUTES);

// Subtracting
Instant dt14 = now.minusSeconds(2);
Instant dt15 = now.minusMillis(1);
Instant dt16 = now.minusNanos(1);
Instant dt17 = now.minus(10, ChronoUnit.SECONDS);

```

The supported ChronoUnit values are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

Finally, the method `toString()` returns the instance in the format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSS`, for example:

```

// Prints 1970-01-01T00:00:00.050Z
System.out.println(fiftyMilliSecondsAfterEpoch.toString());

```

Notice that it contains zone time information (Z). This is because Instant represents a point in time from the epoch of 1970-01-01Z in the UTC zone time.

PERIOD CLASS

The `java.time.Period` class represents an amount of time in terms of years, months and days.

You can create an instance of this class with the following of methods:

```
// Setting years, months, days (can be negative)
Period period5y4m3d = Period.of(5, 4, 3);
// Setting days (can be negative), years and months will be zero
Period period2d = Period.ofDays(2);
// Setting months (can be negative), years and days will be zero
Period period2m = Period.ofMonths(2);
// Setting weeks (can be negative). The resulting period will
// be in days (1 week = 7 days). Years and months will be zero
Period period14d = Period.ofWeeks(2);
// Setting years (can be negative), days and months will be zero
Period period2y = Period.ofYears(2);
```

A `Period` can be also thought as the difference between two `LocalDates`. Luckily, there's a method that support this concept:

```
LocalDate march2003 = LocalDate.of(2003, 3, 1);
LocalDate may2003 = LocalDate.of(2003, 5, 1);
Period dif = Period.between(march2003, may2003); // 2 months
```

The start date is **INCLUDED**, but **NOT** the end date.

Be careful about how the date is calculated.

First complete months are counted, and then the remaining number of days is calculated. The number of months is then split into years (1 years equals 12 months). A month is considered if the end day of the month is greater than or equal to the start day of the month.

The result of this method can be a negative period if the end is before the start (year, month and day will have a negative sign).

Here are some examples:

```
// dif1 will be 1 year 2 months 2 days
Period dif1 = Period.between(
    LocalDate.of(2000, 2, 10), LocalDate.of(2001, 4, 12));
// dif2 will be 25 days
Period dif2 = Period.between(
    LocalDate.of(2013, 5, 9), LocalDate.of(2013, 6, 3));
// dif3 will be -2 years -3 days
Period dif3 = Period.between(
    LocalDate.of(2014, 11, 3), LocalDate.of(2012, 10, 31));
```

Once we have an instance of Period, we can get the information with the following methods:

```
int days = period5y4m3d.getDays();
int months = period5y4m3d.getMonths();
int year = period5y4m3d.getYears();
int days2 = period5y4m3d.get(ChronoUnit.DAYS);
```

The supported ChronoUnit values are:

- DAYS
- MONTHS
- YEARS

Using any other value will throw an exception.

Chapter TWENTY-ONE

Once an instance of Period is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the `with` method and its versions:

```
Period period8d = period2d.withDays(8);
// Since these methods return a new instance, we can chain them!
Period period2y1m2d = period2d.withYears(2).withMonths(1);
```

Another way is by adding or subtracting years, months, or days:

```
// Adding
Period period9y4m3d = period5y4m3d.plusYears(4);
Period period5y7m3d = period5y4m3d.plusMonths(3);
Period period5y4m6d = period5y4m3d.plusDays(3);
Period period7y4m3d = period5y4m3d.plus(period2y);

// Subtracting
Period period5y4m3d = period5y4m3d.minusYears(2);
Period period5y4m3d = period5y4m3d.minusMonths(1);
Period period5y4m3d = period5y4m3d.minusDays(1);
Period period5y4m3d = period5y4m3d.minus(period2y);
```

Methods `plus` and `minus` take an implementation of the interface `java.time.temporal.TemporalAmount` (i.e. another instance of `Period` or an instance of `Duration`).

Finally, the method `toString()` returns the period in the format PNYNMND, for example:

```
System.out.println(period5y4m3d.toString()); // Prints P5Y4M3D
```

A zero period will be represented as zero days, P0D.

DURATION CLASS

The `java.time.Duration` class is like the `Period` class, the only thing is that it represents an amount of time in terms of seconds and nanoseconds.

You can create an instance of this class with the following of methods:

```
Duration oneDay = Duration.ofDays(1); // 1 day = 86400 seconds
Duration oneHour = Duration.ofHours(1); // 1 hour = 3600 seconds
Duration oneMin = Duration.ofMinutes(1); // 1 minute = 60 seconds
Duration tenSeconds = Duration.ofSeconds(10);
// Set seconds and nanoseconds (if they are outside the range
// 0 to 999,999,999, the seconds will be altered, like below)
Duration twoSeconds = Duration.ofSeconds(1, 1000000000);
// Seconds and nanoseconds are extracted from the passed millis
Duration oneSecondFromMillis = Duration.ofMillis(2);
// Seconds and nanoseconds are extracted from the passed nanos
Duration oneSecondFromNanos = Duration.ofNanos(1000000000);
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
```

Valid values of `ChronoUnit` for the method `Duration.of(long amount, TemporalUnit unit)` are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

Chapter TWENTY-ONE

A Duration can also be created as the difference between two implementations of the interface `java.time.temporal.Temporal`, as long as they support seconds (and for more accuracy, nanoseconds), like `LocalTime`, `LocalDateTime`, and `Instant`. So we can have something like this:

```
Duration dif = Duration.between(  
    Instant.ofEpochSecond(123456789), Instant.ofEpochSecond(99999));
```

The result can be negative if the end is before the start. A negative duration is expressed with a negative sign in the seconds part. For example, a duration of -100 nanoseconds is stored as -1 second plus 999,999,900 nanoseconds.

If the objects are of different types, then the duration is calculated based on the type of the first object. But this only works if the first argument is a `LocalTime` and the second is a `LocalDateTime` (because it can be converted to `LocalTime`). Otherwise, an exception is thrown.

Once we have an instance of `Duration`, we can get the information with the following methods:

```
// Nanoseconds part the duration, from 0 to 999,999,999  
int nanos = oneSecond.getNano();  
// Seconds part of the duration, positive or negative  
int seconds = oneSecond.getSeconds();  
// Supports SECONDS and NANOS. Other units throw an exception  
int oneSec = oneSecond.get(ChronoUnit.SECONDS);
```

Notice that the method `get()` only supports `SECONDS` and `NANOS`.

Once an instance of `Duration` is created, we cannot modify it, but we can create another instance from an existing one. One way is to use the `with` method and its versions:

```
Duration duration1sec8nan = oneSecond.withNanos(8);
Duration duration2sec1nan = oneSecond.withSeconds(2).withNanos(1);
```

Another way is by adding or subtracting days, hours, minutes, seconds, milliseconds, or nanoseconds:

// Adding

```
Duration plus4Days = oneSecond.plusDays(4);
Duration plus3Hours = oneSecond.plusHours(3);
Duration plus3Minutes = oneSecond.plusMinutes(3);
Duration plus3Seconds = oneSecond.plusSeconds(3);
Duration plus3Millis = oneSecond.plusMillis(3);
Duration plus3Nanos = oneSecond.plusNanos(3);
Duration plusAnotherDuration = oneSecond.plus(twoSeconds);
Duration plusChronoUnits = oneSecond.plus(1, ChronoUnit.DAYS);
```

// Subtracting

```
Duration minus4Days = oneSecond.minusDays(4);
Duration minus3Hours = oneSecond.minusHours(3);
Duration minus3Minutes = oneSecond.minusMinutes(3);
Duration minus3Seconds = oneSecond.minusSeconds(3);
Duration minus3Millis = oneSecond.minusMillis(3);
Duration minus3Nanos = oneSecond.minusNanos(3);
Duration minusAnotherDuration = oneSecond.minus(twoSeconds);
Duration minusChronoUnits = oneSecond.minus(1, ChronoUnit.DAYS);
```

Methods `plus` and `minus` take either another `Duration` or a valid `ChronoUnit` value (the same values used to create an instance).

Finally, the method `toString()` returns the duration with the format `PTnHnMnS`. Any fractional seconds are placed after a decimal point in the seconds section. If a section has a zero value, it's omitted. For example:

2 days 4 minutes	PT48H4M
45 seconds 99 milliseconds	PT45.099S

KEY POINTS

- `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration` are the core classes of the new Java Date/Time API, located in the package `java.time`. They are immutable, thread-safe, and with the exception of `Instant`, they don't store or represent a time-zone.
- `LocalDate`, `LocalTime`, `LocalDateTime` and `Instant` implement interface `java.time.temporal.Temporal`, so they all have similar methods. While `Period` and `Duration` implement interface `java.time.temporal.TemporalAmount`, which also makes them very similar.
- `LocalDate` represents a date with the year, month, and day of the month information. You can create an instance using:

```
LocalDate.of(2015, 8, 01);
```

- Valid `ChronoField` values to use with the `get` method are: `DAY_OF_WEEK`, `ALIGNED_DAY_OF_WEEK_IN_MONTH`, `ALIGNED_DAY_OF_WEEK_IN_YEAR`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `EPOCH_DAY`, `ALIGNED_WEEK_OF_MONTH`, `ALIGNED_WEEK_OF_YEAR`, `MONTH_OF_YEAR`, `PROLEPTIC_MONTH`, `YEAR_OF_ERA`, `YEAR`, and `ERA`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `DAYS`, `WEEKS`, `MONTHS`, `YEARS`, `DECades`, `CENTURIES`, `MILLENNIA`, and `ERAS`.
- `LocalTime` represents a time with hour, minutes, seconds, and nanoseconds information. You can create an instance using:

```
LocalTime.of(14, 20, 50, 99999);
```

- Valid `ChronoField` values to use with the `get()` method are: `NANO_OF_SECOND`, `NANO_OF_DAY`, `MICRO_OF_SECOND`, `MICRO_OF_DAY`, `ILLI_OF_SECOND`, `ILLI_OF_DAY`, `SECOND_OF_MINUTE`, `SECOND_OF_DAY`,

`MINUTE_OF_HOUR`, `MINUTE_OF_DAY`, `HOUR_OF_AMPM`, `CLOCK_HOUR_OF_AMPM`, `HOUR_OF_DAY`, `CLOCK_HOUR_OF_DAY`, and `AMPM_OF_DAY`.

- Valid ChronoUnits values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, and `HALF_DAYS`.
- `LocalDateTime` is a combination of `LocalDate` or `LocalTime`. You can create an instance using:

```
LocalDateTime.of(2015, 8, 1, 14, 20, 50, 99999);
```

- Valid ChronoField and ChronoUnits values are a combination of the ones used for `LocalDate` and `LocalTime`.
- `Instant` represents a single point in time in seconds and nanoseconds. You can create an instance using:

```
Instant.ofEpochSecond(134556767, 999999999);
```

- Valid ChronoField values to use with the `get()` method are: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS`.
- Valid ChronoUnits values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`.
- `Period` represents an amount of time in terms of years, months and days. You can create an instance using:

```
Period.of(3, 12, 30);
```

- Valid ChronoUnits values to use with the `get()` method are: `DAYS`, `MONTHS`, `YEARS`.
- `Duration` represents an amount of time in terms of seconds and nanoseconds. You can create an instance using:

```
Duration.ofSeconds(50, 999999);
```

- Valid ChronoUnits values to use with the `of()` method are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`. With the `get()` method, only `NANOS` and `SECONDS` are valid.

SELF TEST

1. Which of the following are valid ways to create a LocalDate object?

- A. LocalDate.of(2014);
- B. LocalDate.with(2014, 1, 30);
- C. LocalDate.of(2014, 0, 30);
- D. LocalDate.now().plusDays(5);

2. Given:

```
LocalDate.of(2014, 1, 02).atTime(14, 30, 59, 999999)
```

Which of the following is the result of executing the above line?

- A. A LocalDate of 2014-01-02
- B. A LocalTime of 14:30:59:999999
- C. A LocalDateTime of 2014-01-02 14:30:59:999999
- D. An exception is thrown

3. Which of the following are valid ChronoUnit values for LocalTime?

- A. YEAR
- B. NANOS
- C. DAY
- D. HALF_DAYS

4. Which of the following statements are true?

- A. java.time.Period implements java.time.temporal.Temporal
- B. java.time.Instant implements java.time.temporal.Temporal
- C. java.time.LocalDate and java.time.LocalTime are thread-safe
- D. LocalDateTime.now() will return the current time in UTC zone

5. Which of the following is a valid way to get the nanoseconds part of an Instant object referenced by i?

- A. int nanos = i.getNano();
- B. long nanos = i.get(ChronoField.NANOS);
- C. long nanos = i.get(ChronoUnit.NANOS);
- D. int nanos = i.getEpochNano();

6. Given:

```
System.out.println( Period.between(  
    LocalDate.of(2015, 3, 20), LocalDate.of(2015, 2, 20)) );
```

Which of the following is the result of executing the above line?

- A. P29D
- B. P-29D
- C. P1M
- D. P-1M

7. Given:

```
System.out.println( Duration.between(  
    LocalDateTime.of(2015, 3, 20, 18, 0), LocalTime.of(18, 5) ) );
```

Which of the following is the result of executing the above line?

- A. PT5M
- B. PT-5M
- C. PT300S
- D. An exception is thrown

8. Which of the following are valid ChronoField values for LocalDate?

- A. DAY_OF_WEEK
- B. HOUR_OF_DAY
- C. DAY_OF_MONTH
- D. MILLI_OF_SECOND

ANSWERS

1. The correct answer is D.

Option A is not valid. The method `of()` takes the year, month and day as arguments.

Option B is not valid. The method `with()` is not a static method.

Option C is not valid. Valid month values are from 1 to 12.

Option D is valid. The methods `now()` and `plusDays()` are valid and can be chained.

2. The correct answer is C.

The method `atTime()` of `LocalDate` returns a `LocalDateTime` object with the combined values of the `LocalDate` instance which calls the method and the time parameters passed.

3. The correct answers are B and D.

`LocalTime` doesn't store information about years and (complete) days.

4. The correct answers are B and C.

Option A is incorrect. `Period` implements `TemporalAmount`.

Option B is correct. `java.time.Instant` implements `java.time.temporal.Temporal`.

Option C is correct. `LocalDate` and `LocalTime` are immutable. Therefore, they are thread-safe.

Option D is incorrect. `LocalDateTime` doesn't store zone time information, it just returns the current date-time of the system.

5. The correct answer is A.

The valid ways to get the nanoseconds part of an Instant object are:

```
int nanos = i.getNano();
```

Or:

```
int nanos = now.get(ChronoField.NANO_OF_SECOND);
```

6. The correct answer is D.

Since the first argument is greater than the second, the result is a negative period. A Period counts complete months first, then days.

7. The correct answer is D.

If the parameters types are different, the second parameter is converted to the type of the first parameter. In this case, a LocalTime cannot be converted to a LocalDateTime (it misses the year, month, day part) and an exception is thrown.

8. The correct answers are A and C.

A LocalDate stores the year, month, days and related information. It doesn't store hours or milliseconds.

Chapter TWENTY-ONE

Chapter TWENTY-TWO

Time Zones and Formatting

Exam Objectives

- Work with dates and times across time zones and manage changes resulting from daylight savings including Format date and times values.

CORE TIME ZONE CLASSES

Before Java 8, if we wanted to work with time zone information, we have to use the class `java.util.TimeZone`. Now with the new Date/Time API, there're new and better options.

They are:

ZonedDateTime

Represents the ID of time zone. For example, *Asia/Tokyo*.

ZoneOffset

Represents a time zone offset. It's a subclass of `ZonId`. For example, `-06:00`.

ZonedDateTime

Represents a date/time with time zone information. For example, `2015-08-30T20:05:12.463-05:00[America/Mexico_City]`.

OffsetDateTime

Represents a date/time with an offset from UTC/Greenwich. For example, `2015-08-30T20:05:12.463-05:00`.

OffsetTime

Represents a time with an offset from UTC/Greenwich. For example, `20:05:12.463-05:00`.

Just like the classes of the previous chapter, these one are located in the `java.time` package and are immutable.

ZONEID AND ZONEOFFSET CLASSES

The world is divided into time zones in which the same standard time is kept. By convention, a time zone is expressed as the number of hours different from the Coordinated Universal Time (*UTC*). Since the Greenwich Mean Time (*GMT*) and the Zulu time (*Z*), used in military, have no offset from *UTC*, they're often used as synonyms.

Java uses the Internet Assigned Numbers Authority (IANA) database of time zones, which keeps a record of all known time zones around the world and is updated many times per year.

Each time zone has an ID, represented by the class `java.time.ZoneId`. There are three types of ID:

The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with + or -, for example, `+02:00`.

The second type also states the offset from UTC/GMT time, but with one of the following prefixes: *UTC*, *GMT* and *UT*, for example, `UTC+11:00`. They are also represented by the class `ZoneOffset`.

The third type is region based. These IDs have the format *area/city*, for example, *Europe/London*.

You can get all the available zone IDs with the static method:

```
Set<String> getAvailableZoneIds()
```

For example, to print them in the console:

```
ZoneId.getAvailableZoneIds().stream().forEach(System.out::println);
```

To get the zone ID of your system use the static method:

```
ZoneId.systemDefault()
```

Under the cover, it uses `java.util.TimeZone.getDefault()` to find the default time zone and converts it to a ZoneId.

If you want to create a specific ZoneId object use the method `of()`:

```
ZoneId singaporeZoneId = ZoneId.of("Asia/Singapore");
```

This method parses the ID producing a ZoneOffset or a ZoneRegion (both extend from ZoneId).

Actually, the above line produces a ZoneRegion. A ZoneOffset is returned if for example, ID is Z, or starts with + or -. For example:

```
ZoneId zoneId = ZoneId.of("Z"); // Z represents the zone ID for UTC
ZoneId zoneId = ZoneId.of("-2"); // -02:00
```

The rules for this method are:

- If the zone ID equals Z, the result is ZoneOffset.UTC. Any other letter will throw an exception.
- If the zone ID starts with + or -, the ID is parsed as a ZoneOffset using ZoneOffset.of(String).
- If the zone ID equals GMT, UTC or UT then the result is a ZoneId with the same ID and rules equivalent to ZoneOffset.UTC.
- If the zone ID starts with UTC+, UTC-, GMT+, GMT-, UT+ or UT- then the ID is split in two, with a two or three letter prefix and a suffix starting

Chapter TWENTY-TWO

- with the sign. The suffix is parsed as a ZoneOffset. The result will be a ZoneId with the specified prefix and the normalized offset ID.
- All other IDs are parsed as region-based zone IDs. If the format is invalid (it has to match the expression [A-Za-z][A-Za-z0-9~/._+-]+) or is not found, an exception is thrown.

Remember that a ZoneOffset represents an offset, generally from UTC. This class has a lot more constructors than ZoneId:

```
// The offset must be in the range of -18 to +18
ZoneOffset offsetHours = ZoneOffset.ofHours(1);
// The range is -18 to +18 for hours and 0 to ± 59 for minutes
// If the hours are negative, the minutes must be negative or zero
ZoneOffset offsetHrMin = ZoneOffset.ofHoursMinutes(1, 30);
// The range is -18 to +18 for hours and 0 to ± 59 for mins and secs
// If the hours are negative, mins and secs must be negative or zero
ZoneOffset offsetHrMinSe = ZoneOffset.ofHoursMinutesSeconds(1,30,0);
// The offset must be in the range -18:00 to +18:00
// Which corresponds to -64800 to +64800
ZoneOffset offsetTotalSeconds = ZoneOffset.ofTotalSeconds(3600);
// The range must be from +18:00 to -18:00
ZoneOffset offset = ZoneOffset.of("+01:30:00");
```

The formats accepted by the of() method are:

- Z (for UTC)
- +h
- +hh
- +hh:mm
- -hh:mm
- +hhmm
- -hhmm
- +hh:mm:ss
- -hh:mm:ss

- +hhmmss
- -hhmmss

If you pass an invalid format or an out-of-range value to any of these methods, an exception is thrown.

To get the value of the offset, you can use:

```
// Gets the offset as int  
int offsetInt = offset.get(ChronoField.OFFSET_SECONDS);  
// Gets the offset as long  
long offsetLong= offset.getLong(ChronoField.OFFSET_SECONDS);  
// Gets the offset in seconds  
int offsetSeconds = offset.getTotalSeconds();
```

ChronoField.OFFSET_SECONDS is the only accepted value of ChronoField, so the three statements above return the same result. Other values throw an exception.

Anyway, once you have a ZoneId object, you can use it to create a ZonedDateTime instance.

A ZONEDDATETIME OBJECT

DATE

2015-08-31 T08:45:20.000

TIME

OFFSET

+02:00[Africa/Cairo]

TIME ZONE

ZONEDDATETIME CLASS

A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.

A `ZonedDateTime` object has three parts:

- A date
- A time
- A time zone

Which means that it stores all date and time fields, to a precision of nanoseconds, and a time zone with a zone offset.

So once you have a `ZoneId` object, you can combine it with a `LocalDate`, a `LocalDateTime`, or an `Instant`, to transform it into `ZonedDateTime`:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

LocalDate date = LocalDate.of(2010, 7, 03);
ZonedDateTime zonedDateTime = date.atStartOfDay(australiaZone);

LocalDateTime dateTime = LocalDateTime.of(2010, 7, 3, 9, 0);
ZonedDateTime zonedDateTime = dateTime.atZone(australiaZone);

Instant instant = Instant.now();
ZonedDateTime zonedDateTime = instant.atZone(australiaZone);
```

Or using the `of` method:

```
ZonedDateTime zonedDateTime2 = ZonedDateTime.of(
```

```
LocalDate.now(), LocalTime.now(), australiaZone);  
ZonedDateTime zonedDateTime3 = ZonedDateTime.of(  
    LocalDate.now(), australiaZone);  
ZonedDateTime zonedDateTime4 = ZonedDateTime.ofInstant(  
    Instant.now(), australiaZone);  
// year, month, day, hours, minutes, seconds, nanoseconds, zoneId  
ZonedDateTime zonedDateTime5 = ZonedDateTime.of(  
    2015, 1, 30, 13, 59, 59, 999, australiaZone);
```

You can also get the current date/time from the system clock in the default time zone with:

```
ZonedDateTime now = ZonedDateTime.now();
```

From a ZonedDateTime you can get LocalDate, LocalTime, or a LocalDateTime (without the time zone part) with:

```
LocalDate currentDate = now.toLocalDate();  
LocalTime currentTime = now.toLocalTime();  
LocalDateTime currentDateTime = now.toLocalDateTime();
```

ZonedDateTime also have most of the methods of LocalDateTime that we reviewed in the previous chapter:

```
//To get the value of a specified field  
int day = now.getDayOfMonth();  
int dayYear = now.getDayOfYear();  
int nanos = now.getNano();  
Month monthEnum = now.getMonth();  
int year = now.get(ChronoField.YEAR);  
long micro = now.getLong(ChronoField.MICRO_OF_DAY);  
// This is new, gets the zone offset such as "-03:00"  
ZoneOffset offset = now.getOffset();
```

Chapter TWENTY-TWO

```
// To create another instance
ZonedDateTime zdt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
ZonedDateTime zdt2 = now.withSecond(49);
// Since these methods return a new instance, we can chain them!
ZonedDateTime zdt3 = now.withYear(2013).withMonth(12);

// The following two methods are specific to ZonedDateTime
// Returns a copy of the date/time with a
// different zone, retaining the instant
ZonedDateTime zdt4 = now.withZoneSameInstant(australiaZone);
// Returns a copy of this date/time with a different time zone,
// retaining the local date/time if it's valid for the new time zone
ZonedDateTime zdt5 = now.withZoneSameLocal(australiaZone);

// Adding
ZonedDateTime zdt6 = now.plusDays(4);
ZonedDateTime zdt7 = now.plusWeeks(3);
ZonedDateTime zdt8 = now.plus(2, ChronoUnit.HOURS);

// Subtracting
ZonedDateTime zdt9 = now.minusMinutes(20);
ZonedDateTime zdt10 = now.minusNanos(99999);
ZonedDateTime zdt11 = now.minus(10, ChronoUnit.SECONDS);
```

The method `toString()` returns the date/time in the format of a `LocalDateTime` followed by a `ZoneOffset`, optionally, a `ZoneId` if it is not the same as the offset, and omitting the parts with value zero:

```
// Prints 2014-09-19T00:30Z
System.out.println(
    ZonedDateTime.of(2014, 9, 19, 0, 30, 0, 0, ZoneId.of("Z")));
// Prints 2015-08-31T12:39:27.492-04:00[America/Montreal]
System.out.println(ZonedDateTime.now(ZoneId.of("America/Montreal")));
```

DAYLIGHT SAVINGS

Many countries in the world adopt what is called Daylight Saving Time (DST), the practice of advance the clock by an hour in the summer (well, not exactly in the summer in all countries but let's bear with this) when the daylight savings time starts.

When the daylight time ends, clocks are set back by an hour. This is done to make better use of natural daylight.

`ZonedDateTime` is fully aware of DST.

For an example, let's take a country where DST is fully observed, like Italy (UTC/GMT +2).

In 2015, DST started in Italy on March, 29th and ended on October, 25th. This means that on:

*March, 29 2015 at 2:00:00 A.M. clocks were turned **forward** 1 hour to
March, 29 2015 at 3:00:00 A.M. local daylight time instead
(So a time like March, 29 2015 2:30:00 A.M. didn't actually exist!)*

*October, 25 2015 at 3:00:00 A.M. clocks were turned **backward** 1 hour to
October, 25 2015 at 2:00:00 A.M. local daylight time instead
(So a time like October, 25 2015 2:30:00 A.M. actually existed twice!)*

If we create an instance of `LocalDateTime` with this date/time and print it:

Chapter TWENTY-TWO

```
LocalDateTime ldt = LocalDateTime.of(2015, 3, 29, 2, 30);
System.out.println(ldt);
```

The result will be:

2015-03-29T02:30

But if we create an instance of ZonedDateTime for Italy (notice that the format uses a city, not a country) and printed:

```
ZonedDateTime zdt = ZonedDateTime.of(2015, 3, 29, 2, 30, 0, 0,
                                         ZoneId.of("Europe/Rome"));
System.out.println(zdt);
```

The result will be just like in the real world when using DST:

2015-03-29T03:30+02:00[Europe/Rome]

But be careful. We have to use a regional ZoneId, a ZoneOffset won't do the trick because this class doesn't have the zone rules information to account for DST:

```
ZonedDateTime zdt1 = ZonedDateTime.of(2015, 3, 29, 2, 30, 0, 0,
                                         ZoneOffset.ofHours(2));
ZonedDateTime zdt2 = ZonedDateTime.of(2015, 3, 29, 2, 30, 0, 0,
                                         ZoneId.of("UTC+2"));
System.out.println(zdt1);
System.out.println(zdt2);
```

The result will be:

2015-03-29T02:30+02:00[UTC+02:00]

2015-03-29T02:30+02:00

When DST ends, something similar happens

```
LocalDateTime ldt = LocalDateTime.of(2015, 10, 25, 3, 30);
System.out.println(ldt);
```

The result will be:

2015-10-25T02:30

When we create an instance of ZonedDateTime for Italy, we have to add an hour to see the effect (otherwise we'll be creating the ZonedDateTime at 3:00 of the new time):

```
ZonedDateTime zdt = ZonedDateTime.of(2015, 10, 25, 2, 30, 0, 0,
                                         ZoneId.of("Europe/Rome"));
ZonedDateTime zdt2 = zdt.plusHours(1);
System.out.println(zdt);
System.out.println(zdt2);
```

The result will be:

2015-10-25T02:30+02:00[Europe/Rome]
2015-10-25T02:30+01:00[Europe/Rome]

We also need to be careful when adjusting the time across the DST boundary with a version of the methods `plus()` and `minus()` that takes a `TemporalAmount` implementation, in other words, a `Period` or a `Duration`. This is because both differ in their treatment of daylight savings time.

Consider one hour before the beginning of DST in Italy:

Chapter TWENTY-TWO

```
ZonedDateTime zdt = ZonedDateTime.of(2015, 3, 29, 1, 0, 0, 0,  
ZoneId.of("Europe/Rome"));
```

When we add a Duration of one day:

```
System.out.println(zdt.plus(Duration.ofDays(1)));
```

The result is:

```
2015-03-30T02:00+02:00[Europe/Rome]
```

When we add a Period of one day:

```
System.out.println(zdt.plus(Period.ofDays(1)));
```

The result is:

```
2015-03-30T01:00+02:00[Europe/Rome]
```

The reason is that Period adds a *conceptual* date, while Duration adds *exactly* one day (24 hours or 86,400 seconds) and when it crosses the DST boundary, one hour is added, and the final time is 02:00 instead of 01:00.

OFFSETDATETIME AND OFFSETTIME

OffsetDateTime represents an object with date/time information and an offset from UTC, for example, 2015-01-01T11:30-06:00.

You may think Instant, OffsetDateTime and ZonedDateTime are very much alike, after all, they all store the date and time to a nanosecond precision. However, there are subtle but important differences:

- Instant represents a point in time in the UTC time zone.
- OffsetDateTime represents a point in time with an offset (any offset).
- ZonedDateTime represents a point in time in a time zone (any time zone), adding full time zone rules like daylight saving time adjustments.

OffsetTime represents a time with an offset from UTC, for example, 11:30-06:00. The common way to create an instance of these classes is:

```
OffsetDateTime odt = OffsetDateTime.of(LocalDateTime.now(),
                                         ZoneOffset.of("+03:00"));
OffsetTime ot = OffsetTime.of(LocalTime.now(),
                             ZoneOffset.of("-08:00"));
```

Both classes have practically the same methods that their LocalDateTime, ZonedDateTime, and LocalTime counterparts. With an offset from UTC and without time zone variations, they always represent an exact instant in time, which may be more suited for certain types of applications (the Java documentation recommends OffsetDateTime when communicating with a database or in a network protocol).

PARSING AND FORMATTING

`java.time.format.DateTimeFormatter` is the new class for parsing and formatting dates. It can be used in two ways:

- The date/time classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetDate` and `OffsetDateTime` all have the following three methods:

```
// Formats the date/time object using the specified formatter
String format(DateTimeFormatter formatter)
// Obtains an instance of a date/time object (of type T)
// from a string with a default format
static T parse(CharSequence text)
// Obtains an instance of a date/time object (of type T)
// from a string using a specific formatter

static T parse(CharSequence text, DateTimeFormatter formatter)
```

- `DateTimeFormatter` has the following two methods:

```
// Formats a date/time object using the formatter instance
String format(TemporalAccessor temporal)
// Parses the text producing a temporal object
TemporalAccessor parse(CharSequence text)
```

All format methods throw the runtime exception:

`java.time.DateTimeException`

All parse methods throw the runtime exception:

`java.time.format.DateTimeParseException`

`DateTimeFormatter` provides three ways to format date/time objects:

- Predefined formatters
- Locale-specific formatters
- Formatters with custom patterns

Formatter	Description	Example
BASIC_ISO_DATE	Date fields without separators	20150803
ISO_LOCAL_DATE	Date fields with separators	2015-08-03
ISO_LOCAL_TIME		13:40:10
ISO_LOCAL_DATE_TIME		2015-08-03T13:40:10
ISO_OFFSET_DATE	Date fields with separators and zone offset	2015-08-03+07:00
ISO_OFFSET_TIME		13:40:10+07:00
ISO_OFFSET_DATE_TIME		2015-08-03 T13:40:10+07:00
ISO_ZONED_DATE_TIME	A zoned date and time	2015-08-03 T13:40:10+07:00 [Asia/Bangkok]
ISO_DATE	Date or Time with or without offset	2015-08-03+07:00
ISO_TIME		13:40:10
ISO_DATE_TIME	DateTime with Zoneld	2015-08-03 T13:40:10+07:00 [Asia/Bangkok]
ISO_INSTANT	Date and Time of an Instant	2015-08-03 T13:40:10Z
ISO_ORDINAL_DATE	Year and day of the year	2015-200
ISO_WEEK_DATE	Year, week and day of the week	2015-W34-2
RFC_1123_DATE_TIME	RFC 1123 / RFC 822 date format	Mon, 3 Ago 2015 13:40:10 GMT

Predefined Formatters

Style	Date	Time
SHORT	8/3/15	1:40 PM
MEDIUM	Aug 03, 2015	1:40:00 PM
LONG	August 03, 2015	1:40:00 PM PDT
FULL	Monday, August 03, 2015	1:40:00 PM PDT

Locale-specific Styles

Chapter TWENTY-TWO

Symbol	Meaning	Examples
G	Era	AD; Anno Domini; A
u	Year	2015; 15
y	Year of Era	2015; 15
D	Day of Year	150
M / L	Month of Year	7; 07; Jul; July; J
d	Day of Month	20
Q / q	Quarter of year	2; 02; Q2; 2nd quarter
Y	Week-based Year	2015; 15
w	Week of Week-based Year	30
W	Week of Month	2
E	Day of Week	Tue; Tuesday; T
e / c	Localized Day of Week	2; 02; Tue; Tuesday; T
F	Week of Month	2
a	AM/PM of Day	AM
h	Hour (1-12)	10
K	Hour (0-11)	1
k	Hour (1-24)	20
H	Hour (0-23)	23
m	Minute	10
s	Second	11
S	Fraction of Second	999
A	Milli of Day	2345
n	Nano of Second	865437987
N	Nano of Day	12986497300
V	Time Zone ID	Asia/Manila; Z; -06:00
z	Time Zone Name	Pacific Standard Time; PST
O	Localized Zone Offset	GMT+4; GMT+04:00; UTC-04:00;
X	Zone Offset ('Z' for zero)	Z; -08; -0830; -08:30
x	Zone Offset	+0000; -08; -0830; -08:30
Z	Zone Offset	+0000; -0800; -08:00
'	Escape for Text	
''	Single Quote	
[]	Optional Section Start / End	
# { }	Reserved for future use	

Assuming:

```
LocalDate ldt = LocalDate.of(2015, 1, 20);
```

These are examples of using a predefined formatter:

```
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));  
System.out.println(ldt.format(DateTimeFormatter.ISO_DATE));
```

The output will be:

```
2015-01-20  
2015-01-20
```

These are examples of using a localized style:

```
DateTimeFormatter formatter =  
    DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);  
// With the current locale  
System.out.println(formatter.format(ldt));  
System.out.println(ldt.format(formatter));  
// With another locale  
System.out.println(  
    formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
20/01/15  
20/01/15  
20.01.15
```

Chapter TWENTY-TWO

And these are examples of using a custom pattern:

```
DateTimeFormatter formatter =  
    DateTimeFormatter.ofPattern("QQQQ Y");  
// With the current locale  
System.out.println(formatter.format(ldt));  
System.out.println(ldt.format(formatter));  
// With another locale  
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
1st quarter 2015  
1st quarter 2015  
1. Quartal 2015
```

If the formatter uses information that is not available, a `DateTimeException` will be thrown. For example, using a `DateTimeFormatter.ISO_OFFSET_DATE` with a `LocalDate` instance (it doesn't have offset information).

To parse a date and/or time value from a string, use one of the parse methods. For example:

```
// Format according to ISO-8601  
String dateTimeStr1 = "2015-06-29T14:45:30";  
// Custom format  
String dateTimeStr2 = "2015/06/29 14:45:30";
```

```

LocalDateTime ldt = LocalDateTime.parse(dateTimeStr1);
// Using DateTimeFormatter
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
// DateTimeFormatter returns a TemporalAccessor instance
TemporalAccessor ta = formatter.parse(dateTimeStr2);
// LocalDateTime returns an instance of the same type
ldt = LocalDateTime.parse(dateTimeStr2, formatter);

```

The version of `parse()` of the date/time objects takes a string in a format according to ISO-8601, this is:

Class	Format	Example
LocalDate	<i>uuuu-MM-dd</i>	2007-12-03
LocalTime	<i>HH:mm:ss</i>	10:15
LocalDateTime	<i>uuuu-MM-dd'T'HH:mm:ss</i>	2007-12-03T10:15:30
ZonedDateTime	<i>uuuu-MM-dd'T'HH:mm:ss XXXXX[VV]</i>	2011-12-03T10:15:30 +01:00[Europe/Paris]
OffsetDateTime	<i>uuuu-MM- dd'T'HH:mm:ssXXXXX</i>	2011-12-03T10:15:30 +01:00
OffsetTime	<i>HH:mm:ssXXXXX</i>	10:15:30+01:00

If the formatter uses information that is not available or if the pattern of the format is invalid, a `DateTimeParseException` will be thrown.

KEY POINTS

- `ZoneId`, `ZoneOffset`, `ZonedDateTime`, `OffsetDateTime`, and `OffsetTime` are the classes of the new Java Date/Time API that store information about time zones and time offsets. They are located in the `java.time` package and are immutable.
- Each time zone has an ID, represented by the class `ZoneId`. There are three types of ID.
- The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with + or -, for example `+02:00`.
- The second type also states the offset from UTC/GMT time, but with one of the following prefixes: UTC, GMT and UT, for example, `UTC+11:00`. They are also represented by the class `ZoneOffset`.
- The third type is region based. These IDs have the format *area/city*, for example, `Europe/London`.
- If you want to create a specific `ZoneId` object use the method of:

```
ZoneId.of("Asia/Singapore");  
ZoneId.of("+3")
```

- The first two of the above methods produce an object of type `ZoneRegion`. The last one, an object of type `ZoneOffset`.
- A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.
- A `ZonedDateTime` object has three parts, a date, a time, and a time zone. It can be created with either:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");
```

```
ZonedDateTime zonedDateTime5 = ZonedDateTime.of(  
    2015, 1, 30, 13, 59, 59, 999, australiaZone);
```

- Or through a LocalDate, LocalTime, LocalDateTime or Instant plus a ZoneId.
- If we create an instance of ZonedDateTime for a region where a Daylight Saving Time (DST) is observed, the instance will support it, advancing the clock one hour when DST starts, and setting it back when DST ends.
- Period and Duration differ in their treatment of DST.
- Period adds a *conceptual* day to a date, while Duration adds an *exact* day to a date, without taking into account DST.
- OffsetDateTime represents an object with date/time information and an offset from UTC, for example, `2015-01-01T11:30-06:00`.
- OffsetTime represents a time with an offset from UTC, for example, `11:30-06:00`.
- `java.time.format.DateTimeFormatter` is the new class for parsing and formatting dates. It can be used in two ways:
- The date/time classes LocalDate, LocalTime, LocalDateTime, ZonedDateTime, OffsetDate and OffsetDateTime all have the methods:

```
String format(DateTimeFormatter formatter)  
static T parse(CharSequence text)  
static T parse(CharSequence text, DateTimeFormatter formatter)
```

- `DateTimeFormatter` has the following two methods:


```
String format(TemporalAccessor temporal)  
TemporalAccessor parse(CharSequence text)
```
- All format methods throw the runtime exception `java.time.DateTimeException`, while all parse methods throw the runtime exception `java.time.format.DateTimeParseException`.

SELF TEST

1. Which of the following are valid ways to create a ZoneId object?

- A. ZoneId.ofHours(2);
- B. ZoneId.of("2");
- C. ZoneId.of("-1");
- D. ZoneId.of("America/Canada");

2. Given:

```
ZoneOffset offset = ZoneOffset.of("Z");
System.out.println(offset.get(ChronoField.HOUR_OF_DAY));
```

Which of the following is the result of executing the above lines?

- A. 0
- B. 1
- C. 12:00
- D. An exception is thrown

3. Given:

```
ZonedDateTime zdt =
    ZonedDateTime.of(2015, 02, 28, 5, 0, 0, 0, ZoneId.of("+05:00"));
System.out.println(zdt.toLocalTime());
```

Assuming a local time zone of +2:00, which of the following is the result of executing the above lines?

- A. 05:00
- B. 17:00
- C. 02:00
- D. 03:00

4. Given:

```
ZonedDateTime zdt =  
    ZonedDateTime.of(2015, 10, 4, 0, 0, 0, 0, ZoneId.of("America/Asuncion"))  
    .plus(Duration.ofHours(1));  
System.out.println(zdt);
```

Assuming that DST starts on October, 4, 2015 at 0:00:00, which of the following is the result of executing the above lines?

- A. 2015-10-04T00:00-03:00[America/Asuncion]
- B. 2015-10-04T01:00-03:00[America/Asuncion]
- C. 2015-10-04T02:00-03:00[America/Asuncion]
- D. 2015-10-03T23:00-03:00[America/Asuncion]

5. Given:

```
ZonedDateTime zdt =  
    ZonedDateTime.of(2015, 3, 22, 0, 0, 0, 0, ZoneId.of("America/Asuncion"))  
    .minus(Period.ofDays(1));  
System.out.println(zdt);
```

Assuming that DST ends on March, 22, 2015 at 0:00:00, which of the following is the result of executing the above lines?

- A. 2015-03-21T01:00-03:00[America/Asuncion]
- B. 2015-03-21T00:00-03:00[America/Asuncion]
- C. 2015-03-20T23:00-03:00[America/Asuncion]
- D. 2015-03-21T02:00-03:00[America/Asuncion]

6. Which of the following statements are true?

- A. java.time.ZoneOffset is a subclass of java.time.ZoneId.
- B. java.time.Instant can be obtained from java.time.ZonedDateTime.
- C. java.time.ZoneOffset can manage DST.
- D. java.time.OffsetDateTime represents a point in time in the UTC time zone.

Chapter TWENTY-TWO

7. Given:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(
    formatter
    .withLocale(Locale.ENGLISH)
    .format(LocalDateTime.of(2015, 5, 7, 16, 0))
);
```

Which of the following is the result of executing the above lines?

- A. 5/7/15 4:00 PM
- B. 5/7/15
- C. 4:00 PM
- D. 4:00:00 PM

8. Given:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("HH:mm:ss X");
OffsetDateTime odt =
    OffsetDateTime.parse("11:50:20 Z", formatter);
```

Which of the following statements is true about the above lines?

- A. The pattern *HH:mm:ss X* is invalid.
- B. An `OffsetDateTime` is created successfully.
- C. `Z` is an invalid offset.
- D. An exception is thrown at runtime.

ANSWERS

1. The correct answer is C.

Option A is invalid. Method `ofHours(int)` belongs to `ZoneOffset`, not to `ZoneId`.

Option B is invalid. The format of the offset is incorrect. It has to start with a sign (+ or -).

Option C is valid for the above reason.

Option D is invalid. The format for zone regions is *area/city* not *area/country*. A valid example would be *America/Montreal*.

2. The correct answer is D.

The instantiation of `ZoneOffset` is valid, Z correspond to *UTC*, but `ChronoField.OFFSET_SECONDS` is the only accepted value for the method `get`, so a runtime exception is thrown.

3. The correct answer is A.

The local time zone has no effect here. From a `ZonedDateTime`, you can get a `LocalDate`, a `LocalTime`, or a `LocalDateTime` just without the time zone part.

4. The correct answer is C.

On October, 4, 2015 at 0:00:00, the clock turned forward 1 hour. A `ZonedDateTime` is created at that time and added one hour, setting it at 1:00, but since the clock is already forwarded, that time becomes 2:00.

5. The correct answer is B.

Period subtracts one conceptual day, making up for any daylight saving variation and leaving the same time. However,

```
ZonedDateTime zdt =  
    ZonedDateTime.of(2015, 3, 22, 0, 0, 0,  
                      ZoneId.of("America/Asuncion"))  
    .minus(Duration.ofDays(1));  
System.out.println(zdt);
```

It's different. The result will be:

```
2015-03-21T01:00-03:00[America/Asuncion]
```

Because 0:00 it's actually 1:00 (when DST ended at 0:00, the clock was set at 23:00 of the previous day) and Duration subtracts exactly 24 hours.

6. The correct answers are A and B.

Option A is true. `java.time.ZoneOffset` is a subclass of `java.time.ZoneId`.

Option B is true. A `java.time.Instant` instance can be obtained from `java.time.ZonedDateTime`.

Option C is false. `java.time.ZoneOffset` cannot manage DST.

Option D is false. `java.time.Instant` is the one that represents a point in time in the UTC time zone, `java.time.OffsetDateTime` represents a point in time from UTC.

7. The correct answer is C.

A `DateTimeFormatter` for time is created with the style `FormatStyle.SHORT`:

```
DateTimeFormatter formatter =  
    DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
```

So only the time part of the `LocalDateTime` is formatted. Option D is the result of applying `FormatStyle.MEDIUM`.

8. The correct answer is D.

Option A is false. The pattern is valid:

- *HH* represents hours (0-23)
- *mm* represents minutes
- *ss* represents seconds
- *X* represents a zone offset (with support for Z)

Option B is false. An `OffsetDateTime` is not created because the string to parse is missing the date part, so an exception is thrown.

Option C is false. Z represents a zero offset (Zulu time, the same as UTC).

Part SEVEN

Java I/O

Chapter TWENTY-THREE

Java I/O Fundamentals

Exam Objectives

- Read and write data from the console.
- Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.

I/O STREAMS

Java I/O (Input/Output or Reading/Writing) is a large topic, but let's start by defining what is a stream in I/O.

Although conceptually, they're kind of similar, I/O streams are not related in any way with the Stream API. Therefore, all references to streams in this chapter refer to I/O stream.

In simple words, a stream is a sequence of data.

In the context of this chapter, that sequence of data is the content of a file. Take for example this sequence of bytes:

...10101010001011101001010100000111101010101101000...

When we read that sequence of bytes from a file, we are reading an *input stream*.

When we write that sequence of bytes to a file, we are writing an *output stream*.

Moreover, the content of a file can be so large that it might not fit into memory, so when working with streams, we can't focus on the entire stream at once, but only in small portions (either byte by byte or a group of bytes) at any time.

But Java not only supports streams of bytes.

In the `java.io` package, we can find classes to work with byte and character streams as well.

FILES

Let's think of a *file* as a resource that stores data (either in byte or character format).

Files are organized in *directories*, which in addition to files, can contain other directories as well.

Files and directories are managed by a *file system*.

Different operating systems can use different file systems, but the files and directories are organized in a hierarchical way. For example, in a Unix-based file system, that would be:

```
/  
|- home  
|   |- documents  
|   |   |- file.txt  
|   |- music  
|   |- user.properties
```

Where a file or a directory can be represented by a String called *path*, where the value to the left of a separator character (which changes between file systems) is the parent of the value to the right of the separator, like `/home/documents/file.txt`, or `c:\home\documents\file.txt` on Windows.

In Java, the `java.io.File` class represents either a file or a directory path of a file system (there isn't a `Directory` class, at least in the standard Java I/O API, for directories):

```
File file = new File("/home/user.properties");
```

When you create an instance of the `File` class, you're not creating a new file, you are just creating an object that may represent an actual file or directory (it may not even exist yet).

But once you have a `File` object, you can use some methods to work with that file/directory. For example:

```
String path = ...
File file = new File(path);
if(file.exists()) {
    String name = file.getName(); // Name of the file/directory
    String parent = file.getParent(); // Path of its parent
    // Returning the time the file/directory was modified
    // in milliseconds since 00:00:00 GMT, January 1, 1970
    long millis = file.lastModified();
    // If the object represents a file
    if(file.isFile()) {
        // Returning the size of the file in bytes
        long size = file.length();
    }
    // If the object represents a directory
    else if(file.isDirectory()) {
        // Returns true only if the directory was created
        boolean dirCreated = file.mkdir();
        // Returns true only if the directory was created,
        // along with all necessary parent directories
        boolean dirsCreated = file.mkdirs();
        // Get all the files/directories in a directory
        String[] fileNames = file.list(); // Just the names
        File[] files = file.listFiles(); // As File instances
    }
    boolean wasRenamed = file.renameTo(new File("new file"));
    boolean wasDeleted = file.delete();
}
```

UNDERSTANDING THE JAVA.IO PACKAGE

Now we have reviewed the basics, we can dissect the Java I/O API.

There are a lot of classes in the `java.io` package, but in this chapter, we'll only review the ones covered by the exam.

We can start by listing four **ABSTRACT CLASSES** that are the parents of all the other classes.

The first two deal with byte streams:

- `InputStream`
- `OutputStream`

The other two deal with character streams:

- `Reader`
- `Writer`

So we can say that all the classes that have **Stream** in their name read or write streams of **BYTES**.

And all the classes that have **Reader** or **Writer** in their name read or write streams of **CHARACTERS**.

You shouldn't have any problem in knowing that classes that have **Input** or **Reader** in their names are for **READING** (either bytes or characters).

That all the classes that have **Output** and **Writer** in their names are for **WRITING** (either bytes or characters).

And that every class that **READS** something (or do it in a certain way), has a corresponding class that **WRITES** that something (or do it in that certain way), like `FileReader` and `FileWriter`.

However, this last rule has exceptions. The ones you should know about are `PrintStream` and `PrintWriter` (looking at the name I think it's obvious that these classes are just for output, but at least, you can tell which works with bytes and which with characters).

Next, we have classes that have **Buffered** in their name, which use a buffer to read or write data in groups (of bytes or characters) to do it more efficiently.

Finally, based on the idea that we can use some classes in **COMBINATION**, we can further classify the classes in the API as wrappers and non-wrappers.

Non-wrapper classes generally take an instance of `File` or a `String` to create an instance, while wrapper classes take another stream class to create an instance. The following is an example of a wrapper class:

```
ObjectInputStream ois =  
    new ObjectInputStream(new FileInputStream("obj.dat"));
```

When combining classes this way, in almost all cases, it's not valid mix **OPPOSITE** concepts, like combining a Reader with a Writer or a Reader with an `InputStream`:

```
BufferedReader br =  
    new BufferedReader(new FileInputStream("file.txt")); //error
```

This classification isn't that evident by looking at the name of the classes, but if you know what each of the classes do and think about it, you'll know if they can wrap other `java.io` classes or not.

FILEINPUTSTREAM

`FileInputStream` reads bytes from a file. It inherits from `InputStream`.

It can be created either with a `File` object or a `String` path:

```
FileInputStream(File file)  
FileInputStream(String path)
```

Here's how you use it:

```
try (InputStream in = new FileInputStream("c:\\file.txt")) {  
    int b;  
    while((b = in.read()) != -1) { // -1 indicates the end of the file  
        // Do something with the byte read  
    }  
} catch(IOException e) { /** ... */ }
```

There's also a `read()` method that reads bytes into an array of bytes:

```
byte[] data = new byte[1024];  
int number0fBytesRead;  
while((number0fBytesRead = in.read(data)) != -1) {  
    // Do something with the array data  
}
```

All the classes we'll review should be closed. Fortunately, they implement `AutoClosable` so they can be used in a `try-with-resources`.

Also, almost all methods of these classes throw `IOExceptions` or one of its subclasses (such a `FileNotFoundException`, which is pretty descriptive).

FILEOUTPUTSTREAM

FileOutputStream writes bytes to a file. It inherits from OutputStream.

It can be created either with a File object or a String path and an optional boolean that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(String path)
FileOutputStream(String path, boolean append)
```

Here's how you use it:

```
try (OutputStream out = new FileOutputStream("c:\\file.txt")) {
    int b;
    // Made up method to get some data
    while((b = getData()) != -1) {
        out.write(b); // Writes b to the file output stream
        out.flush();
    }
} catch(IOException e) { /* ... */ }
```

When you write to an OutputStream, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the OutputStream, you can call the flush() method every once in a while.

FileOutputStream also contains overloaded versions of write() that allow you to write data contained in a byte array.

FILEREADER

FileReader reads characters from a text file. It inherits from Reader.

It can be created either with a File object or a String path:

```
FileReader(File file)  
FileReader(String path)
```

Here's how you use it:

```
try (Reader r = new FileReader("/file.txt")) {  
    int c;  
    while((c = r.read()) != -1) { // -1 indicates the end of the file  
        char character = (char)c;  
        // Do something with the character  
    }  
} catch(IOException e) { /* ... */ }
```

There's also a read() method that reads characters into an array of chars:

```
char[] data = new char[1024];  
int number_of_chars_read = r.read(data);  
while((number_of_chars_read = r.read(data)) != -1) {  
    // Do something with the array data  
}
```

FileReader assumes that you want to decode the characters in the file using the default character encoding of the machine your program is running on.

FILEWRITER

FileWriter writes characters to a text file. It inherits from Writer.

It can be created either with a File object or a String path and an optional boolean that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(String path)
FileWriter(String path, boolean append)
```

Here's how you use it:

```
try (Writer w = new FileWriter("/file.txt")) {
    w.write('-'); // writing a character
    w.write("Writing to the file..."); // writing a string
} catch(IOException e) { /** ... */ }
```

Just like an OutputStream, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the FileWriter, you can call the flush() method every once in a while.

FileWriter also contains overloaded versions of write() that allow you to write data contained in a char array, or in a String.

FileWriter assumes that you want to encode the characters in the file using the default character encoding of the machine your program is running on.

BUFFEREDREADER

BufferedReader reads text from a character stream. Rather than read one character at a time, BufferedReader reads a large block at a time into a buffer. It inherits from Reader.

This is a wrapper class that is created by passing a Reader to its constructor, and optionally, the size of the buffer:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

BufferedReader has one extra read method (in addition to the ones inherited by Reader), readLine(). Here's how you use it:

```
try ( BufferedReader br =
      new BufferedReader( new FileReader("/file.txt") ) ) {
    String line;
    // null indicates the end of the file
    while((line = br.readLine()) != null) {
        // Do something with the line
    }
} catch(IOException e) { /** ... */ }
```

When the BufferedReader is closed, it will also close the Reader instance it reads from.

BUFFEREDWRITER

BufferedWriter writes text to a character stream, buffering characters for efficiency. It inherits from Writer.

This is a wrapper class that is created by passing a Writer to its constructor, and optionally, the size of the buffer:

```
BufferedWriter(Writer out)  
BufferedWriter(Writer out, int size)
```

BufferedWriter has one extra write method (in addition to the ones inherited by Writer), newLine(). Here's how you use it:

```
try ( BufferedWriter bw =  
      new BufferedWriter( new FileWriter("/file.txt") ) ) {  
    w.newLine("Writing to the file...");  
} catch(IOException e) { /* ... */ }
```

Since data is written to a buffer first, you can call the flush() method to make sure that the text written until that moment is indeed written to the disk.

When the BufferedWriter is closed, it will also close the Writer instance it writes to.

OBJECTINPUTSTREAM / OBJECTOUTPUTSTREAM

The process of converting an object to a data format that can be stored (in a file for example) is called *serialization* and converting that stored data format into an object is called *deserialization*.

If you want to serialize an object, its class must implement the `java.io.Serializable` interface, which has no methods to implement, it only tags the objects of that class as serializable.

If you try to serialize a class that doesn't implement that interface, a `java.io.NotSerializableException` (a subclass of `IOException`) will be thrown at runtime.

`ObjectOutputStream` allows you to serialize objects to an `OutputStream` while `ObjectInputStream` allows you to deserialize objects from an `InputStream`. So both are considered wrapper classes.

Here's the constructor of the `ObjectOutputStream` class:

```
ObjectOutputStream(OutputStream out)
```

This class has methods to write many primitive types, like:

```
void writeInt(int val)
void writeBoolean(boolean val)
```

But the most useful is `writeObject(Object)`. Here's an example:

```
class Box implements java.io.Serializable { /** ... */ }
...
```

```
try( ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("obj.dat") ) ) {
    Box box = new Box();
    oos.writeObject(box);
} catch(IOException e) { /** ... */ }
```

To deserialize the file obj.dat, we use `ObjectInputStream` class. Here's its constructor:

```
ObjectInputStream(InputStream in)
```

This class has methods to read many data types, among them:

```
Object readObject() throws IOException, ClassNotFoundException
```

Notice that it returns an `Object` type. Thus, we have to cast the object explicitly. This can lead to a `ClassCastException` thrown at runtime. Note that this method also throws a `ClassNotFoundException` (a checked exception), in case the class of a serialized object cannot be found.

Here's an example:

```
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("obj.dat") ) ) {
    Box box = null;
    Object obj = ois.readObject();
    if(obj instanceof Box) {
        box = (Box)obj;
    }
} catch(IOException ioe) { /** ... */ }
catch(ClassNotFoundException cnfe) { /** ... */ }
```

Two important notes. When deserializing an object, the constructor, and any initialization block are not executed. Second, null objects are not serialized/deserialized.

PRINTWRITER

`PrintWriter` is a subclass of `Writer` that writes formatted data to another (wrapped) stream, even an `OutputStream`. Just look at its constructors:

```
PrintWriter(File file)
    throws FileNotFoundException
PrintWriter(File file, String charset)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName)
    throws FileNotFoundException
PrintWriter(String fileName, String charset)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

By default, it uses the default charset of the machine you're running the program, but at least, this class accepts the following charsets (there are other optional charsets):

- US-ASCII
- ISO-8859-1
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-16

As any `Writer`, this class has the `write()` method we've seen in other `Writer` subclasses, but it overwrites them to avoid throwing an `IOException`.

It also adds the methods `format()`, `print()`, `printf()`, `println()`.

Here's how you use this class:

```
// Opens or creates the file without automatic line flushing
// and converting characters by using the default character encoding
try(PrintWriter pw = new PrintWriter("/file.txt")) {

    pw.write("Hi"); // Writing a String
    pw.write(100); // Writing a character

    // write the string representation of the argument
    // it has versions for all primitives, char[], String, and Object
    pw.print(true);
    pw.print(10);

    // same as print() but it also writes a line break as defined by
    // System.getProperty("line.separator") after the value
    pw.println(); // Just writes a new line
    pw.println("A new line...");

    // format() and printf() are the same methods
    // They write a formatted string using a format string,
    // its arguments and an optional Locale
    pw.format("%s %d", "Formatted string ", 1);
    pw.printf("%s %d", "Formatted string ", 2);
    pw.format(Locale.GERMAN, "%.2f", 3.1416);
    pw.printf(Locale.GERMAN, "%.3f", 3.1416);

} catch(FileNotFoundException e) { /** ... */ } // if the file cannot
                                                // be opened or created
```

You can learn more about format strings for `format()` and `printf()` in
<https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html>

STANDARD STREAMS

Java initializes and provides three stream objects as public static fields of the `java.lang.System` class:

- `InputStream System.in`
The standard input stream (typically the input from the keyboard)
- `PrintStream System.out`
The standard output stream (typically the default display output)
- `PrintStream System.err`
The standard error output stream (typically the default display output)

`PrintStream` does exactly the same and has the same features that `PrintWriter`, it just works with `OutputStreams` only.

The following example shows how to read a single character (a byte) from the command line:

```
System.out.print("Enter a character: ");
try {
    int c = System.in.read();
} catch(IOException e) {
    System.err.println("Error: " + e);
}
```

Or to read Strings:

```
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));
String line = br.readLine();
// Or using the java.util.Scanner class
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

JAVA.IO.CONSOLE

Since Java 6, we have the `java.io.Console` class to access the *console* of the machine your program is running on.

You can get a reference to this class (is a singleton) with `System.console()`.

But keep in mind that if your program is running in an environment that doesn't have access to a console (like an IDE or if your program is running as a background process), `System.console()` will return `null`.

With the `Console` object, you can easily read user input with the `readLine()` method and even read passwords with `readPassword()`.

For output, this class has the `format()` and `printf()` methods that work just like the ones of `PrintWriter`.

Finally, the methods `reader()` and `writer()` return an instance of `Reader` and `Writer` respectively:

```
Console console = System.console();
if(console != null) { // Check if the console is available
    console.writer().println("Enter your user and password");
    String user = console.readLine("Enter user: ");
    // readPassword() hides what the user is typing
    char[] pass = console.readPassword("Password: ");
    // Clear password from memory by overwriting it
    Arrays.fill(pass, 'x');
}
```

`readPassword()` returns a `char` array so it can be totally and immediately removed from memory (Strings live in a pool in memory and are garbage collected, so an array is safer).

KEY POINTS

- An I/O Stream is a sequence of data that represents the content of a file.
- An input stream is for reading and an output stream is for writing.
- In the `java.io` package, we can find classes to work with byte and character streams.
- There are four main abstract classes from which the rest of the classes extend from `InputStream`, `OutputStream`, `Reader`, `Writer`.
- `java.io` classes can be classified as:
 - Either for byte streams or character streams
 - Either for input or for output
 - Either wrappers or non-wrappers
- Java initializes and provides three stream objects as public static fields of the `java.lang.System` class:
 - `InputStream System.in`
The standard input stream (typically the input from the keyboard)
 - `PrintStream System.out`
The standard output stream (typically the default display output)
 - `PrintStream System.err`
The standard error output stream (typically the default display output)
- The table on the next page summarizes the classes reviewed in this chapter:

Class	Extends From	Main Constructor Arguments	Main Methods	Description
File	–	• String	• exists • getParent • isDirectory • isFile • listFiles • mkdirs • delete • renameTo	Represents a file or directory.
FileInputStream	InputStream	• File • String	• read	Reads file content as bytes.
FileOutputStream	OutputStream	• File • File, boolean • String • String, boolean	• write	Writes file content as bytes.
FileReader	Reader	• File • String	• read	Read file content as character.
FileWriter	Writer	• File • File, boolean • String • String, boolean	• write	Write file content as character.
BufferedReader	Reader	• Reader • Reader, int	• readLine • read	Reads text to a character stream, buffering characters for efficiency.
BufferedWriter	Writer	• Writer • Writer, int	• newLine • write	Writes text to a character stream, buffering characters for efficiency.
ObjectInputStream	InputStream	• InputStream	• readObject	Deserializes primitive data and objects.
ObjectOutputStream	OutputStream	• OutputStream	• writeObject	Serializes primitive data and objects.
PrintWriter	Writer	• File • OutputStream • String • Writer	• format • print • printf • println	Writes formatted data to a character stream.
Console	–	–	• readLine • readPassword • format • printf	Provides access to the console, if any.

SELF TEST

1. Which of the following is a valid way to create a PrintWriter object?

- A. new PrintWriter(new Writer("file.txt"));
- B. new PrintWriter();
- C. new PrintWriter(new FileReader("file.txt"));
- D. new PrintWriter(new FileOutputStream("file.txt"));

2. Given:

```
try (Writer w = new FileWriter("/file.txt")) {  
    w.write('1');  
} catch(IOException e) { /* ... */ }
```

Which of the following is the result of executing the above lines if the file already exists?

- A. It overwrites the file
- B. It appends 1 to the file
- C. Nothing happens since the file already exists
- D. An IOException is thrown

3. Which of the following is the type of the System.in object?

- A. Reader
- B. InputStream
- C. BufferedReader
- D. BufferedInputStream

4. Given:

```
class Test {  
    int val = 54;  
}  
  
public class Question_23_4 {  
    public static void main(String[] args) {  
        Test t = new Test();  
        try (ObjectOutputStream oos =  
new ObjectOutputStream(new FileOutputStream("d.dat"))) {  
            oos.writeObject(t);  
        } catch (IOException e) {  
            System.out.println("Error");  
        }  
    }  
  
}
```

Which of the following is the result of executing the above lines?

- A. Nothing is printed, the class is serialized in d.dat
- B. Nothing is printed, but the class is not serialized
- C. Error
- D. An runtime exception is thrown

5. What does the flush() method do?

- A. It marks the stream as ready to be written.
- B. It closes the stream.
- C. It writes the data stored in disk to a cache.
- D. It writes the data stored in a cache to disk.

ANSWERS

1. The correct answer is D.

Option A is not valid. Writer is an abstract class and cannot be instantiated.

Option B is not valid. PrintWriter has no default constructor.

Option C is not valid. PrintWriter doesn't accept a Reader as an argument in any of its constructors.

Option D is valid. PrintWriter accepts a subclass of OutputStream as a constructor argument.

2. The correct answer is A.

The default behavior of FileOutputStream and FileWriter is to overwrite the file.

3. The correct answer is B.

System.in is an InputStream.

4. The correct answer is C.

The class the program is trying to serialize doesn't implement the java.io.Serializable interface. When that happens, the writeObject() method throws a java.io.NotSerializableException, which is an IOException. This way, the exception is caught and "Error" is printed.

5. The correct answer is D.

The flush() method writes to the disk the data cached in memory.

Chapter TWENTY-FOUR

NIO.2

Exam Objectives

- 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.

NIO.2

In the last chapter, we reviewed the classes of the `java.io` package.

In the first versions of Java, this package, especially the `File` class, provided support for file operations. However, it had some problems, like lacking functionality and limited file attribute support.

For that reason, Java 1.4 introduced the NIO (Non-blocking Input/Output) API in the package `java.nio` implementing new functionality like channels, buffering, and new charsets.

However, this API didn't entirely solve the problems with the `java.io` package, so in Java 7, the NIO.2 API was added in the `java.nio.file` package (actually, since this is a new package, NIO.2 is not an update to the NIO API, besides, they focus on different things).

NIO.2 provides better support for accessing files and the file system, symbolic links, interoperability, and exceptions among others.

The primary classes of `java.nio.file`, `Path`, `Paths`, and `Files`, are intended to provide an easier way to work with files and to be a replacement for the `java.io.File` class.

These classes will be the focus of this chapter. Let's start with `Path` and `Paths`.

THE PATH INTERFACE

In the previous chapter, we also reviewed concepts like file systems and paths.

Well, the Path interface defines an object that represents the path to a file or a directory.

When you think about the fact that a path varies between different file systems, it makes sense that Path is an interface. Thanks to this, Java transparently handles different implementations between platforms.

For example, here are some differences between Windows-based and Unix-based systems:

- Windows-based systems are not case sensitive while Unix-based systems are.
- In Windows-based systems, paths are separated by backslashes. In Unix-based systems, by forward slashes.
- In Windows-based systems, the root path is a drive letter (generally c:\). In Unix-based systems, it's a forward slash (/).
- Because of that, in Windows-based systems, an absolute path starts with a drive letter (like c:\temp\file.txt). In Unix-based systems, it starts with a forward slash (like /temp/file.txt).

Since Path is an interface and Java handles its implementations, we have to use a utility class to create Path instances.

`java.nio.file.Paths` is this class. It provides two methods to create a `Path` object:

```
static Path get(String first, String... more)
static Path get(URI uri)
```

Be very careful with the names:

Path is the **interface** with methods to work with paths.

Paths is the **class** with static methods to create a `Path` object.

With the first version of `Paths.get()` you can create a `Path` object in these ways:

```
// With an absolute path in windows
Path pathWin = Paths.get("c:\\\\temp\\\\file.txt");
// With an absolute path in unix
Path pathUnix = Paths.get("/temp/file.txt");
// With a relative path
Path pathRelative = Paths.get("file.txt");
//Using the varargs parameter
// (the separator is inserted automatically)
Path pathByParts = Paths.get("c:", "temp", "file.txt");
```

With the second version, you have to use a `java.net.URI` instance. Since we're working with files, the `URI` schema must be `file://`:

```
try {
    Path fileURI = Paths.get(new URI("file:///c:/temp/file.txt"));
} catch (URISyntaxException e) {
    //This checked exception is thrown by the URI constructor
}
```

Chapter TWENTY-FOUR

If you don't want to catch `URISyntaxException`, you can use the static method `URI.create(String)`, that wraps that exception in a `IllegalArgumentException` (a subclass of `RuntimeException`):

```
Path fileURI = Paths.get(URI.create("file:///c:/temp/file.txt"));
```

Notice the three slashes. `file:///` represents an absolute path (the `file://` schema plus another slash for the root directory). We can test this with the help of the `toAbsolutePath()` method, which returns the absolute path representation of a `Path` object:

```
Path fileURI = Paths.get(URI.create("file:///file.txt"));
System.out.println(fileURI.toAbsolutePath());
```

This will print either:

```
C:\file.txt // in Windows-based systems
/file.txt // Or in Unix-based systems
```

We can also create a `Path` from a `File` and vice-versa:

```
File file = new File("/file.txt");
Path path = file.toPath();

path = Paths.get("/file.txt");
file = path.toFile();
```

And just to make clear that the `Path` instance is system-dependent, let me tell you that `Paths.get()` is actually equivalent to:

```
Path path = FileSystems.getDefault().getPath("c://temp");
```

As you can see from the examples, the absolute path representation of a Path object has a root component (either c:\ or /) and a sequence of names separated by a (forward or backward) slash.

These names represent the directories needed to navigate to the target file or directory. The last name in the sequence represents the name of the target file or directory.

For example, the elements of the path c:\temp\dir1\file.txt (or its Unix equivalent, /temp/dir1/file.txt) are:

Root: c:\ (or /)
Name 1: temp
Name 2: dir1
Name 3: file.txt

The Path object has some methods to get this information. Except for `toString()` and `getNameCount()`, each of these methods returns a Path object:

```
Path path = Paths.get("C:\\temp\\dir1\\file.txt");
// Or Path path = Paths.get("/temp/dir1/file.txt");
System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " + path.getNameCount());
// Indexes start from zero
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("getName(2): " + path.getName(2));
// subpath(beginIndex, endIndex) from beginIndex to endIndex-1
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

The output:

Chapter TWENTY-FOUR

```
toString(): C:\temp\dir1\file.txt // Or /temp/dir1/file.txt
getFileName(): file.txt
getNameCount(): 3
getName(0): temp
getName(1): dir1
getName(2): file.txt
subpath(0,2): temp\dir1           // Or temp\dir1
getParent(): C:\temp\dir1         // Or /temp/dir1
getRoot(): C:\                   // Or /
```

Passing an invalid index to getName() and subpath() will throw an IllegalArgumentException (a RuntimeException).

If the path is specified as a relative one (and assuming this code is executed from the c:\temp directory):

```
Path path = Paths.get("dir1\\file.txt");// Or dir1/file.txt
System.out.println("toString(): " + path.toString());
System.out.println("getFileName(): " + path.getFileName());
System.out.println("getNameCount(): " + path.getNameCount());
System.out.println("getName(0): " + path.getName(0));
System.out.println("getName(1): " + path.getName(1));
System.out.println("subpath(0,2): " + path.subpath(0,2));
System.out.println("getParent(): " + path.getParent());
System.out.println("getRoot(): " + path.getRoot());
```

The output:

```
toString(): dir1\file.txt          // Or dir1/file.txt
getFileName(): file.txt
getNameCount(): 2
getName(0): dir1
getName(1): file.txt
subpath(0,2): dir1\file.txt        // Or dir1/file.txt
getParent(): dir1
getRoot(): null
```

When working with paths you can use:

- . to refer to the current directory
- .. to refer to the parent directory

For example:

```
Path p1 = Paths.get("/temp./file.txt"); // refers to /temp/file.txt
Path p2 = Paths.get(
    "/temp/dir1.../file.txt"); // refers to /temp//file.txt
```

In these cases, you can use the `normalize()` method to remove redundancies like . and .. (in other words, to "normalize" it):

```
Path path = Paths.get("/temp/dir1.../file.txt");
System.out.println(path); // /temp/dir1.../file.txt
Path path2 = path.normalize();
System.out.println(path2); // /temp/file.txt
```

This method does not access the file system to know if a file exists, so removing .. and a preceding name from a path may result in a path that no longer references the original file. This can happen when that previous name is a symbolic link (a reference to another file).

It's better to use the `toRealPath()` method:

```
Path toRealPath(LinkOption... options) throws IOException
```

This method does the following:

- If `LinkOption.NOFOLLOW_LINKS` is passed as an argument, symbolic links are not followed (by default it does).
- If the path is relative, it returns an absolute path.
- It returns a Path with redundant elements removed (if any).

Chapter TWENTY-FOUR

We can combine two paths. There are two cases.

First case. If we have an absolute path and we want to combine it with a second path that doesn't have a root element (a partial path), the second path is appended:

```
Path path = Paths.get("/temp");
System.out.println(path.resolve("newDir")); // /temp/newDir
```

Second case. If we have a partial or relative path, and we want to combine it with an absolute path, this absolute path is returned:

```
Path path = Paths.get("newDir");
System.out.println(path.resolve("/temp")); // /temp
```

`relativize()` is another interesting method.

`path1.relativize(path2)` is like saying *give me a path that shows how to get from path1 to path2*.

For example, if we are in directory `/temp` and we want to go to `/temp/dir1/subdir`, we have to go first to `dir1` and then to `subdir`:

```
Path path1 = Paths.get("temp");
Path path2 = Paths.get("temp/dir1/file.txt");
Path path1ToPath2 = path1.relativize(path2); // dir1/file.txt
```

If the paths represent two relatives paths without any other information, they are considered siblings, so you have to go to the parent directory and then go to the other directory:

```
Path path1 = Paths.get("dir1");
Path path1ToPath2 = path1.relativize(Paths.get("dir2")); // ../dir2
```

Notice that both examples use relative paths.

If one of the paths is an absolute path, a relative path cannot be constructed because of the lack of information and a `IllegalArgumentException` will be thrown.

If both paths are absolute, the result is system-dependent.

Path implements the `Iterable` interface so you can do something like this:

```
Path path = Paths.get("c:\\\\temp\\\\dir1\\\\file.txt");
for(Path name : path) {
    System.out.println(name);
}
```

The output:

```
temp
dir1
file.txt
```

Path implements the `Comparable` interface and the `equals()` method to test two paths for equality.

`compareTo()` compares two paths lexicographically. It returns:

- Zero if the argument is equal to the path,
- A value less than zero if this path is lexicographically less than the argument, or
- A value greater than zero if this path is lexicographically greater than the argument.

The `equals()` implementation is system-dependent (for example, it's case insensitive on Windows systems). However, it returns `false` if the argument is not a Path or if it belongs to a different file system.

Chapter TWENTY-FOUR

In addition, the methods `startsWith()` and `endsWith()` both test whether a path begins or ends with some String (in this case, the methods return true only if the string represents an actual element) or Path. So given:

```
Path absPath = Paths.get("c:\\temp\\dir1\\file.txt");
Path relPath = Paths.get("temp\\dir1\\file.txt");
```

boolean startsWith(Path other)

```
absPath.startsWith(Paths.get("c:\\temp\\file.txt"));           // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\img.jpg"));      // false
absPath.startsWith(Paths.get("c:\\temp\\dir1\\"));             // true
absPath.startsWith(relPath);                                  // false
```

boolean startsWith(String other)

```
relPath.startsWith("t");                                     // false
relPath.startsWith("temp");                                  // true
relPath.startsWith("temp\\d");                             // false
relPath.startsWith("temp\\dir1");                         // true
```

boolean endsWith(Path other)

```
absPath.endsWith("file.txt");                            // true
absPath.endsWith("d:\\temp\\dir1\\file.txt");            // false
relPath.endsWith(absPath);                             // false
```

boolean endsWith(String other)

```
relPath.endsWith("txt");                                // false
relPath.endsWith("file.txt");                           // true
relPath.endsWith("\\\\dir1\\\\file.txt");                 // false
relPath.endsWith("dir1\\\\file.txt");                   // true
```

These methods don't take into account trailing separators, so if we have the Path `temp/dir1`, invoking, for example, `endsWith()` with `dir1/`, it returns true.

THE FILES CLASS

The `java.nio.file.Files` class has static methods for common operations on files and directories. In contrast with the `java.io.File` class, all methods of `Files` work with `Path` objects (so don't confuse `File` and `Files`).

For example, we can check if a path actually exists (or doesn't exist) with the methods:

```
static boolean exists(Path path, LinkOption... options)  
static boolean notExists(Path path, LinkOption... options)
```

If `LinkOption.NOFOLLOW_LINKS` is present, symbolic links are not followed (by default they are).

We can check if a path is readable (it's not if the file doesn't exist or if the JVM doesn't have the privileges to access it):

```
static boolean isReadable(Path path)
```

We can check if a path is writable (it's not if the file doesn't exist or if the JVM doesn't have the privileges to access it):

```
static boolean isWritable(Path path)
```

We can check if a file exists and is executable:

```
static boolean isExecutable(Path path)
```

Or even check if two paths refer to the same file (useful if one path represents a symbolic link). If both `Path` objects are equal then this method returns true without checking if the file exists:

```
static boolean isSameFile(Path path,  
                         Path path2) throws IOException
```

Chapter TWENTY-FOUR

To read a file, we can load the entire file into memory (only useful for small files) with the methods:

```
static byte[] readAllBytes(Path path) throws IOException
static List<String> readAllLines(Path path) throws IOException
static List<String> readAllLines(Path path,
                                Charset cs) throws IOException
```

For example:

```
try {
    // By default it uses StandardCharsets.UTF_8
    List<String> lines = Files.readAllLines(Paths.get("file.txt"));
    lines.forEach(System.out::println);
}
} catch (IOException e) { /** */ }
```

Or to read a file in an efficient way:

```
static BufferedReader newBufferedReader(Path path) throws IOException
static BufferedReader newBufferedReader(Path path,
                                         Charset cs) throws IOException
```

For example:

```
Path path = Paths.get("/temp/dirl/files.txt");
// By default it uses StandardCharsets.UTF_8
try (BufferedReader reader = Files.newBufferedReader(path,
                                                       StandardCharsets.ISO_8859_1)) {
    String line = null;
    while((line = reader.readLine()) != null)
        System.out.println(line);
} catch (IOException e) { /** ... */ }
```

The Files class has two methods to delete files/directories.

```
static void delete(Path path) throws IOException
```

It removes the file/directory or throws an exception if something fails:

```
try {
    Files.delete(Paths.get("/temp/dir1/file.txt"));
    Files.delete(Paths.get("/temp/dir1"));
} catch (NoSuchFileException nsfe) {
    // If the file/directory doesn't exists
} catch (DirectoryNotEmptyException dnee) {
    // To delete a directory, it must be empty,
    // otherwise, this exception is thrown
} catch (IOException ioe) {
    // File permission or other problems
}
```

The second method is:

```
static boolean deleteIfExists(Path path) throws IOException
```

This method returns true if the file was deleted or false if the file could not be removed because it did not exist, in other words, unlike the first method, this doesn't throw a NoSuchFileException (but it still throws a DirectoryNotEmptyException and an IOException for other problems):

```
try {
    Files.deleteIfExists(Paths.get("/temp/dir1/file.txt"));
} catch (DirectoryNotEmptyException dnee) {
    // To delete a directory, it must be empty,
} catch (IOException ioe) {
    // File permission or other problems
}
```

Chapter TWENTY-FOUR

To copy files/directories, we have the method:

```
static Path copy(Path source, Path target,  
CopyOption... options) throws IOException
```

It returns the path to the target file, and when copying a directory, its content won't be copied.

By default, the copy fails if the destination file already exists. Also, file attributes won't be copied, and when copying a symbolic link, its target will be copied.

We can customize this behavior with the following CopyOption enums:

- **StandardCopyOption.REPLACE_EXISTING**

Performs the copy when the target already exists. If the target is a symbolic link, the link itself is copied and If the target is a non-empty directory, a FileAlreadyExistsException is thrown.

- **StandardCopyOption.COPY_ATTRIBUTES**

Copies the file attributes associated with the file to the target file. The exact attributes supported are file system and platform dependent, except for last-modified-time, which is supported across platforms.

- **LinkOption.NOFOLLOW_LINKS**

Indicates that symbolic links should not be followed, just copied.

Here's an example:

```
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;  
...  
try {  
    Files.copy(Paths.get("in.txt"), Paths.get("out.txt"),  
        REPLACE_EXISTING);  
} catch (IOException e) { /* ... */ }
```

There are methods to copy between a stream and a Path also:

```
static long copy(InputStream in, Path target,  
                  CopyOption... options) throws IOException
```

Copies all bytes from an input stream to a file. By default, the copy fails if the target already exists or is a symbolic link. If the StandardCopyOption.REPLACE_EXISTING option is specified, and the target file already exists, then it is replaced if it's not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link is replaced. Actually, in Java 8, the REPLACE_EXISTING option is the only option required to be supported by this method.

```
static long copy(Path source,  
                 OutputStream out) throws IOException
```

Copies all bytes from a file to an output stream.

For example:

```
try (InputStream in = new FileInputStream("in.csv");  
      OutputStream out = new FileOutputStream("out.csv")) {  
    Path path = Paths.get("/temp/in.txt");  
    // Copy stream data to a file  
    Files.copy(in, path);  
    // Copy the file data to a stream  
    Files.copy(path, out);  
}  
catch (IOException e) { /* ... */ }
```

Chapter TWENTY-FOUR

To move or rename a file/directory, we have the method:

```
static Path move(Path source, Path target,  
                 CopyOption... options) throws IOException
```

By default, this method will follow links, throw an exception if the file already exists, and not perform an atomic move.

We can customize this behavior with the following CopyOption enums:

- **StandardCopyOption.REPLACE_EXISTING**
Performs the move when the target already exists. If the target is a symbolic link, only the link itself is moved.
- **StandardCopyOption.ATOMIC_MOVE**
Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown.

This method can move a non-empty directory. However, if the target exists, trying to move a non-empty directory will throw a `DirectoryNotEmptyException`. This exception will also be thrown when trying to move a non-empty directory across a drives or partitions.

For example:

```
try {  
    // Move or rename dir1 to dir2  
    Files.move(Paths.get("c:\\temp\\dir1"),  
              Paths.get("c:\\temp\\dir2"));  
} catch (IOException e) { /* ... */ }
```

MANAGING METADATA

When talking about a file system, metadata give us information about a file or directory, like its size, permissions, creation date, etc. This information is referred as attributes, and some of them are system-dependent.

The Files class has some methods to get or set some attributes from a Path object:

```
static long size(Path path) throws IOException
```

Returns the size of a file (in bytes).

```
static boolean isDirectory(Path path, LinkOption... options)
```

Tests whether a file is a directory.

```
static boolean isRegularFile(Path path, LinkOption... options)
```

Tests whether a file is a regular file.

```
static boolean isSymbolicLink(Path path)
```

Tests whether a file is a symbolic link.

```
static boolean isHidden(Path path) throws IOException
```

Tells whether a file is considered hidden.

```
static FileTime getLastModifiedTime(Path path,  
LinkOption... options) throws IOException
```

```
static Path setLastModifiedTime(Path path,  
FileTime time) throws IOException
```

Returns or updates a file's last modified time.

```
static UserPrincipal getOwner(Path path,  
LinkOption... options) throws IOException
```

```
static Path setOwner(Path path,  
UserPrincipal owner) throws IOException
```

Returns or updates the owner of the file.

Chapter TWENTY-FOUR

In methods that take an optional `LinkOption.NOFOLLOW_LINKS`, symbolic links are not followed (by default they are).

In the case of `getLastModifiedTime()` and `setLastModifiedTime()` the class `java.nio.file.attribute.FileTime` represents the value of a file's time stamp attribute.

We can create an instance of `FileTime` with these static methods:

```
static FileTime from(Instant instant)
static FileTime from(long value, TimeUnit unit)
static FileTime fromMillis(long value)
```

And from a `FileTime` we can get an `Instant` or milliseconds as `long`:

```
Instant toInstant()
long toMillis()
```

For example:

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    FileTime ft = Files.getLastModifiedTime(path);
    Files.setLastModifiedTime(path,
        FileTime.fromMillis(ft.toMillis + 1000)); // adds a second
} catch (IOException e) { /** ... */ }
```

In the case of `getOwner()` and `setOwner()` the interface `java.nio.file.attribute.UserPrincipal` is an abstract representation of an identity that can be used like this:

```
try {
    Path path = Paths.get("/temp/dir1/file.txt");
    // FileSystems.getDefault() also gets a FileSystem object
    UserPrincipal owner = path.getFileSystem()
        .getUserPrincipalLookupService()
        .lookupPrincipalByName("jane");

    Files.setOwner(path, owner);
} catch (IOException e) { /** ... */ }
```

These methods are useful to get or update a single attribute. But we can also get a group of related attributes by functionality or by a particular systems implementation as a *view*.

The three most common view classes are:

- **java.nio.file.attribute.BasicFileAttributeView**

Provides a view of basic attributes supported by all file systems.

- **java.nio.file.attribute.DosFileAttributeView**

Extends BasicFileAttributeView to support additionally a set of DOS attribute flags that are used to indicate if the file is read-only, hidden, a system file, or archived.

- **java.nio.file.attribute.PosixFileAttributeView**

Extends BasicFileAttributeView with attributes supported on POSIX systems, such as Linux and Mac. Examples of these attributes are file owner, group owner, and related access permissions.

You can get a file attribute view of a given type to read or update a set of attributes with the method:

```
static <V extends FileAttributeView> V getFileAttributeView(  
    Path path, Class<V> type, LinkOption... options)
```

For example, BasicFileAttributeView has only one update method:

```
try {  
    Path path = Paths.get("/temp/dir/file.txt");  
    BasicFileAttributeView view =  
        Files.getFileAttributeView(path, BasicFileAttributeView.class);  
    // Get a class with read-only attributes  
    BasicFileAttributes readOnlyAttrs = view.readAttributes();  
    FileTime lastModifiedTime = FileTime.from(Instant.now());  
    FileTime lastAccessTime = FileTime.from(Instant.now());  
    FileTime createTime = FileTime.from(Instant.now());  
    //If any argument is null, the corresponding value is not changed  
    view.setTimes(lastModifiedTime, lastAccessTime, createTime);  
} catch (IOException e) { /* ... */ }
```

Chapter TWENTY-FOUR

Most of the time, you'll work with the read-only versions of the file views. In this case, you can use the following method to get them directly:

```
static <A extends BasicFileAttributes> A  
    readAttributes(Path path, Class<A> type, LinkOption... options)  
        throws IOException
```

The second parameter is the return type of the method, the class that contains the attributes to use (notice that all attributes classes extend from `BasicFileAttributes` because it contains attributes common to all file systems). The third argument is when you want to follow symbolic links.

Here's an example of how to access the file attributes of a file using the `java.nio.file.attribute.BasicFileAttributes` class:

```
try {  
    Path path = Paths.get("/temp/dirl/file.txt");  
    BasicFileAttributes attr = Files.readAttributes(  
        path, BasicFileAttributes.class);  
  
    // Size in bytes  
    System.out.println("size(): " + attr.size());  
    // Unique file identifier (or null if not available)  
    System.out.println("fileKey(): " + attr.fileKey());  
  
    System.out.println("isDirectory(): " + attr.isDirectory());  
    System.out.println("isRegularFile(): " + attr.isRegularFile());  
    System.out.println("isSymbolicLink(): " + attr.isSymbolicLink());  
    // Is something other than a file, directory, or symbolic link?  
    System.out.println("isOther(): " + attr.isOther());  
  
    // The following methods return a FileTime instance  
    System.out.println("creationTime(): " + attr.creationTime());  
    System.out.println("lastModifiedTime(): " + attr.lastModifiedTime());  
    System.out.println("lastAccessTime(): " + attr.lastAccessTime());  
} catch (IOException e) { /* ... */ }
```

KEY POINTS

- The primary classes of `java.nio.file` are `Path`, `Paths`, and `Files`. They are intended to be a replacement of the `java.io.File` class.
- The `Path` interface defines an object that represents the path to a file or a directory.
- `java.nio.file.Paths` provides methods to create a `Path` object.
- The absolute path representation of a `Path` object has a root component (either `c:\` or `/`) and a sequence of names separated by a (forward or backward) slash.
- The `Path` interface has methods to get the elements of the path, normalize paths, and get attributes of the path (`isAbsolute()`, `getFileSystem()`, etc), among others. It also implements `Comparable` and `equals()` to test for equality.
- The `java.nio.file.Files` class has static methods for common operations on files and directories. In contrast with the `java.io.File` class, all methods of `Files` work with `Path` objects.
- Examples of these operations are checking the existence of a file, copying, moving, deleting, and reading.
- You can also get attributes of a file individually (with methods like `isHidden()`) or in a group through views.
- The three most common view classes are `BasicFileAttributeView`, `DosFileAttributeView`, and `PosixFileAttributeView`.
- You can get a file attribute view of a given type to read or update a set of attributes with the method `getFileAttributeView()`.
- You can get a class that is a read-only version of the view with the method `readAttributes()`.

SELF TEST

1. Given:

```
Path path1 = Paths.get("/projects/work/..../fun");
Path path2 = Paths.get("games");
System.out.println(path1.resolve(path2));
```

Which of the following is the result of executing the above lines?

- A. /project/work/fun/games
- B. /project/fun/games
- C. /project/work/..../fun/games
- D. games

2. Given:

```
Path path = Paths.get("c:\\\\Users\\\\mark");
```

Which of the following will return Users?

- A. path.getRoot()
- B. path.getName(0)
- C. path.getName(1)
- D. path.subpath(0, 0);

3. Which of the following is not a valid CopyOption for Files.copy()?

- A. NOFOLLOW_LINKS
- B. REPLACE_EXISTING
- C. ATOMIC_MOVE
- D. COPY_ATTRIBUTES

4. Given:

```
Path path =  
    Paths.get("c:\\\\.\\temp\\\\data\\\\..\\..\\dir\\..\\file.txt");  
try {  
    path = path.toRealPath();  
} catch (IOException e) {}  
System.out.println(path.subpath(1,2));
```

Which is the result?

- A. temp
- B. data
- C. dir
- D. file.txt

5. Which of the following is a valid way to set a file's create time?

- A. FileTime time = FileTime.from(Instance.now());
Files.getFileAttributeView(path, BasicFileAttributeView.class)
 .setTimes(null, time, null);
- B. Files.setCreateTime(path,
 FileTime.from(Instance.now()));
- C. Files.getFileAttributeView(path, BasicFileAttributeView.class)
 .setTimes(null, null, Instance.now());
- D. FileTime time = FileTime.from(Instance.now());
Files.getFileAttributeView(path, BasicFileAttributeView.class)
 .setTimes(null, null, time);

ANSWERS

1. The correct answer is C.

`Path.resolve()` doesn't remove redundancy in paths (that's done with `Path.normalize()` or `Path.toRealPath()`), it just combines two paths.

2. The correct answer is B.

`path.getRoot()` returns `c:\.`

`path.getName(0)` returns `Users`.

`path.getName(1)` returns `mark`.

`path.subpath(0, 0)` throws an exception.

`getName()` returns the element parts of a path. The first element is at index zero (the root doesn't count as an element).

3. The correct answer is C.

`CopyOption` is an interface implemented by the enumerations `StandardCopyOptions` (with the values `ATOMIC_MOVE` (not supported), `REPLACE_EXISTING`, and `COPY_ATTRIBUTES`) and `LinkOption` (with the value `NOFOLLOW_LINKS`).

4. The correct answer is D.

`.` means the current directory and `..` means the parent directory. The method `toRealPath()` turns the redundant path into `C:\temp\file.txt`. `path.subpath(1, 2)` give us the element at index 1, `file.txt`.

5. The correct answer is D.

A file creation time attribute can only be set with `BasicFileAttributeView`. The right way to do it is with a `FileTime` instance as shown in Option D.

Chapter TWENTY-FIVE

Files and Streams

Exam Objectives

- Use Stream API with NIO.2.

NEW STREAM METHODS IN NIO.2

With the arrival of streams to Java, some complicated file operations to implement in NIO.2 (that often required an entire class) have been simplified.

`java.nio.file.Files`, a class with `static` methods that we reviewed in the last chapter, added operations that return implementations of the `Stream` interface.

The methods are:

```
static Stream<Path> find(Path start,  
                           int maxDepth,  
                           BiPredicate<Path, BasicFileAttributes> matcher,  
                           FileVisitOption... options)  
  
static Stream<Path> list(Path dir)  
  
static Stream<String> lines(Path path)  
static Stream<String> lines(Path path, Charset cs)  
  
static Stream<Path> walk(Path start,  
                           FileVisitOption... options)  
static Stream<Path> walk(Path start,  
                           int maxDepth,  
                           FileVisitOption... options)
```

An important thing to notice is that the returned streams are **LAZY**, which means that the elements are not loaded (or read) until they are used. This is a significant performance enhancement.

Let's describe each method starting with `Files.list()`.

FILES.LIST()

```
static Stream<Path> list(Path dir) throws IOException
```

This method iterates over a directory to return a stream whose elements are Path objects that represent the entries of that directory.

```
try(Stream<Path> stream = Files.list(Paths.get("/temp"))) {  
    stream.forEach(System.out::println);  
} catch(IOException e) {  
    e.printStackTrace();  
}
```

A possible output:

```
/temp/dir1  
/temp/dir2  
/temp/file.txt
```

The use of a try-with-resources is recommended so the stream's close method can be invoked to close the file system resources.

As you can see, this method lists directories and files in the specified directory. However, it is not recursive, in other words, it **DOESN'T** traverses subdirectories.

Another two important considerations about this method:

- If the argument doesn't represent a directory, an exception is thrown.
- This method is thread safe but is *weakly consistent*, meaning that while iterating a directory, updates to it may or may not be reflected in the returned stream.

FILES.WALK()

```
static Stream<Path> walk(Path start,
                           FileVisitOption... options)
                           throws IOException
static Stream<Path> walk(Path start,
                           int maxDepth,
                           FileVisitOption... options)
                           throws IOException
```

This method also iterates over a directory to return a stream whose elements are Path objects that represent the entries of that directory.

The difference with `Files.list()` is that `Files.walk()` **DOES** recursively traverse the subdirectories.

It does it with a *depth-first* strategy, which traverses the directory structure from the root to all the way down a subdirectory before exploring another one.

For example, considering this structure:

```
/temp/
    /dir1/
        /subdir1/
            111.txt
        /subdir2/
            121.txt
            122.txt
    /dir2/
        21.txt
        22.txt
    file.txt
```

Chapter TWENTY-FIVE

The following code:

```
try(Stream<Path> stream = Files.walk(Paths.get("/temp"))) {  
    stream.forEach(System.out::println);  
} catch(IOException e) {  
    e.printStackTrace();  
}
```

Will output:

```
/temp  
/temp/dir1  
/temp/dir1/subdir1  
/temp/dir1/subdir1/111.txt  
/temp/dir1/subdir2  
/temp/dir1/subdir2/121.txt  
/temp/dir1/subdir2/122.txt  
/temp/dir2  
/temp/dir2/21.txt  
/temp/dir2/22.txt  
/temp/file.txt
```

By default, this method uses a maximum subdirectory depth of `Integer.MAX_VALUE`. But you can use the overloaded version that takes the maximum depth as the second parameter:

```
try(Stream<Path> stream = Files.walk(Paths.get("/temp"), 1)) {  
    stream.forEach(System.out::println);  
} catch(IOException e) {  
    e.printStackTrace();  
}
```

The output is:

```
/temp  
/temp/dir1  
/temp/dir2  
/temp/file.txt
```

A value of 0 means that only the starting directory is visited.

Also, this method doesn't follow symbolic links by default.

Symbolic links can cause a *cycle*, an infinite circular dependency between directories. However, this method is smart enough to detect a cycle and throw a `FileSystemLoopException`.

To follow symbolic links, just use the argument of type `FileVisitOption` (preferably, also using the maximum depth argument) this way:

```
try(Stream<Path> stream =  
        Files.walk(Paths.get("/temp"),  
                  2,  
                  FileVisitOption.FOLLOW_LINKS)  
    ) {  
    stream.forEach(System.out::println);  
} catch(IOException e) {  
    e.printStackTrace();  
}
```

Just like `Files.list()`, it's recommended to use `Files.walk()` with try-with-resources, if the argument doesn't represent a directory, an exception is thrown and it's considered a *weakly consistent* method.

FILES.FIND()

```
static Stream<Path> find(Path start,  
                           int maxDepth,  
                           BiPredicate<Path, BasicFileAttributes> matcher,  
                           FileVisitOption... options)  
throws IOException
```

This method is similar to `Files.walk()`, but takes an additional argument of type `BiPredicate` that is used to filter the files and directories.

Remember that a `BiPredicate` takes two arguments and returns a boolean. In this case:

- The first argument is the `Path` object that represents the file or directory.
- The second argument is a `BasicFileAttributes` object that represents the attributes of the file or directory in the file system (like creation time, if it's a file, directory or symbolic link, size, etc.).
- The returned boolean indicates if the file should be included in the returned stream.

The following example returns a stream that includes just directories:

```
BiPredicate<Path, BasicFileAttributes> predicate =  
    (path, attrs) -> {  
        return attrs.isDirectory();  
    };  
int maxDepth = 2;
```

```
try(Stream<Path> stream =  
    Files.find(Paths.get("/temp"), maxDepth, predicate)) {  
    stream.forEach(System.out::println);  
} catch(IOException e) {  
    e.printStackTrace();  
}
```

A possible output can be:

```
/temp  
/temp/dir1  
/temp/dir1/subdir1  
/temp/dir1/subdir2  
/temp/dir2
```

Like `Files.walk()`, it can also take a `FileVisitOption` for visiting symbolic links, it's recommended to use it in a `try-with-resources` and throws an exception if it cannot read a file or directory.

FILES.LINES()

```
static Stream<String> lines(Path path, Charset cs)
    throws IOException
static Stream<String> lines(Path path)
    throws IOException
```

This method reads all the lines of a file as a stream of Strings.

As the stream is lazy, it doesn't load all the lines into memory, only the line read at any given time. If the file doesn't exist, an exception is thrown.

The file's bytes are decoded using the specified charset or with UTF - 8 by default.

For example:

```
try(Stream<String> stream =
    Files.lines(Paths.get("/temp/file.txt")))
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

In Java 8, a `lines()` method was added to `java.io.BufferedReader` as well:

```
Stream<String> lines()
```

The stream is lazy and its elements are the lines read from the `BufferedReader`.

KEY POINTS

- In Java 8, new methods that return implementations of the Stream interface have been added to `java.nio.file.Files`.
- The returned streams are **LAZY**, which means that the elements are not loaded (or read) until they are used.
- The use of a try-with-resources with these methods is recommended so that the stream's close method can be invoked to close the file system resources.
- `Files.list()` iterates over a directory to return a stream whose elements are Path objects that represent the entries of that directory.
- This method lists directories and files of the specified directory. However, it is not recursive, in other words, it **DOESN'T** traverse subdirectories.
- `Files.walk()` also iterates over a directory in a depth-first strategy to return a stream whose elements are Path objects that represent the entries of that directory.
- The difference with `Files.list()` is that `Files.walk()` **DOES** recursively traverse the subdirectories. You can also pass the maximum traversal depth and an option to follow symbolic links.
- `Files.find()` is similar to `Files.walk()`, but takes an additional argument of type `BiPredicate<Path, BasicFileAttributes>` that is used to filter the files and directories.
- `Files.lines()` reads all the lines of a file as a stream of Strings without loading them all into memory.

SELF TEST

1. Given the following structure and class:

```
/temp/
```

```
    /dir1/
```

```
        1.txt
```

```
        0.txt
```

```
public class Question_20_1 {  
    public static void main(String[] args) {  
        try(Stream<Path> stream =  
            Files.walk(Paths.get("/temp"), 0)) {  
            stream.forEach(System.out::println);  
        } catch(IOException e) {}  
    }  
}
```

What is the result?

- A. /temp
- B. /temp/dir1
- /temp/0.txt
- C. /temp/0.txt
- D. Nothing is printed

2. Which of the following statements is true?

- A. Files.find() has a default subdirectory depth of Integer.MAX_VALUE.
- B. Files.find() follows symbolic links by default.
- C. Files.walk() follows symbolic links by default.
- D. Files.walk() traverses subdirectories recursively.

3. Which of the following options is equivalent to

```
Files.walk(Paths.get("."))  
    .filter(p -> p.toString().endsWith("txt"));
```

- A. Files.list(Paths.get("."))
 .filter(p -> p.toString().endsWith("txt"));
- B. Files.find(Paths.get("."),
 (p,a) -> p.toString().endsWith("txt"));
- C. Files.find(Paths.get("."), Integer.MAX_VALUE,
 p -> p.toString().endsWith("txt"));
- D. Files.find(Paths.get("."), Integer.MAX_VALUE,
 (p,a) -> p.toString().endsWith("txt"));

4. Which is the behavior of Files.lines(Path) if the Path object represents a file that doesn't exist?

- A. It returns an empty stream.
- B. It creates the file.
- C. It throws an IOException when the method is called.
- D. It throws an IOException when the stream is first used.

ANSWERS

1. The correct answer is A.

A value of zero for the `maxDepth` argument means that only the starting directory is visited (no files or subdirectories).

2. The correct answer is D.

Option A is false. `Files.find()` doesn't have a default subdirectory depth. This is passed as an argument to the method.

Option B is false. `Files.find()` follows symbolic links only if `FileVisitOption.FOLLOW_LINKS` is passed as an argument.

Option C is false. `Files.walk()` follows symbolic links only if `FileVisitOption.FOLLOW_LINKS` is passed as an argument.

Option D is true. `Files.walk()` traverses subdirectories recursively.

3. The correct answer is D.

Option A is not equivalent. `Files.list()` doesn't traverse subdirectories recursively.

Option B is not equivalent. `Files.find()` is missing the max depth argument.

Option C is not equivalent. The last argument must be a `BiPredicate`.

4. The correct answer is C.

If the file doesn't exist, `Files.line(Path)` throws an `IOException` (a `java.nio.file.NoSuchFileException`) when the method is called, before the stream is created.

Part EIGHT

Threads and Concurrency

Chapter TWENTY-SIX

Thread Basics

Exam Objectives

- Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks.
- Identify potential threading problems among deadlock, starvation, livelock, and race conditions.

THREADS

In simple words, concurrency means doing things simultaneously, in parallel. In Java, concurrency is done with threads.

Threads are units of code that can be executed at the same time. They are sometimes called lightweight processes, although, in fact, a thread is executed within a process (and every process has, at least, one thread, the main thread).

At a low level, we can create a thread in two ways.

The first (and recommendable way) is to implement the `java.lang.Runnable` interface, which only has the `run()` method:

```
class RunnableTask implements Runnable {  
    public void run() {  
        System.out.println("Running");  
    }  
}
```

And then, to create a single thread, you have to pass an instance to the constructor of the `java.lang.Thread` class and **REQUEST** the thread to start (it might not start immediately. Besides, the order and time of execution of a thread are **NOT** guaranteed):

```
Thread thread = new Thread(new RunnableTask());  
thread.start();
```

The `start()` method will call the `run()` method of the `Runnable` instance to start executing.

Chapter TWENTY-SIX

Of course, you can use an anonymous class to do it:

```
Thread thread = new Thread(new RunnableTask() {
    public void run() {
        System.out.println("Running");
    }
});
thread.start();
```

Or since Runnable is a functional interface, a lambda expression:

```
Thread thread = new Thread(() -> {
    System.out.println("Running");
});
thread.start();
```

The other (discouraged) way is to subclass Thread, which implements Runnable so you just have to override run():

```
class ThreadTask extends Thread {
    public void run() {
        System.out.println("Running");
    }
}
```

Then create an instance and call the start() method:

```
Thread thread = new ThreadTask();
thread.start();
```

However, it's better to implement Runnable on your own because with the new concurrency API, you don't have to create Thread objects directly anymore (not to mention that implementing an interface is the recommended object-oriented way to do it).

EXECUTOR SERVICE

Java 5 introduced a high-level API for concurrency, most of it implemented in the `java.util.concurrent` package.

One of the features of this API is the Executor interfaces that provide an alternative (better) way to launch and manage threads.

The `java.util.concurrent` package defines three executor interfaces:

Executor

This interface has the `execute()` method, which is designed to replace:

```
Runnable r = ...  
Thread t = new Thread(r);  
t.start();
```

With:

```
Runnable r = ...  
Executor e = ...  
e.execute(r);
```

ExecutorService

This interface extends Executor to provide more features, like the `submit()` method that accepts Runnable and Callable objects and allows them to return a value.

ScheduledExecutorService

This interface extends ExecutorService to execute tasks at repeated intervals or with a particular delay.

Chapter TWENTY-SIX

Executors use thread pools, which use worker threads. These threads are different than the threads you create with the Thread class.

When worker threads are created, they just stand idle, waiting for work. When work arrives, the executor assigns it to the idle threads from the thread pool.

This way, threads are generic, they exist independently from the Runnable tasks they execute (in contrast to a traditional thread created with the Thread class).

One type of thread pool is the fixed thread pool, which has a fixed number of threads running. If a thread is terminated while it is still in use, it's automatically replaced with a new thread. There are also expandable thread pools.

Now, most of the time, you'd want to work with ExecutorService, since it has more functionality than Executor. Since they are interfaces, to create an instance of an Executor you have to use a helper class, java.util.concurrent.Executors.

The Executors class has many static methods to create an ExecutorService, like:

```
static ExecutorService newSingleThreadExecutor()
```

which creates an Executor that uses a single worker thread

```
static ExecutorService newFixedThreadPool(int nThreads)
```

which creates a thread pool that reuses a fixed number of threads, and

```
static ScheduledExecutorService
```

```
newScheduledThreadPool(int corePoolSize)
```

which creates a thread pool that can schedule task

Let's start by creating an ExecutorService with a single thread pool:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.rangeClosed(1, 4).forEach(System.out::println);
};
System.out.println("before executing");
executor.execute(r);
System.out.println("after executing");
executor.shutdown();
```

A possible output can be:

```
before executing
after executing
1
2
3
4
```

Since there's only one thread (in addition to the main thread, don't forget that), tasks are guaranteed to be executed in the order they were submitted, and no more than one task will be active at any given time. In a real-world application, you may want to use `newFixedThreadPool()` with pool size equal to the number of processors available.

Once you're done with an ExecutorService, you have to shut it down to terminate threads and close resources. We have two methods to do it:

```
void shutdown()
List<Runnable> shutdownNow()
```

The `shutdown()` method tells the executor to stop accepting new tasks, but the previous tasks are allowed to continue until the finish. During this time, the method `isTerminated()` will return `false` until all tasks are completed, while the method `isShutdown()` will return `true` at all times.

Chapter TWENTY-SIX

The shutdownNow() method will also tell the executor to stop accepting new tasks but it will **TRY** to stop all executing tasks immediately (by interrupting the threads, but if the thread doesn't respond to interrupts, it may never terminate) and return a list of the tasks that were never started.

For example, if we execute:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    try {
        Thread.sleep(5_000);
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
};
executor.execute(r);
executor.shutdown();
```

We will have to wait around five seconds for the program to finish, but if we change shutdown() to shutdownNow(), "Interrupted" will be printed and the program will terminate immediately.

Besides execute(), there are other methods to submit a task. Let's define, first, the most critical methods and the new classes they use:

Future<?> submit(**Runnable** task)

Executes a Runnable and returns a Future representing that task.

<T> Future<T> submit(**Callable<T>** task)

Executes a Callable and returns a Future representing the result of the task.

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
    throws InterruptedException, ExecutionException
```

Executes the given tasks, returning the result of one that has completed without throwing an exception, if any. The other tasks are canceled.

```
<T> List<Future<T>> invokeAll(
    Collection<? extends Callable<T>> tasks)
    throws InterruptedException
```

Executes the given tasks returning a list of Future objects holding their status and results when all complete (either normally or by an exception).

The `java.util.concurrent.Future` class has these methods:

```
boolean cancel(boolean mayInterruptIfRunning)
```

Attempts to cancel the execution of the task. If the argument is true, the thread is interrupted. Otherwise, the task is allowed to complete.

```
V get() throws InterruptedException, ExecutionException
```

Waits for the task to complete (indefinitely), and then retrieves its result.

```
V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException
```

Waits the given time at most and then retrieves the result (if the time is reached and the result is not ready, a `TimeoutException` is thrown).

```
boolean isCancelled()
```

Returns true if this task was canceled before it completed normally.

```
boolean isDone()
```

Returns true if the task is completed.

`java.util.concurrent.TimeUnit` is an enum with the following values:

<code>TimeUnit.NANOSECONDS</code>	<code>TimeUnit.MICROSECONDS</code>
<code>TimeUnit.MILLISECONDS</code>	<code>TimeUnit.SECONDS</code>
<code>TimeUnit.MINUTES</code>	<code>TimeUnit.HOURS</code>
<code>TimeUnit.DAYS</code>	

Chapter TWENTY-SIX

`java.util.concurrent.Callable` is a functional interface that defines this method:

```
V call() throws Exception;
```

As you can see, the difference with `Runnable` is that a `Callable` can return a value and throw a checked exception.

Now the code examples. When the `submit()` method is called with a `Runnable`, the returned `Future` object returns null (because `Runnable` doesn't return a result):

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Runnable r = () -> {
    IntStream.range(1, 1_000_000).forEach(System.out::println);
};
Future<?> future = executor.submit(r);
try {
    future.get(); // Blocks until the Runnable has finished
} catch (InterruptedException | ExecutionException e) { /** ... */ }
```

When this method is called with a `Callable`, the returned `Future` object contains the result when it has finished executing:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Callable<Long> c = () ->
    LongStream.rangeClosed(1, 1_000_000).sum();
Future<Long> future = executor.submit(c);
try {
    // Blocks 1 second until the Callable has finished
    Long result = future.get(1, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException
        | TimeoutException e) { /** ... */ }
```

Assuming the following list of Callable objects:

```
List<Callable<String>> callables = new ArrayList<>();
callables.add(() -> "Callable 1");
callables.add(() -> "Callable 2");
callables.add(() -> "Callable 3");
```

`invokeAny()` executes the given tasks returning the result of one that has completed successfully. You have no guarantee about which of the Callable's results you'll get, just one of the ones that finish:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    String result = executor.invokeAny(callables);
    System.out.println(result);
} catch(InterruptedException | ExecutionException e) {/** ... */}
```

Sometimes this will print "Callable 1", sometimes "Callable 2", and other times "Callable 3".

`invokeAll()` executes the given tasks returning a list of Future objects that will hold the status and results until all tasks are completed. `Future.isDone()` returns true for each element of the returned list:

```
ExecutorService executor = Executors.newFixedThreadPool(3);
try {
    List<Future<String>> futures = executor.invokeAll(callables);
    for(Future<String> f : futures){
        System.out.format("%s - %s%n", f.get(), f.isDone());
    }
} catch(InterruptedException | ExecutionException e) {/** ... */}
```

One possible output:

```
Callable 1 - true
Callable 2 - true
Callable 3 - true
```

THREADING PROBLEMS

When an application has two or more threads that make it behave in an unexpected way, well, that's a problem.

Generally, the cause of this issue is that threads are executed in parallel in such a way that sometimes they have to compete to access resources and other times, the actions of one thread will cause side-effects over the actions of another one (like modifying or deleting some shared values).

A solution to this problem is the concept of locking, where something like a resource or a block of code is locked with some mechanism in such a way that only one thread at a time can use or access it.

However, if we're not careful, locking can turn in one of the following three problems.

Deadlock

In simple words, a deadlock situation occurs when two or more threads are blocked forever, waiting for each other to acquire/release some resource.

For example, let's say you and I got into a big discussion, you say concurrency is hard to get right and I say it's easy. We're really mad at each other.

After a while, you're ready to say "*All right, it can be easy*" to end the discussion, but only if I say I'm wrong. In the same way, I'm willing to say that I'm wrong, that it is not easy, but only if you say it can be easy.

You hold the *lock* on "*It can be easy*" and you're waiting for me to release the lock on "*It's not easy*."

I'm holding the *lock* on "*It's not easy*" and I'm waiting for you to release the lock on "*It can be easy*."

But neither of us is going to admit that he/she is wrong (*release the lock we have*), because, who does that? So we are going to be waiting for each other (forever?). This is a deadlock situation.

Starvation

Starvation occurs when a thread is constantly waiting for a lock, never able to take it because other threads with higher priority are continually acquiring it.

Suppose you're in the supermarket, waiting in the checkout line. Then, a customer with a VIP membership arrives and is served first without waiting. And then, another VIP customer comes. And then another. And you just wait, forever. That's starvation.

Livelock

A livelock is like a deadlock in the sense that two (or more) threads are blocking each other, but in a livelock, each thread tries to resolve the problem on its own (live) instead of just waiting (dead). They are not blocked, but they are unable to make further progress.

Suppose we are walking in a narrow alley, in opposite directions. When we met, each of us moves aside to let the other pass, but we end up moving to the same side at the same time, repeatedly. That's a livelock.

Chapter TWENTY-SIX

There's another threading problem related with how concurrency works and some consider it the root cause of the previous problems.

Race condition

A race condition is a situation where two threads compete to access or modify the same resource at the same time in a way that causes unexpected results (generally, invalid data).

Let's say there's a movie you and I want to see. Each in our own house, we both go to the cinema's website to buy a ticket. When we check availability, there's only left. We both hurry and click the buy button at the same time. In theory, there can be three outcomes:

- Both of us get to buy the ticket
- Only one of us gets the ticket
- Neither of us gets the ticket (some other person wins it)

That's a race condition (in most race conditions there's a read and then a write).

The definite solution to this problem is never modify a variable (by making them immutable, for example). However, that's not always possible.

Another solution to avoid race conditions is to perform the read and write operations atomically (together in a single step). Another solution (also effective for the other problems) is to ensure the part where the problem happens is executed by only one thread at a time properly.

In the next chapter, we'll see how to implement these solutions with the concurrency API that Java provides.

KEY POINTS

- At a low level, we can create a thread in two ways, either by implementing Runnable or by subclassing Thread and overriding the run() method.
- At a high-level, we use Executors, which use thread pools, which in turn use worker threads.
- One type of thread pool is the fixed thread pool, which has a fixed number of threads running. We can also use single-thread pools.
- ExecutorService has methods to execute thread pools that either take a Runnable or Callable task. A Callable returns a result and throws a checked exception.
- The submit() method returns a Future object that represents the result of the task (if the task is a Runnable, null is returned).
- An executor has to be shutdown to close the pool thread with either shutdown() (gracefully) or shutdownNow() (forcefully).
- A deadlock situation occurs when two or more threads are blocked forever, waiting for each other to acquire/release some resource.
- Starvation happens when a thread is constantly waiting for a lock, never able to take it because other threads with higher priority are continually acquiring it.
- A livelock is like a deadlock in the sense that two (or more) threads are blocking each other, but in a livelock, each thread tries to resolve the problem on its own (live) instead of just waiting (dead).
- A race condition is a situation where two threads compete to access or modify the same resource at the same time in a way that causes unexpected results.

SELF TEST

1. Given:

```
ExecutorService service = Executors.newFixedThreadPool(2);  
Future result = service.submit(() -> 1);
```

Assuming it compiles correctly, what is the type of the lambda expression?

- A. Predicate
- B. Callable
- C. Supplier
- D. Function

2. Which of the following statements are true?

- A. When working with Runnable, you cannot use a Future object.
- B. Executor implements AutoCloseable, so it can be used in a try-with-resources.
- C. A Callable task can be canceled.
- D. Thread pools contain generic threads.

3. Given:

```
Future future = executor.submit(callable);  
future.get(3, TimeUnit.MILLISECONDS);
```

What does the get() method do?

- A. It can return a value, after at most, 3 milliseconds. Otherwise, an exception is thrown.
- B. It can return a value, after at most, 3 milliseconds. Otherwise, null is returned.
- C. It returns a value after exactly 3 milliseconds.
- D. It blocks the program for 3 milliseconds without returning anything.

4. Given:

```
public class Question_26_1 {  
    private static Object A = new Object();  
    private static Object B = new Object();  
    public static void main(String[] args) {  
        new Thread(() -> {  
            acquireLock(A);  
            System.out.println("Just acquired A");  
            acquireLock(B);  
            System.out.println("Just acquired B");  
            releaseLock(B); releaseLock(A);  
        }).start();  
        new Thread(() -> {  
            acquireLock(B);  
            System.out.println("Just acquired B");  
            acquireLock(A);  
            System.out.println("Just acquired A");  
            releaseLock(A); releaseLock(B);  
        }).start();  
    }  
    private static void acquireLock(Object o) {  
        // Code to acquire lock on object o  
    }  
    private static void releaseLock(Object o) {  
        // Code to release lock on object o  
    }  
}
```

What threading problem is more likely to occur in this code?

- A. Race Condition
- B. Deadlock
- C. Livelock
- D. No problem at all

ANSWERS

1. The correct answer is B.

submit() only takes either a Runnable or a Callable instance as an argument. The method signature of the Callable interface doesn't take any argument and returns a value.

2. The correct answers are C and D.

Option A is false. The submit() method accepts a Runnable and returns a Future object.

Option B is false. Executors don't implement AutoCloseable. They are closed by calling the shutdown() or shutdownNow() methods.

Option C is true. A Callable task can be canceled with the Future.cancel() method.

Option D is true. A thread pool contains worked threads, which are generic threads that can execute any Runnable or Callable task.

3. The correct answer is A.

The get(long, TimeUnit) method waits for at most the given time for the task to complete, and then retrieves its result. If there's no result yet, an exception is thrown.

4. The correct answer is B.

One thread locks A and is trying to acquire the lock of B. At the same another thread is doing the opposite, it locks B and is trying to acquire the lock of A, so they are permanently blocked. This is a deadlock situation.

Chapter TWENTY-SEVEN

Concurrency

Exam Objectives

- Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.
- Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList.

SYNCHRONIZATION

In the last chapter, we reviewed some problems that can happen in a concurrent environment and briefly talked about locking.

For example, one solution to avoid a race condition is to ensure that only one thread at a time can access the code that causes the problem; a process known as *synchronizing* that block of code.

Synchronization works with locks. Every object comes with a built-in lock and since there is only lock per object, only one thread can hold that lock at any time. The other threads cannot take the lock until the first thread releases it. Meanwhile, they are blocked.

You define a lock by using the `synchronized` keyword on either a block or a method. That lock is acquired when a thread enters an unoccupied synchronized block or method.

In a synchronized block, you use the `synchronized` keyword followed by either a reference variable:

```
Object o = new Object();

synchronized (o) { // Get the lock of Object o
    // Code guarded by the lock
}
```

Or the `this` keyword:

```
// Get the lock of the object this code belongs to
synchronized (this) {
    // Code guarded by the lock
}
```

Chapter TWENTY-SEVEN

The lock is released when the block ends.

You can also synchronize an entire method:

```
public synchronized void method() {  
    // Code guarded by the lock  
}
```

In this case, the lock belongs to the object on which the method is declared and is released when the method ends.

Notice that synchronizing a method is equivalent to this:

```
public void method() {  
    synchronized (this) {  
        // Code guarded by the lock  
    }  
}
```

Static code can also be synchronized but instead of using this to acquire the lock of an instance of the class, it is acquired on the class object that every class has associated:

```
class MyClass {  
    synchronized static void method() { /** .. */ }  
}
```

Is equivalent to:

```
class MyClass {  
    static void method() {  
        synchronized (MyClass.class) { /** ... */ }  
    }  
}
```

Now, for example, if we execute the following code:

```
public class Test {  
    static int n = 0;  
    static void add() {  
        n++;  
        System.out.println(n);  
    }  
  
    public static void main(String[] args) {  
        ExecutorService service = Executors.newFixedThreadPool(4);  
        Runnable r = () -> add();  
        for(int i = 0; i < 4; i++) {  
            service.execute(r);  
        }  
        service.shutdown();  
    }  
}
```

By looking at one possible output, we can see that we have a race condition problem because four threads are executing the same code:

2
2
3
4

But when we synchronize the `add()` method to ensure only one thread can access it, the problem goes away:

1
2
3
4

ATOMIC CLASSES

The `java.concurrent.atomic` package contains classes to perform atomic operations, which are performed in a single step without the intervention of more than thread.

We have for example:

`AtomicBoolean`, `AtomicInteger`, `AtomicLong`, and
`AtomicReference<V>`

To update a value of the corresponding type (or object reference) atomically.

`AtomicIntegerArray`, `AtomicLongArray`, and
`AtomicReferenceArray<E>`

To update the elements of the corresponding array type (or object reference) atomically.

Each class has methods that perform operations like increments or updates in an atomic way. Take for example the `AtomicInteger` class (the other classes have similar methods):

`int addAndGet(int delta)`

Atomically adds the given value to the current value.

`boolean compareAndSet(int expect, int update)`

Atomically sets the value to the given updated value if the current value equals the expected value.

`int decrementAndGet()`

Atomically decrements by one the current value.

`int get()`

Gets the current value.

```
int getAndAdd(int delta)
```

Atomically adds the given value to the current value.

```
int getAndDecrement()
```

Atomically decrements by one the current value.

```
int getAndIncrement()
```

Atomically increments by one the current value.

```
int getAndSet(int newValue)
```

Atomically sets to the given value and returns the old value.

```
int incrementAndGet()
```

Atomically increments by one the current value.

```
void set(int newValue)
```

Sets to the given value.

The example about incrementing a variable we reviewed before can be rewritten as:

```
public class Test {
    static AtomicInteger n = new AtomicInteger(0);
    static void add() {
        System.out.println(n.incrementAndGet());
    }
    public static void main(String[] args) {
        ...
    }
}
```

When we execute this, we'll get all the numbers but not in order, because atomic classes only ensure data consistency:

1
4
2
3

CONCURRENT COLLECTIONS

The `java.util.concurrent` package provides some thread-safe classes equivalent to the collection classes of `java.util`.

This way, instead of explicitly synchronizing an operation like this:

```
Map<String, Integer> map = new HashMap<>();  
...  
void putIfNew(String key, Integer val) {  
    if(map.get(key) == null) {  
        map.put(key, val);  
    }  
}
```

You can use a `ConcurrentHashMap` like this:

```
Map<String, Integer> map = new ConcurrentHashMap<>();  
...  
map.putIfAbsent(key, val);
```

So when working in concurrent environments, if you're to modify collections, it's better to use the collections of `java.util.concurrent` (besides, they often perform better than plain synchronization).

In fact, if you only need plain `get()` and `put()` methods (when working with a map, for example) you only have to change the implementation:

```
//Map<String, Integer> map = new HashMap<>();  
Map<String, Integer> map = new ConcurrentHashMap<>();  
map.put("one", 1);  
Integer val = map.get("one");
```

Java provides a lot of concurrent collections, but we can classify them by the interface they implement:

- `BlockingQueue` (for queues)
- `BlockingDeque` (for deques)
- `ConcurrentMap` (for maps)
- `ConcurrentNavigableMap` (for navigable maps like `TreeMap`)

Since these interfaces extends from the `java.util` collection interfaces, they inherit the methods we already know (and behave like one of them) so here you'll only find the added methods that support concurrency. And for lists, you can use the class `CopyOnWriteArrayList`.

BlockingQueue

It represents a thread-safe queue that waits (with an optional timeout) for an element to be inserted if the queue is empty or for an element to be removed if the queue is full:

	Blocks	Times Out
Insert	<code>void put(E e)</code>	<code>boolean offer(E e, long timeout, TimeUnit unit)</code>
Remove	<code>E take()</code>	<code>E poll(long timeout, TimeUnit unit)</code>

After the waiting time elapses, `offer()` returns `false` and `poll()` returns `null`.

The main implementations of this interface are:

`ArrayBlockingQueue`. A bounded blocking queue backed by an array.

`DelayQueue`. An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired.

`LinkedBlockingQueue`. An optionally-bounded blocking queue based on linked nodes.

Chapter TWENTY-SEVEN

LinkedTransferQueue. An unbounded TransferQueue based on linked nodes.

PriorityBlockingQueue. An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking and retrieval operations.

SynchronousQueue. A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice-versa. You cannot peek at a synchronous queue because an element is only present when you try to remove it.

BlockingDeque

It represents a thread-safe deque (a double-ended queue, a queue which you can insert and take elements from in both ends). It extends from BlockingQueue and Deque, and also waits (with an optional timeout, after which, it returns false or null too) for an element to be inserted if the deque is empty or for an element to be removed if the deque is full:

First Element (head)		
	Blocks	Times Out
Insert	void putFirst(E e)	boolean offerFirst(E e, long timeout, TimeUnit unit)
Remove	E takeFirst()	E pollFirst(long timeout, TimeUnit unit)
Last Element (tail)		
	Blocks	Times Out
Insert	void putLast(E e)	boolean offerLast(E e, long timeout, TimeUnit unit)
Remove	E takeLast()	E pollLast(long timeout, TimeUnit unit)

The implementation of this interface is LinkedBlockingDeque, an optionally-bounded blocking deque that uses linked nodes.

ConcurrentMap

This interface represents a thread-safe map and it's implemented by the ConcurrentHashMap class.

Here are some of its most important methods:

```
V compute(K key,
BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

Atomically computes the value of a specified key with a BiFunction

```
V computeIfAbsent(K key,
Function<? super K, ? extends V> mappingFunction)
```

Atomically computes the value of a key only if it's not present in the map

```
V computeIfPresent(K key,
BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

Atomically computes the value of a key only if it's present in the map

```
V getOrDefault(Object key, V defaultValue)
```

Returns the value of the key or a default value if the key is not present

```
V merge(K key, V value,
BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

If the specified key is not already associated with a (non-null) value, associates it with the given value. Otherwise, replaces the value with the results of the given remapping function, or removes if null. This is performed atomically.

```
V putIfAbsent(K key, V value)
```

If the key is not present in the map, it's put with the given value atomically

ConcurrentNavigableMap

It represents a thread-safe NavigableMap (like TreeMap) and is implemented by the ConcurrentSkipListMap class, sorted according to the natural ordering of its keys, or by a Comparator provided in its constructor.

Chapter TWENTY-SEVEN

A ConcurrentNavigableMap also implements Map and ConcurrentMap, so it has methods like computeIfAbsent(), getOrDefault(), etc.

CopyOnWriteArrayList

It represents a thread-safe List, similar to an ArrayList, but when it's modified (with methods like add() or set()), a new copy of the underlying array is created (hence the name).

When iterating an ArrayList, methods like remove(), add() or set() can throw a java.util.ConcurrentModificationException. With a CopyOnWriteArrayList, this exception is not thrown because the iterator works with an unmodified copy of the list. But this also means that calling, for example, the remove() method on the Iterator is not supported (it throws an UnsupportedOperationException).

Consider this example, where the last element of the list is changed in every iteration but the original value is still printed inside the iterator:

```
List<Integer> list = new CopyOnWriteArrayList<>();
list.add(10); list.add(20); list.add(30);
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
    int i = it.next();
    System.out.print(i + " ");
    list.set(list.size() -1, i * 10); // No exception thrown
    // it.remove(); throws an exception
}
System.out.println(list);
```

The output:

```
10 20 30 [10, 20, 300]
```

With a `CopyOnWriteArrayList`, there is no lock on reads, so this operation is faster. For that reason, `CopyOnWriteArrayList` is most useful when you have few updates and inserts and many concurrent reads.

There are other classes like `ConcurrentSkipListSet`, which represents a thread-safe `NavigableSet` (like a `TreeSet`).

Or `CopyOnWriteArraySet`, which represents a thread-safe `Set` and internally uses a `CopyOnWriteArrayList` object for all of its operations (so these classes are very similar).

Besides all these classes, for each type of collection, the `Collections` class has methods like the following that take a normal collection and wrap it in a synchronized one:

```
static <T> Collection<T> synchronizedCollection(Collection<T> c)
static <T> List<T>     synchronizedList(List<T> list)
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
static <T> Set<T>      synchronizedSet(Set<T> s)
```

However, it's required to synchronize these collections manually when traversing them via an `Iterator` or `Stream`:

```
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
    Iterator i = c.iterator();
    ...
}
```

Also, they throw an exception if they are modified within an iteration.

Only use these methods if you have to work with an existing collection in a concurrent environment (otherwise, from the beginning use a `java.util.concurrent` collection).

CYCCLICBARRIER

The `synchronized` keyword helps us coordinate access to a shared resource by multiple threads.

But this is very low-level work. I mean, it takes a lot of effort to coordinate complex concurrent tasks. Luckily, Java provides high-level classes to more easily implement these kinds of synchronization tasks.

One of these classes is `CyclicBarrier`. It provides a synchronization point (a barrier point) where a thread may need to wait until all other threads also reach that point.

This class has two constructors:

`CyclicBarrier(int threads)`

Creates a `CyclicBarrier` with the specified number of threads waiting for it.

`CyclicBarrier(int parties, Runnable barrierAction)`

Creates a `CyclicBarrier` with the specified number of threads waiting upon it, and which will execute the given action when the barrier is reached.

The methods:

```
int await() throws InterruptedException, BrokenBarrierException  
int await(long timeout, TimeUnit unit) throws  
    InterruptedException, BrokenBarrierException, TimeoutException
```

Block a thread until all the other threads have called `await()` (reached the barrier), or optionally, until the specified waiting time elapses (when this happens, a `TimeoutException` is thrown).

These methods throw an `InterruptedException` if the current thread was interrupted while waiting and a `BrokenBarrierException` if another thread was interrupted or timed out, or the barrier was reset (with the `reset()` method), or the barrier action failed due to an exception.

Here's an example:

```
public class CyclicBarrierExample {  
    static void checkStep(CyclicBarrier barrier, String step) {  
        // Do something to prepare the step  
        System.out.println(step + " is ready");  
        try {  
            barrier.await(); // Wait the other threads  
        } catch (Exception e) { /** ... */ }  
    }  
    public static void main(String[] args) {  
        String[] steps = {"Read the recipe",  
                          "Gather the ingredients", "Wash hands"};  
        System.out.println("Preparing everything:");  
        Runnable allSet = () ->  
            System.out.println(  
                "Everything's ready. Let's begin.");  
        ExecutorService executor =  
            Executors.newFixedThreadPool(steps.length);  
        CyclicBarrier barrier =  
            new CyclicBarrier(steps.length, allSet);  
        for(String step : steps) {  
            executor.submit(() -> checkStep(barrier, step));  
        }  
        executor.shutdown();  
    }  
}
```

Chapter TWENTY-SEVEN

The output:

Preparing everything:

Gather the ingredients is ready

Read the recipe is ready

Wash hands is ready

Everything's ready. Let's begin.

We have three steps and one thread to process each one. The threads will print the given step and call the `await()` method. When the three threads have called that method, the Runnable represented by the `allSet` variable is executed (the one that prints the "Everything's ready..." message).

As you can see, the steps are not printed in order, but the program cannot proceed until all of them are executed.

Notice that the `CyclicBarrier` is created with the same number of threads. It has to be this way. Otherwise, the program will wait forever.

If the number of threads is less than the `CyclicBarrier` expects, it will wait forever for the missing threads.

To understand why the program will block when the number of threads is greater, you need to know that a `CyclicBarrier` can be reused and how that works.

When all the expected threads call `await()`, the number of waiting threads on the `CyclicBarrier` goes back to zero and it can be used again for a new set of threads. This way, if the `CyclicBarrier` expects three threads but there are four, a cycle of three will be completed and for the next, two will be missing, which will block the program.

KEY POINTS

- Synchronization works with locks. Every object comes with a built-in lock and since there is only lock per object, only one thread can hold that lock at any time. You acquire a lock by using the synchronized keyword in either a block or a method.
- Static code can also be synchronized but instead of using this to acquire the lock of an instance of the class, it is acquired on the class object that every class has associated.
- The `java.concurrent.atomic` package contains classes (like `AtomicInteger`) to perform atomic operations, which are performed in a single step without the intervention of more than thread.
- `BlockingQueue` represents a thread-safe queue and `BlockingDeque` represents a thread-safe deque. Both wait (with an optional timeout) for an element to be inserted if the queue/deque is empty or for an element to be removed if the queue/deque is full.
- `ConcurrentMap` represents a thread-safe map and it's implemented by the `ConcurrentHashMap` class. `ConcurrentNavigableMap` represents a thread-safe `NavigableMap` (like `TreeMap`) and is implemented by the `ConcurrentSkipListMap` class.
- `CopyOnWriteArrayList` represents a thread-safe `List`, similar to an `ArrayList`, but when it's modified (with methods like `add()` or `set()`), a new copy of the underlying array is created (hence the name).
- `CyclicBarrier` that provides a synchronization point (a barrier point) where a thread may need to wait until all other threads reach that point.

SELF TEST

1.Which of the following statements is true?

- A. ConcurrentHashMap has two implementations:
ConcurrentSkipListMap and ConcurrentSkipListSet
- B. The remove() method of CopyOnWriteArrayList creates a new copy
of the underlying array on which this class is based.
- C. A static method can acquire the lock of an instance variable.
- D. Constructors can be synchronized.

2. Which of the following options will correctly increment a value inside
a map in a thread-safe way?

A.

```
Map<String, Integer> map = new ConcurrentHashMap<>();  
int i = map.get(key);  
map.put(key, ++i);
```

B.

```
ConcurrentMap<String, Integer> map = new ConcurrentHashMap<>();  
map.put(key, map.get(key)+1);
```

C.

```
Map<String, Integer> map = new HashMap<>();  
Map<String, Integer> map2 = Collections.synchronizedMap(map);  
int i = map.get(key);  
map.put(key, ++i);
```

D.

```
Map<String, AtomicInteger> map = new ConcurrentHashMap<>();  
map.get(key).incrementAndGet();
```

3. Given:

```
public class Question_27_3 {  
    public static void main(String[] args) throws Exception {  
        BlockingDeque<Integer> deque  
            = new LinkedBlockingDeque<>();  
        deque.offerLast(10, 5, TimeUnit.SECONDS);  
        System.out.print(  
            deque.pollLast(5, TimeUnit.SECONDS) + " ");  
        System.out.print(deque.pollFirst(5, TimeUnit.SECONDS));  
    }  
}
```

What is the result?

- A. The program just prints 10
- B. The program prints 10 and hangs forever
- C. After 5 seconds, the program prints 10 and after another 5 seconds, it prints null
- D. The program prints 10 and 5 seconds later, it prints null

4. Under what circumstances can the await() method of CyclicBarrier throw an exception?

- A. If the thread goes to sleep
- B. When the last thread calls await()
- C. If any thread is interrupted
- D. If the number of threads that call await() is different than the number of threads CyclicBarrier was created with.

ANSWERS

1. The correct answer is B.

Option A is false. `ConcurrentSkipListMap` is the only implementation of `ConcurrentNavigableMap`.

Option B is true. Whenever there's a modification to a `CopyOnWriteArrayList` list, it creates a new copy of the underlying array.

Option C is false. A static method can only use static variables, so it can only acquire the lock of static variables or the lock of variables defined inside the method.

Option D is false. A constructor cannot be synchronized because only one thread has access to the object when is creating it.

2. The correct answer is D.

Only option D performs the operation atomically. The other options perform the operation in two steps, which makes it vulnerable to race conditions.

3. The correct answer is D.

`offerLast()` inserts 10 to the Deque (the timeout value doesn't apply here because the Deque is not full). Then, `pollLast()` removes that value and returns it. At this point, the deque is empty, so `pollFirst()` will wait for 5 seconds to see if another thread inserts another value. Since this didn't happen, `null` is finally returned.

4. The correct answer is C.

The `await()` method throws an `InterruptedException` if the current thread was interrupted while waiting and a `BrokenBarrierException` if another thread was interrupted or timed out.

Chapter TWENTY-EIGHT

Fork/Join Framework

Exam Objectives

- Use parallel Fork/Join Framework.

THE FORK/JOIN FRAMEWORK

The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.

This is done through recursion, where you keep splitting up the task until you meet the *base case*, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.

Splitting up the problem is known as **FORKING** and combining the results is known as **JOINING**.

The main class of the Fork/Join framework is `java.util.concurrent.ForkJoinPool`, which is actually a subclass of `ExecutorService`.

We create a `ForkJoinPool` instance mostly through two constructors:

```
ForkJoinPool()  
ForkJoinPool(int parallelism)
```

The first version is the recommended way. It creates an instance with a number of threads equal to the number of processors of your machine (using the method `Runtime.getRuntime().availableProcessors()`).

The other version allows you to define the number of threads that will be used.

Chapter TWENTY-EIGHT

Just like an ExecutorService executes a task represented by a Runnable or a Callable, in the Fork/Join framework a task is usually represented by a subclass of either:

- RecursiveAction, which is the equivalent of Runnable in the sense that it **DOESN'T** return a value.
- RecursiveTask<V>, which is the equivalent of Callable in the sense that it **DOES** return a value.

Both of them extend from the abstract class `java.util.concurrent.ForkJoinTask`.

However, unlike the worker threads that an ExecutorService uses, the threads of a ForkJoinPool use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work.

To implement this, three methods of your ForkJoinTask-based class are important to the framework:

```
ForkJoinTask<V> fork()  
V join()  
protected abstract void compute() //if you extend RecursiveAction  
protected abstract V compute() //if you extend RecursiveTask
```

And each thread in the ForkJoinPool has a queue of these tasks.

In the beginning, you have a large task. This task is divided into (generally) two smaller tasks recursively until the base case is reached.

Each time a task is divided, you call the `fork()` method to place the first subtask in the current thread's queue, and then you call the `compute()` method on the second subtask (to recursively calculate the result).

This way, the first subtask will wait in the queue to be processed or *stolen* by an idle thread to repeat the process. The second subtask will be processed immediately (also repeating the process).

Of course, you have to divide the task enough times to keep all threads busy (preferably into a number of tasks greater than the number of threads to ensure this).

All right, let's review this. The first subtask is waiting in a queue to be processed, and the second one is processed immediately. So when or how do you get the result of the first subtask?

To get the result of the first subtask you call the `join()` method on this first subtask.

This should be the last step because `join()` will block the program until the result is returned.

This means that the **ORDER** in which you call the methods is **IMPORTANT**.

If you don't call `fork()` before `join()`, there won't be any result to get.

If you call `join()` before `compute()`, the program will perform like if it was executed in one thread and you'll be wasting time, because while the second subtask is recursively calculating the value, the first one can be stolen by another thread to process it. This way, when `join()` is finally called, either the result is ready or you don't have to wait a long time to get it.

But remember, the Fork/Join framework is not for every task. You can use it for any task that can be solved (or algorithm that can be implemented) recursively, but it's best for tasks that can be divided into smaller subtasks **AND** that they can be computed independently (so the order doesn't matter).

Chapter TWENTY-EIGHT

So let's pick a simple example, finding the minimum value of an array. This array can be split up in many subarrays and locate the minimum of each of them. Then, we can find the minimum value between those values.

Let's code this example with a `RecursiveAction` first, to see how this fork/join works. Remember that this class doesn't return a result so we're only going to print the partial results.

Another thing. The most basic scenario we can have (the base case) is when we just have to compare two values. However, having subtasks that are too small won't perform well.

For that reason, when working with fork/join, generally you divide the elements in sets of a certain size (that can be handled by a single thread), for which you solve the problem sequentially.

For this example, let's process five numbers per thread:

```
class FindMinimumAction extends RecursiveAction {  
    // A thread can easily handle, let's say, five elements  
    private static final int SEQUENTIAL_THRESHOLD = 5;  
    // The array with the numbers (we'll pass the same array in  
    // every recursive call to avoid creating a lot of arrays)  
    private int[] data;  
    // The index that tells use where a (sub)task starts  
    private int start;  
    // The index that tells use where a (sub)task ends  
    private int end;  
  
    // Since compute() doesn't take parameters, you have to  
    // pass in the task's constructor the data to work
```

```

public FindMinimumAction(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
}

@Override
protected void compute() {
    int length = end - start;
    if (length <= SEQUENTIAL_THRESHOLD) {      // base case
        int min = computeMinimumDirectly();
        System.out.println("Minimum of this subarray: "+ min);
    } else {                                     // recursive case
        // Calculate new subtasks range
        int mid = start + length / 2;
        FindMinimumAction firstSubtask =
            new FindMinimumAction(data, start, mid);
        FindMinimumAction secondSubtask =
            new FindMinimumAction(data, mid, end);
        firstSubtask.fork(); // queue the first task
        secondSubtask.compute(); // compute the second task
        firstSubtask.join(); // wait for the first task result
    }
}

/** Method that find the minimum value */
private int computeMinimumDirectly() {
    int min = Integer.MAX_VALUE;
    for (int i = start; i < end; i++) {
        if (data[i] < min) {
            min = data[i];
        }
    }
    return min;
}
}

```

Chapter TWENTY-EIGHT

The `compute()` method defines the base case, when the (sub)array has five elements or less, in which case the minimum is found sequentially. Otherwise, the array is split into two subarrays recursively until the condition of the base case is fulfilled.

Dividing the tasks may not always result in evenly distributed subtasks. But to keep this simple, let's try the class with twenty elements, which is very likely to be split up into four sets:

```
public static void main(String[] args) {
    int[] data = new int[20];
    Random random = new Random();
    for (int i = 0; i < data.length; i++) {
        data[i] = random.nextInt(1000);
        System.out.print(data[i] + " ");
        // Let's print each subarray in a line
        if( (i+1) % SEQUENTIAL_THRESHOLD == 0 ) {
            System.out.println();
        }
    }
    ForkJoinPool pool = new ForkJoinPool();
    FindMinimumAction task =
        new FindMinimumAction(data, 0, data.length);
    pool.invoke(task);
}
```

A possible output can be:

```
109 411 348 938 776
188 42 28 818 825
642 454 431 742 463
33 832 705 910 456
```

```
Minimum of this subarray: 33
Minimum of this subarray: 28
Minimum of this subarray: 431
Minimum of this subarray: 109
```

Notice that we didn't need to shut down the pool explicitly. When the program exits, the pool is shut down, so it can be reused.

We also have the `invokeAll()` method, that doesn't return a value but does something equivalent to the call of `fork()`, `compute()`, and `join()` methods. So instead of having something like:

```
...
FindMinimumAction firstSubtask =
    new FindMinimumAction(data, start, mid);
FindMinimumAction secondSubtask =
    new FindMinimumAction(data, mid, end);
firstSubtask.fork(); // queue the first task
secondSubtask.compute(); // compute the second task
firstSubtask.join(); // wait for the first task result
...
```

We can simply have:

```
...
FindMinimumAction firstSubtask =
    new FindMinimumAction(data, start, mid);
FindMinimumAction secondSubtask =
    new FindMinimumAction(data, mid, end);
invokeAll(firstSubtask, secondSubtask);
...
```

Now, let's change this example to use a `RecursiveTask` so we can return the minimum value of all.

Chapter TWENTY-EIGHT

Actually, the only changes we need to do are in the `compute()` method:

```
class FindMinimumTask extends RecursiveTask<Integer> {  
    // ...  
  
    @Override  
  
    protected Integer compute() { //Return type matches the generic  
        int length = end - start;  
        if (length <= SEQUENTIAL_THRESHOLD) { // base case  
            return min = computeMinimumDirectly();  
        } else { // recursive case  
            // Calculate new subtasks range  
            int mid = start + length / 2;  
            FindMinimumAction firstSubtask =  
                new FindMinimumAction(data, start, mid);  
            FindMinimumAction secondSubtask =  
                new FindMinimumAction(data, mid, end);  
            firstSubtask.fork(); // queue the first task  
            // Return the minimum of all subtasks  
            return Math.min(firstSubtask.compute(),  
                           secondSubtask.join());  
        }  
    }  
    // ...  
}
```

In the `main()` method, the only changes are printing the value that `pool.invoke(task)` returns and a little formatting to the output, which can be:

```
Array values:  
819 997 124 425 669 657 487 447 386 979 31 748 194 644 893 209 913 810 142 565  
Minimum value: 31
```

KEY POINTS

- The Fork/Join framework is designed to work with large tasks that can be split up into smaller tasks.
- This is done through recursion, where you keep splitting up the task until you meet the *base case*, a task so simple that can be solved directly, and then combining all the partial results to compute the final result.
- Splitting up the problem is known as **FORKING** and combining the results is known as **JOINING**.
- The main class of the Fork/Join framework is `java.util.concurrent.ForkJoinPool`, which is actually a subclass of `ExecutorService`.
- Just like an `ExecutorService` executes a task represented by a `Runnable` or a `Callable`, in the Fork/Join framework a task is represented by a subclass of either `RecursiveAction` (that **DOESN'T** return a value) or `RecursiveTask<V>` (that **DOES** return a value).
- However, unlike the worker threads that an `ExecutorService` uses, the threads of a `ForkJoinPool` use a work-stealing algorithm, which means that when a thread is free, it **STEALS** the pending work of other threads that are still busy doing other work.
- A `ForkJoinTask` object has three main methods, `fork()`, `join()`, and `compute()`. The **ORDER** in which you call the methods is **IMPORTANT**.
- You must first call `fork()` to queue the first subtask, then `compute()` on the second subtask to process it recursively, and then `join()` to get the result of the first subtask.

SELF TEST

1. What of the following statements is true?
 - A. RecursiveAction is a subclass of ForkJoinPool.
 - B. When working with the Fork/Join framework, by default, one thread per CPU is created.
 - C. You need to shut down a ForkJoinPool explicitly.
 - D. fork() blocks the program until the result is ready.

2. Which of the following is the right order to call the methods of a ForkJoinTask?
 - A. compute(), fork(), join()
 - B. fork(), compute(), join()
 - C. join(), fork(), compute()
 - D. fork(), join(), compute()

3. When using a RecursiveTask, which of the following statements is true?
 - A. You can use the invokeAll() method instead of the fork()/join()/compute() methods.
 - B. You can use ExecutorService directly with this class.
 - C. An action is triggered when the task is completed.
 - D. ForkJoinTask.invoke() returns the same type as the generic type of RecursiveTask.

4. Given:

```
public class Question_28_4 extends RecursiveTask<Integer> {  
    private int n;  
  
    Question_28_4(int n) {  
        this.n = n;  
    }  
  
    public Integer compute() {  
        if (n <= 1) {  
            return n;  
        }  
        Question_28_4 t1 = new Question_28_4(n - 1);  
        Question_28_4 t2 = new Question_28_4(n - 2);  
        t1.fork();  
        return t2.compute() + t1.join();  
    }  
}
```

What is not right about this implementation of the Fork/Join framework?

- A. Everything is right, it's a perfect implementation of the Fork/Join framework.
- B. The order of the fork(), join(), compute() methods is not right.
- C. This implementation is very inefficient, the subtasks will be very small.
- D. It doesn't compile.

ANSWERS

1. The correct answer is B.

Option A is false. RecursiveAction is a subclass of ForkJoinTask.

Option B is true. By default, one thread per CPU is created.

Option C is false. You don't need to shut down a ForkJoinPool explicitly. It's closed when the program ends.

Option D is false. fork() doesn't block the program, just add a task to the thread's queue.

2. The correct answer is B.

If you don't call fork() before join(), there won't be any result to get.

If you call join() before compute(), the program will perform like if it was executed in one thread because of the blocking produced by join().

3. The correct answer is D.

ForkJoinTask.invoke() returns the same type as the generic type of RecursiveTask is the only true statement. You can't use invokeAll() because this method doesn't return a result. You can't use an ExecutorService because it uses a different type of threads and you can't have an action triggered when the task is completed.

4. The correct answer is C.

The program tries to get the value of the nth Fibonacci number with the Fork/Join framework, but it does it in a very inefficient way because the subtasks will be very small.

Remember that Fork/Join is not the best choice for everything. Actually, solving this problem using this recursive algorithm is not the best way, but implementing it with an algorithm where Karatsuba multiplication and parallelism can be used will certainly provide much better results.

Part NINE

JDBC and Localization

Chapter TWENTY-NINE

JDBC API

Exam Objectives

- Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations.
- Identify the components required to connect to a database using the DriverManager class including the JDBC URL.
- 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.

INTRODUCTION

The Java Database Connectivity API (or JDBC) defines how we can access a relational database. You can find its classes and interfaces in the `java.sql` package.

The latest version of this API, included in Java 8, is JDBC 4.2.

The main benefit of using JDBC is that it hides the implementation details between databases by using a set of interfaces so you don't have to write different code for accessing different databases.

This chapter assumes you know how to use SQL to select, insert, update, and delete records from a database entity.

Also, for this chapter, it's recommended (although not required) to have access to a database to do some practices and test some concepts, however, it won't teach how to install or use one.

If you don't have a database installed (or access to one), perhaps the easiest way is to use JavaDB, a database that is shipped with Java that can be used as an embedded or network server database. You can find it in the `db` directory of the JDK installation.

For the exam, you are expected to know the main interfaces of JDBC, how to connect to a database, read results of a query, and perform some operations such as inserts or updates.

JDBC INTERFACES

When using JDBC, you work with interfaces rather than implementations. Those implementations come from a JDBC driver.

A driver is just a JAR file with classes that know how to talk to a specific database. For example, for MySQL, there's a `mysql-connector-java-XXX.jar`, where XXX is the version of the database.

But you don't need to know how the classes inside the driver are implemented or named, you just have to know the four main interfaces that they have to implement:

- `java.sql.Driver`
Every JDBC driver must implement this interface to know how to connect to the database.
- `java.sql.Connection`
The implementation provides methods for getting information about the database, create statements, and managing connections.
- `java.sql.Statement`
The implementation is used to execute SQL statements and to return results.
- `java.sql.ResultSet`
The implementation is used for retrieving and updating the results of a query.

In addition to these interfaces, an important class is `java.sql.DriverManager`, which keeps track of the loaded JDBC drivers and gets the actual connection to the database.

CONNECTING TO A DATABASE

First, to work with a database, you need to connect to it. This is done using the `DriverManager` class.

Before attempting to establish a connection, the `DriverManager` class has to load and register any implementations of `java.sql.Driver` (which know how to establish the connection).

Maybe you've seen in some program a line like this:

```
Class.forName("com.database.Driver")
```

This loads the `Driver` implementation so `DriverManager` can register the driver. However, this is no longer required since JDBC 4.0, because `DriverManager` automatically loads any JDBC 4.0 driver in the classpath.

Once the drivers are loaded, you can connect to a database with the static method `DriverManager.getConnection()`:

```
Connection getConnection(String url)
Connection getConnection(String url, Properties info)
Connection getConnection(String url, String user, String passw)
```

This method will look through the registered drivers to see if it can find one that can make a connection using the given URL.

The URL varies depending on the database used. However, they have three parts in common:

JDBC URL FORMAT

PROTOCOL
(ALWAYS THE SAME)

jdbc:mysql:

SUBPROTOCOL
(MOST OF THE TIME THE NAME
OF THE DATABASE/TYPE OF THE
DRIVER)

DATABASE SPECIFIC CONNECTION PROPERTIES

(MOST OF THE TIME THE LOCATION OF THE DATABASE WITH FORMAT:
//SERVER:HOST/DATABASE_NAME)

//host:3306/db

Chapter TWENTY-NINE

- The first part is always the same, *jdbc*.
- The second part, most of the time, is the name of the database and/or the type of the driver, like *mysql*, *postgresql*, *oracle:thin*.
- The last part varies according to the database, but most of the time, it contains the name (or IP) of the host, the port and the name of the database you're connecting to, like `//192.168.0.1:3306/db1`.

Here are some URL examples:

```
jdbc:postgresql://localhost/test  
jdbc:sqlserver://localhost\SQLEXPRESS;dbname=db  
jdbc:derby:db;create=true
```

We can connect to the database by calling this method to get a `Connection` object:

```
Connection con =  
    DriverManager.getConnection("jdbc:mysql://localhost/db");
```

If you need to pass a user or password for authentication, we can use:

```
Properties props = new Properties();  
props.put("user", "db_user");  
props.put("password", "db_p4assw0rd");  
Connection con =  
    DriverManager.getConnection("jdbc:mysql://localhost/db",  
                               props);
```

Or:

```
Connection con =  
    DriverManager.getConnection("jdbc:mysql://localhost/db",  
                               "db_user",  
                               "db_p4assw0rd");
```

Before your program finishes, you need to close the connection (otherwise you can run out of connections).

Luckily, `Connection` implements `AutoCloseable`, which means you can use a `try-with-resources` to automatically close the connection:

```
String url = "jdbc:mysql://localhost/db";
String user = "db_user";
String passw = " db_p4assw0rd";

try(Connection con =
        DriverManager.getConnection(url, user, passw)) {
    // Database operations
} catch(SQLException e) {
    System.out.format("%d-%s-%s",
                      e.getErrorCode(),
                      e.getSQLState(),
                      e.getMessage());
}
```

An `SQLException` is thrown whenever there's an error in JDBC (like when the driver to connect to the database is not found in the classpath) and many methods of the JDBC API throw it, so it has to be caught (or retrieved in the case of suppressed exceptions).

As this exception is used for many errors, to know what went wrong, you have to use `getMessage()`, which returns a description of the error, `getSQLState()` that returns a standard error code, or `getErrorCode()`, which returns the database-specific code for the error.

Now that we have a `Connection` object, we can execute some SQL statements.

EXECUTING QUERIES

You need a Statement object to execute queries and perform database operations.

There are three Statement interfaces:

- `java.sql.Statement`
Represents simple SQL statements to the database, without parameters.
- `java.sql.PreparedStatement`
Represents precompiled SQL statements to execute statements multiple times efficiently. It can accept input parameters.
- `java.sql.CallableStatement`
Used to execute stored procedures. It can accept input as well as output parameters.

In practice, PreparedStatement is the one often used, but for the exam, you only need to know Statement.

You can get a Statement from a Connection object using the `createStatement()` method:

```
Statement createStatement()  
Statement createStatement(int resultSetType,  
                           int resultSetConcurrency)
```

When creating a Statement, you can define the type of the result set and its concurrency mode.

There are three types of result sets:

- `ResultSet.TYPE_FORWARD_ONLY`
This is the default type. When specified, you can only go once through the results and in the order they were retrieved.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
When specified, you can go both forward and backward through the results and to a particular position in the result set.
- `ResultSet.TYPE_SCROLL_SENSITIVE`
When specified, you can also go forward, backward and to a particular position in the result set, but you will always see the latest changes to the data while using it, in contrast with `TYPE_SCROLL_INSENSITIVE`, which it's not "sensitive" to changes to the data.

In practice, most drivers don't support `TYPE_SCROLL_SENSITIVE`. If you ask for it and is not available, you will get either `TYPE_FORWARD_ONLY` or (more likely) `TYPE_SCROLL_INSENSITIVE`.

There are two concurrency modes:

- `ResultSet.CONCUR_READ_ONLY`
This is the default mode. When specified, you can't update (using an `INSERT`, `UPDATE`, or `DELETE` statement) a result set.
- `ResultSet.CONCUR_UPDATABLE`
It indicates that the result set can be updated.

If you ask for a `CONCUR_UPDATABLE` mode and your driver doesn't support it, you can get a `CONCUR_READ_ONLY` mode.

Most of the time, the default values are used:

```
Statement stmt = con.createStatement();
```

Chapter TWENTY-NINE

Now that we have a Statement object, we have three methods at our disposal to execute SQL commands:

Method	Supported SQL statements	Return type
execute()	SELECT INSERT UPDATE DELETE CREATE	boolean (true for SELECT, false for the rest)
executeQuery()	SELECT	ResultSet
executeUpdate()	INSERT UPDATE DELETE CREATE	Number of affected rows (zero for CREATE)

A Statement object has to be closed, but like Connection, it implements AutoCloseable so it can be used with a try-with-resources also:

```
try(Connection con =
        DriverManager.getConnection(url, user, passw);
Statement stmt = con.createStatement()) {
    boolean hasResults = stmt.execute("SELECT * FROM user");
    if(hasResults) {
        // To retrieve the object with the results
        ResultSet rs = stmt.getResultSet()
    } else {
        // To get the number of affected rows
        int affectedRows = stmt.getUpdateCount();
    }
}
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM user");

stmt.executeUpdate("INSERT INTO user(id, name) "
    + "VALUES(1, 'George')"); // Returns 1
stmt.executeUpdate("UPDATE user SET name='Joe' "
    + "WHERE id = 1"); // Returns 1
stmt.executeUpdate("DELETE FROM user "
    + "WHERE id = 1"); // Returns 1

} catch(SQLException e) {
    e.printStackTrace();
}
```

Choose the correct `execute()` method based on the type of SQL statement you're using, because if you use the wrong method, an `SQLException` will be thrown.

When you use a `SELECT` statement, you can read the result through a `ResultSet` object, which we'll review next.

READING RESULTS

A `ResultSet` object is used to read the results of a query in a tabular format (rows containing the columns specified).

This object keeps a *cursor* that points to the current row and you can only read one row at a time.

In the beginning, the cursor is just before the first row. Calling the `next()` method will advance the cursor one position and return `true` if there's data, or `false` if there isn't any. This way, you can iterate in a loop over the entire result set.

Like `Connection` and `Statement`, a `ResultSet` object needs to be closed too and also implements `AutoCloseable`:

```
try(Connection con =
        DriverManager.getConnection(url, user, passw);
    Statement stmt = con.createStatement();
    ResultSet rs =
        stmt.executeQuery("SELECT * FROM user")) {
    while(rs.next()) {
        // Read row
    }
} catch(SQLException e) {
    e.printStackTrace();
}
```

It's a good practice to close these resources in this way, but it's not required. Here are the rules for closing JDBC resources:

- The ResultSet object is closed first, then the Statement object, then the Connection object.
- A ResultSet is automatically closed when another ResultSet is executed from the same Statement object.
- Closing a Statement also closes the ResultSet.
- Closing a Connection also closes the Statement and ResultSet objects.

If the query doesn't return results, a ResultSet object is still returned (although next() will return false at the first call).

Remember, next() doesn't just tell if there are more records to process, it also advances the cursor to the next row.

This means that even when you want to access only the first row of the result, you still have to call next() (preferably with an if statement, because if there are no elements, an SQLException is thrown if you try to access any):

```
if(rs.next) {  
    // Access the first element if there's any  
}
```

Now, to actually get the data, ResultSet has getter methods for a lot of data types, for example:

- getInt() returns an int
- getLong() returns a Long
- getString() returns a String
- getObject() returns an Object
- getDate() returns a java.sql.Date
- getTime() returns a java.sql.Time
- getTimestamp() returns java.sql.Timestamp

Chapter TWENTY-NINE

For each method, there are two versions:

- One that takes a `String` that represents the name of the column (this is **NOT** case-sensitive).
- Another one that takes an `int` that represents the column index according to the order declared in the `SELECT` clause. The first column starts with **1**, not **0**.

For example:

```
Result rs = stmt.executeQuery("SELECT id, name FROM user");
while(rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    // Do something
}
```

It's equivalent to:

```
Result rs = stmt.executeQuery("SELECT id, name FROM user");
while(rs.next()) {
    int id = rs.getInt(1);
    String name = rs.getString(2);
    // Do something
}
```

If we reference a non-existent column (either by index or name), an `SQLException` will be thrown.

Notice that methods `getDate()`, `getTime()`, and `getTimestamp()` don't return standard date or time objects, so they might need to be converted, for example:

```
Result rs = stmt.executeQuery("SELECT insertion_date FROM user");
while(rs.next()) {
    // Getting the date part
    java.sql.Date sqlDate = rs.getDate(1);
    // Getting the time part
    java.sql.Time sqlTime = rs.getTime(1);
    // Getting both, the date and time part
    java.sql.Timestamp sqlTimestamp = rs.getTimestamp(1);

    // Converting date
    LocalDate localDate = sqlDate.toLocalDate();
    // Converting time
    LocalTime localTime = sqlTime.toLocalTime();
    // Converting timestamp
    Instant instant = sqlTimestamp.toInstant();
    LocalDateTime localDateTime = sqlTimestamp.toLocalDateTime();
}
```

So that's how you work with TYPE_FORWARD_ONLY result sets.

When working with scrollable result sets (TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE), we have a lot of options to move the cursor.

Here's the complete list of methods for moving the cursor (just remember that all methods, except next(), need a scrollable result set):

Chapter TWENTY-NINE

Method	Description
<code>boolean absolute(int row)</code>	Moves the cursor to the given row number in the result set, counting from the beginning (if the argument is positive) or the end (if negative). If the argument is zero, the cursor is moved before the first row. It returns true if the cursor is moved to a valid position or false if the cursor is before the first row or after the last row.
<code>void afterLast()</code>	Moves the cursor after the last row.
<code>void beforeFirst()</code>	Moves the cursor before the first row.
<code>boolean first()</code>	Moves the cursor to the first row. It returns true if the cursor is on a valid row or false if there are no rows in the result set.
<code>boolean last()</code>	Moves the cursor to the last row. Returns true if the cursor is on a valid row or false if there are no rows in the result set.
<code>boolean next()</code>	Moves the cursor to the next row. It returns true if the new current row is valid or false if there are no more rows.
<code>boolean previous()</code>	Moves the cursor to the previous row. It returns true if the new current row is valid or false if the cursor is before the first row.
<code>boolean relative(int rows)</code>	Moves the cursor a relative number of rows, either positive or negative. Moving beyond the first/last row in the result set positions the cursor before/after the first/last row. It returns true if the cursor is on a valid row, false otherwise.

So considering this table:

ID	NAME	INSERTION_DATE
1	THOMAS	2016 / 03 / 01
2	LAURA	2016 / 03 / 01
3	MAX	2016 / 03 / 01
4	KIM	2016 / 03 / 01

The following program shows some of these methods. Try to follow it:

```

try(Connection con =
        DriverManager.getConnection(url, user, passw);
    Statement stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
    ResultSet rs = stmt.executeQuery("SELECT * FROM user")) {
    System.out.println(rs.absolute(3)); // true
    System.out.println(rs.getInt(1)); // 3
    System.out.println(rs.absolute(-3)); // true
    System.out.println(rs.getInt(1)); // 2
    System.out.println(rs.absolute(0)); // false
    System.out.println(rs.next()); // true
    System.out.println(rs.getInt(1)); // 1
    System.out.println(rs.previous()); // false
    System.out.println(rs.relative(2)); // true
    System.out.println(rs.getInt(1)); // 2
    System.out.println(rs.relative(0)); // true
    System.out.println(rs.getInt(1)); // 2
    System.out.println(rs.relative(10)); // false
    System.out.println(rs.previous()); // true
    System.out.println(rs.getInt(1)); // 4
} catch(SQLException e) {
    e.printStackTrace();
}

```

KEY POINTS

- The Java Database Connectivity API (or JDBC) defines how we can access a relational database. You can find its classes and interfaces in the `java.sql` package.
- When using JDBC, you work with interfaces rather than implementations. Those implementations come from a JDBC driver. The four main interfaces to implement are:
 - `java.sql.Driver`
 - `java.sql.Connection`
 - `java.sql.Statement`
 - `java.sql.ResultSet`
- In addition to these interfaces, an important class is `java.sql.DriverManager`, which keeps track of the loaded JDBC drivers and gets the actual connection to the database.
- Once the drivers are loaded, you can connect to a database with the static method `DriverManager.getConnection(String url)`.
- The URL varies depending on the database used. However, they have three parts in common:
 - The first part is always the same, `jdbc`.
 - The second part, most of the time, is the name of the database and/or the type of the driver, like `mysql`, `postgresql`, `oracle:thin`.
 - The last part varies according to the database, but most of the time, it contains the name (or IP) of the host, the port and the name of the database you're connecting to, like `//192.168.0.1:3306/db1`.
- Once we have a `Connection` object, we can execute some SQL statements by getting a `Statement` object.
- You can get a `Statement` from a `Connection` object using the `createStatement()` method.

- When creating a Statement, you can define the type of the result set and its concurrency mode.
- There are three types of result sets, `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, `ResultSet.TYPE_SCROLL_SENSITIVE`.
- There are two concurrency modes, `ResultSet.CONCUR_READ_ONLY`, `ResultSet.CONCUR_UPDATABLE`.
- We have three methods in Statement to execute SQL commands:
 - `boolean execute(String sql)`
 - `ResultSet executeQuery(String sql)`
 - `int executeUpdate(String sql)`
- A ResultSet object is used to read the results of a query in a tabular format (rows containing the columns specified). This object keeps a *cursor* that points to the current row and you can only read one row at a time.
- The rules for closing JDBC resources are:
 - The ResultSet object is closed first, then the Statement object, then the Connection object.
 - A ResultSet is automatically closed when another ResultSet is executed from the same Statement object.
 - Closing a Statement also closes the ResultSet.
 - Closing a Connection also closes the Statement and ResultSet objects.
- To actually get the data, ResultSet has getter methods for a lot of data types. For each method, there are two versions, one that takes the name of the column and another that takes its column index.
- When working with scrollable result sets (TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE), we have a lot of options to move the cursor, like `absolute(int row)`, `first()`, `previous()`, and `relative(int rows)`.

SELF TEST

1. Given:

```
public class Question_29_1 {  
    public static void main(String[] args) {  
        try(Connection con =  
            DriverManager.getConnection("jdbc:mysql://localhost");  
            Statement stmt = con.createStatement();  
            ResultSet rs =  
                stmt.executeQuery("SELECT * FROM user")) {  
            while(rs.next()) {  
                System.out.println(rs.getObject(1));  
            }  
        } catch(SQLException e) {  
            System.out.println("SQLException");  
        }  
    }  
}
```

What is the result if the query doesn't return any result?

- A. SQLException
- B. Nothing is printed
- C. Compilation fails
- D. An uncaught exception occurs at runtime

2. Which of the following is equivalent to `rs.absolute(-1)`?

- A. `rs.absolute(1);`
- B. `rs.afterLast();`
- C. `rs.last();`
- D. `rs.relative(-1);`

3. Which of the following options shows the correct order to close database resources?

- A. ResultSet, Connection, Statement
- B. Statement, ResultSet, Connection
- C. Connection, Statement, ResultSet
- D. ResultSet, Statement, Connection

4. Given:

```
public class Question_29_4 {  
    public static void main(String[] args) {  
        try(Connection con =  
                DriverManager.getConnection("jdbc:mysql://localhost");  
            Statement stmt = con.createStatement()) {  
            System.out.println(  
                stmt.execute("INSERT INTO user VALUES(1, 'Joe')"));  
        } catch(SQLException e) { /* ... */ }  
    }  
}
```

What is the result?

- A. true
- B. false
- C. 1
- D. An exception occurs at runtime

5. Which of the following can be a valid way to get the value of the first column of a row?

- A. rs.getInteger(1);
- B. rb.getString("0");
- C. rb.getObject(0);
- D. rb.getBoolean(1);

ANSWERS

1. The correct answer is B.

If the result set is empty, next() returns false, so nothing is printed.

2. The correct answer is C.

If the argument of absolute() is negative, the cursor is positioned starting from the end of the result set. Since last() moves the cursor to the last element of the result set, this and absolute(-1) are equivalent.

3. The correct answer is D.

The first object you have to close is ResultSet, then Statement, and finally Connection.

4. The correct answer is B.

execute() returns a boolean value, true for SELECT and false for other statements.

5. The correct answer is D.

Option A is invalid. The method getInteger() doesn't exist (the correct one is getInt()).

Option B is invalid. The string argument must be the name of the column.

Option C is invalid. In JDBC, indexes start at 1.

Option D is valid. rb.getBoolean(1) can be a valid way to get the value of the first column of a row.

Chapter THIRTY

Localization

Exam Objectives

- Read and set the locale by using the Locale object.
- Create and read a Properties file.
- Build a resource bundle for each locale and load a resource bundle in an application.

LOCALIZATION

Localization (abbreviated as l10N because of the number of characters between the first and the last letter) is the mechanism by which an application is adapted to a specific language and region.

It's related to the concept of *internationalization* (abbreviated as i18n for the same reason as localization), which is about designing an application that can handle different languages and regions.

The most common things that can be customized by language and/or region are messages, dates, and numbers.

In Java, it all starts with one class, `java.util.Locale`.

The `Locale` class basically represents a language and a country although, to be precise, a locale can have the following information:

- An ISO 639 alpha-2 or alpha-3 language code, like `ja` (Japanese)
- An ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code, like `JP` (Japan)
- A variant name, usually empty but can be any string.
- An ISO 15924 alpha-4 script code, like `Latn` (Latin)
- A set of extensions represented by single characters, like `u`.

But most of the time, we just work with languages and countries.

A LOCALE REPRESENTATION

THE REQUIRED PART

fr

LANGUAGE
(NOTICE THE LOWER CASE)

THE OPTIONAL PART

CA

JUST AN
 underscore

COUNTRY
(NOTICE THE UPPER CASE)

Chapter THIRTY

You can get the default locale of your machine with:

```
Locale locale = Locale.getDefault();
```

And get information like:

```
System.out.println("Country Code: "
    + locale.getCountry());
System.out.println("Country Name: "
    + locale.getDisplayCountry());
System.out.println("Language Code: "
    + locale.getLanguage());
System.out.println("Language Name: "
    + locale.getDisplayLanguage());
```

The output (notice how the actual names are localized in spanish):

```
Country Code: MX
Country Name: México
Language Code: es
Language Name: español
```

You can also get all the locales supported by Java:

```
Locale [] locales = Locale.getAvailableLocales();
Arrays.stream(locales)
    .forEach(System.out::println);
```

This would output around 160 locales in the form of language[_country], for example:

```
it
pt_BR
ro_RO
```

SETTING THE LOCALE

There are three different ways to create a Locale instance:

1. Using a constructor

There are three constructors:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String variant)
```

For example:

```
Locale chinese = new Locale("zh");
Locale CHINA = new Locale("zh", "CN");
```

2. Using the forLanguageTag(String) factory method

This method expects a language code, for example:

```
Locale german = Locale.forLanguageTag("de");
```

3. Using Locale.Builder

You can set the properties you need and build the object at the end, for example:

```
Locale japan = new Locale.Builder()
    .setRegion("JP")
    .setLanguage("jp")
    .build();
```

Chapter THIRTY

Passing an invalid argument to any of the above three methods will not throw an exception, it will just create an object with invalid options that will make your program behave incorrectly:

```
Locale badLocale = new Locale("a", "A"); // No error
System.out.println(badLocale); // It prints a_A
```

So it's good to know that `Locale` class also provides predefined constants for some common languages and countries, for example:

```
Locale.GERMAN
Locale.KOREAN
Locale.UK
Locale.ITALY
```

For the exam, you don't have to know all these constants, or some obscure language and country codes, just that there are four ways to start working with locales.

Once you have a `Locale` object, you can change the locale of your program with the `setDefault(Locale)` method:

```
System.out.println(Locale.getDefault()); // Prints let's say en_GB
Locale.setDefault(new Locale("en", "US"));
System.out.println(Locale.getDefault()); // Now prints en_US
```

PROPERTY FILES

Property files define strings in key/value pairs separated by lines.

There are some rules, like:

- Spaces at the beginning of the line (if any) are ignored.
- Any line that starts with # or ! will be treated as a comment.
- You can break a line for readability purposes with a backslash.

For example, we can have the following file:

Messages.properties

```
# Video related messages
video.added = The video has been added
video.deleted = The video has been deleted
```

To read it, you create an instance of `java.util.Properties` and load it with either a `java.io.Reader` or a `java.io.InputStream`, for example:

```
Properties prop = new Properties();
try (InputStream is = getClassLoader()
        .getResourceAsStream("Messages.properties")) {
    prop.load(is); // Load properties
    // Prints The video has been added
    System.out.println(prop.getProperty("video.added"));
    // It prints a default value if the key is not found
    System.out.println(prop.getProperty("video.add", "default"));
    // Get all properties keys
    Enumeration<?> e = prop.propertyNames();
} catch (IOException e) {
    e.printStackTrace();
}
```

RESOURCE BUNDLES

To localize an application, we have Resource Bundles, which define a set of keys with localized values. Resource Bundles can be property files or classes.

To support this, we have an abstract class `java.util.ResourceBundle` with two subclasses:

java.util.PropertyResourceBundle

Each locale is represented by a property file. Keys and values are of type `String`.

java.util.ListResourceBundle

Each locale is represented by a subclass of this class that overrides the method `Object[][][] getContents()`. The returned array represents the keys and values. Keys must be of type `String`, but values can be any object.

In both methods, the name (of the file or the class) follows a convention which allows Java to search for resource bundles and match them to their corresponding locales.

That name convention is on the next page.

Only the resource bundle name is required (and the name of the package if it is not the default one).

package.Bundle_language_country_variant

FOR EXAMPLE:

com.example.MyBundle_fr_FR

Chapter THIRTY

For example, we can have bundles with the following names (assuming we're working with property files, although it's the same with classes):

```
MyBundle.properties  
MyBundle_en.properties  
MyBundle_en_NZ.properties  
MyBundle_en_US.properties
```

To determine which bundle belongs to a particular locale, Java tries to find the most specific bundle that matches the properties of the locale.

This means that:

1. Java first searches for a bundle whose name matches the complete locale:

```
package.bundle_language_country_variant
```

2. If it cannot find one, it drops the last component of the name and repeats the search:

```
package.bundle_language_country
```

3. If it cannot find one, again, it drops the last component of the name and repeats the search:

```
package.bundle_language
```

4. If still cannot find one, the last component is dropped again, leaving just the name of the bundle:

```
package.bundle
```

If nothing is found, a `MissingBundleException` is thrown.

If a class and a property file share the same name, Java gives priority to the class.

But there's another important point.

In your program, you can use the keys of the matching resource bundle and **ANY** of its **PARENTS**.

The parents of a resource bundle are the ones with the same name but fewer components. For example, the parents of MyBundle_es_ES are:

MyBundle_es
MyBundle

For example, let's assume the default locale en_US, and that your program is using those and other property files, all in the default package, with the values::

```
MyBundle_EN.properties
s = buddy

MyBundle_es_ES.properties
s = tío

MyBundle_es.properties
s = amigo

MyBundle.properties
hi = Hola
```

We can create a resource bundle like this:

```
public class Test {
    public static void main(String[] args) {
        Locale spain = new Locale("es", "ES");
        Locale spanish = new Locale("es");

        ResourceBundle rb =
```

Chapter THIRTY

```
    ResourceBundle.  
    getBundle("MyBundle", spain);  
    System.out.format("%s %s\n",  
                      rb.getString("hi"), rb.getString("s"));  
  
    rb = ResourceBundle.getBundle("MyBundle", spanish);  
    System.out.format("%s %s\n",  
                      rb.getString("hi"), rb.getString("s"));  
}  
}
```

The output:

```
Hola tío  
Hola amigo
```

As you can see, each locale picks different values for key s, but they both use the same for hi since this key is defined in their parent.

If you don't specify a locale, the ResourceBundle class will use the default locale of your system:

```
ResourceBundle rb = ResourceBundle.getBundle("MyBundle");  
System.out.format("%s %s\n",  
                  rb.getString("hi"), rb.getString("s"));
```

Since we assume that the default locale is en_US, the output is:

```
Hola buddy
```

We can also get all the keys in a resource bundle with the method keySet():

```
ResourceBundle rb =
```

```
 ResourceBundle.getBundle("MyBundle", spain);

Set<String> keys = rb.keySet();
keys.stream()
    .forEach(key ->
        System.out.format("%s %s\n", key, rb.getString(key)));
```

The output (notice it also prints the parent key):

```
hi Hola
s tío
```

If instead of using property files we were using classes, the program would look like this:

```
package bundles;
public class MyBundle_EN extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "buddy" }
        };
    }
}

package bundles;
public class MyBundle_es_ES extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "tío" }
        };
    }
}
```

Chapter THIRTY

```
package bundles;
public class MyBundle_es extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "s", "amigo" }
        };
    }
}

package bundles;
public class MyBundle extends ListResourceBundle {
    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            { "hi", "Hola" }
        };
    }
}

public class Test {
    public static void main(String[] args) {
        Locale spain = new Locale("es", "ES");
        Locale spanish = new Locale("es");

        ResourceBundle rb =
            ResourceBundle.getBundle("bundles.MyBundle", spain);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));
        rb = ResourceBundle.getBundle("bundles.MyBundle", spanish);
        System.out.format("%s %s\n",
            rb.getString("hi"), rb.getString("s"));
    }
}
```

The only thing that changed in the Test class was the name of the bundle (we had to reference the package). This should not surprise you, after all, both `PropertyResourceBundle` and `ListResourceBundle` inherit from the same class.

Remember also, when using classes we can have values of types other than `String`, for example:

```
public class MyBundle extends ListResourceBundle {  
    @Override  
    protected Object[][][] getContents() {  
        return new Object[][][] {  
            { "hi", "Hola" },  
            { "number", new Integer(100) }  
        };  
    }  
}
```

To get an object value, we use:

```
Integer num = (Integer)rb.getObject("number");
```

Instead of `rb.getString(key)`. In fact, this method is just a shortcut to:

```
String val = (String)rb.getObject("hi");
```

KEY POINTS

- *Localization* (abbreviated as l10N because of the number of characters between the first and the last letter) is the mechanism by which an application is adapted to a particular language and region.
- The `java.util.Locale` class basically represents a language and a country, and is the starting point for localization in Java.
- You can get the default locale of your machine with:

```
Locale locale = Locale.getDefault();
```

- You can also get all the locales supported by Java:

```
Locale [] locales = Locale.getAvailableLocales();
```

- You can create a `Locale` object using a constructor:

```
Locale(String language)
```

```
Locale(String language, String country)
```

```
Locale(String language, String country, String variant)
```

- By using the `forLanguageTag(String)` factory method:

```
Locale german = Locale.forLanguageTag("de");
```

- By using `Locale.Builder`:

```
Locale japan = new Locale.Builder()  
    .setRegion("JP")  
    .setLanguage("jp")  
    .build();
```

- By using predefined constants for some common languages and countries, for example:

```
Locale.GERMAN
```

```
Locale.KOREAN
```

- Once you have a Locale object, you can change the locale of your program with the `setDefault(Locale)` method:

```
Locale.setDefault(new Locale("en", "US"));
```

- Property files define strings in key/value pairs separated by lines.
- To localize an application, we have Resource Bundles, which define a set of keys with localized values. Resource Bundles can be property files or classes.
- java.util.PropertyResourceBundle**. Each locale is represented by a property file. Keys and values are of type String.
- java.util.ListResourceBundle**. Each locale is represented by a subclass of it that overrides the method `Object[][][] getContents()`. The returned array represents the keys and values. Keys must be of type String, but values can be any object.
- To determine which bundle belongs to a particular locale, Java tries to find the most specific bundle that matches the properties of the locale.
 - If it cannot locate one, the last component of the name is dropped until it's just the name of the bundle.
 - If nothing is found, a `MissingBundleException` is thrown.
 - If a class and a property file share the same name, Java gives priority to the class.
 - You can use the keys of the matching resource bundle and **ANY** of its **PARENTS**. The parents of a resource bundle are the ones with the same name but fewer components.

SELF TEST

1. Given:

```
public class Question_30_1 {  
    public static void main(String[] args) {  
        Locale locale = new Locale("", "");  
        ResourceBundle rb =  
            ResourceBundle.getBundle("Bundle1", locale);  
        System.out.println(rb.getString("key1"));  
    }  
}  
Bundle1.properties  
key1 = Hi
```

What is the result?

- A. Hi
- B. null
- C. Compilation fails
- D. An exception occurs at runtime

2. Which of the following are valid ways to create a locale?

- A. new Locale();
- B. Locale.Builder().setLanguage("de");
- C. new Locale.Builder().setRegion("DE").build();
- D. Locale.forRegionTag("it");

3. Assuming a default locale de_DE, which of the following resource bundles will be loaded first with

```
 ResourceBundle rb = ResourceBundle.getBundle("MyBundle");
```

- A. MyBundle.class
- B. MyBundle.properties
- C. MyBundle_de.class
- D. MyBundle_de.properties

4. Given:

```
public class Question_30_4 {  
    public static void main(String[] args) {  
        Locale locale = new Locale("en", "CA");  
        System.out.println(locale.toLanguageTag());  
    }  
}
```

What is the result?

- A. en
- B. en_CA
- C. CA
- D. CA_en

5. Which of the following are valid ways to get a value given its key from a property file resource bundle rb?

- A. rb.getValue("key");
- B. rb.getProperty("key");
- C. rb.getObject("key");
- D. rb.get("key");

ANSWERS

1. The correct answer is A.

Even though the program creates an invalid `Locale` object, it doesn't throw any exception. When Java looks up a resource bundle with that locale, it can find any so it resolves to the default bundle (`Bundle1.properties`).

2. The correct answer is C.

Option A is not a valid constructor of `Locale`. Option B is wrong, `Builder()` is not a `static` method. Option D shows an invalid method (the correct one is `forLanguageTag()`).

3. The correct answer is C.

The most specific resource bundle will be loaded first, in this case, `MyBundle_de`. Classes take precedence over properties, so Option C is the correct answer.

4. The correct answer is B.

The string representation of a `Locale` object is `language_country`.

5. The correct answer is C.

You can get a value from a resource bundle with two methods:

```
String getString(String key)  
Object getObject(String key)
```

In the case of property files, you may want to cast to `String` the returned object of `getObject()`.

FINALLY

Thank you for trusting me and reading this book.

I'm sure this book will help you pass the certification exam, but you have to do your part. You have to keep studying and practicing to make it easier and become a better (Java) developer.

Let me know if I can help you with anything, I mean it.

Until next time,

Esteban Herrera
@eh3rrera
estebanhb2@yahoo.com.mx

