
StayAhead Training
the training specialist
bringing people and technology together
Oracle Database
Oracle Development
Oracle Fusion Middleware
Oracle Business Intelligence
Oracle e-Business
MySQL
MariaDB
MongoDB
Unit Testing Frameworks
Maven
Hibernate
Acceptance Testing Frameworks
Java Technology
Web Development HTML & CSS
JavaScript & jQuery
PHP, Python Perl & Ruby
OOAD
Spring
UNIX
Solaris, AIX & HP-UX
Red Hat Linux
SUSE Linux
LPI Linux
Business Analysis ITIL® & PRINCE2®
London | Birmingham | Leeds | Manchester | Sunderland | Bristol | Edinburgh
scheduled | closed | virtual training
www.stayahead.com
+44 (0) 20 7600 6116
sales@stayahead.com

Introduction to Java 8 for Non-Programmers

JP Morgan

Introduction to Java 8 for Non-Programmers

Duration: 5 days

Introduction to Java 8 for Non-Programmers Course Overview

The Introduction to Java 8 for Non-Programmers course comprises sessions dealing with Java apps, variables and operators, decision and loop constructs, arrays, methods, objects, classes, static members, enums, object-oriented principles, exception handling, packages and imports, selected APIs and lambda expressions.

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

Skills Gained

The delegate will practice:

- Writing, compiling, and executing simple Java applications
- Declaring and initialising variables
- Constructing expressions using relational, arithmetic and logical operators
- Constructing decision and loop statements
- Constructing and manipulating arrays of data
- Declaring and invoking methods using both parameters and return values
- Constructing classes and objects
- Creating static members
- Constructing enums
- Encapsulating classes
- Implementing a hierarchical application design using inheritance
- Exploiting polymorphism
- Throwing and catching exceptions
- Organising classes into packages
- Producing code that uses wrappers, Strings, dates and ArrayLists
- Constructing a simple lambda expression

Who will the Course Benefit?

The Introduction to Java 8 for Non-Programmers course is aimed at anyone who wants to learn Java as a first language and developers/engineers who want to migrate to Java from another language, particularly those with little or no object-oriented knowledge.

Course Objectives

This course aims to provide the delegate with the knowledge to be able to produce simple Java applications that exploit all core elements of the language including variables, expressions, selection and iteration, arrays, methods, classes and objects, encapsulation, inheritance and polymorphism, exceptions, and some commonly used library classes.

Requirements

Delegates should be able to navigate the filesystem, edit a file and browse the web. No programming experience is necessary.

Follow-On Courses

- Unit Testing using Junit

Notes

- Course technical content is subject to change without notice.
- Course content is structured as sessions, this does not strictly map to course timings. Concepts, content and practicals often span sessions.

Table of Contents

Chapter 1: Applications

Chapter 2: Variables

Chapter 3: Operators

Chapter 4: Decisions

Chapter 5: Loops

Chapter 6: Arrays

Chapter 6: Arrays (continued)

Chapter 7: Methods

Chapter 8: Objects

Chapter 9: Classes

Chapter 10: Static Members

Chapter 11: Enums

Chapter 12: Encapsulation

Chapter 13: Inheritance & Polymorphism

Chapter 14: Exception Handling

Chapter 15: Packages and Imports

Chapter 16: Selected APIs

Chapter 16: Selected APIs (continued)

Chapter 17: Lambda Expressions

CHAPTER 1

Applications

Introduction

Introduced in 1995 by Sun Microsystems and now owned by Oracle, Java is a high-level, object-oriented programming language and platform that is used to develop applications for mobile phones, desktop computers, web sites and enterprise systems.

NOTE

The word platform is used here to describe the software necessary to compile and execute Java applications.

Java is currently at version 8 (sometimes referred to as 1.8) with version 9 due for release in 2017. Each version is periodically updated. At the time of writing (September 2016), the latest update for version 8 was 101.

There are three Java editions as follows:

Edition	Description
Java SE	Standard Edition. Software and libraries for the development of desktop applications. Libraries include classes for input/output, database access, networking, security, XML parsing and GUIs.
Java EE	Enterprise Edition. Software and libraries for the development of large, scalable, and secure network applications. Java EE is built on top of Java SE.
Java ME	Micro Edition. Small-scale software and libraries for small device applications.

NOTE

This course deals only with Java SE software and libraries.

Structure

A Java application is composed of one or more classes.

```
class MyClass {
   // code
}
```

Typically, each class resides in its own file, the name of which must match the class name precisely, e.g. MyClass.java.

To be executable, one of the application's classes must contain a main method. The class with the main method is the application's entry point.

```
class MyExecutableClass {
   public static void main(String[] args) {
      // code to execute
   }
}
```

NOTE

The main method signature (that part before the opening curly brace) must match the example given above precisely. Only the name of the parameter args may be changed.

Compilation and Execution

Source code (code written by developers) is usually compiled into a language the operating system can understand, i.e. machine code.

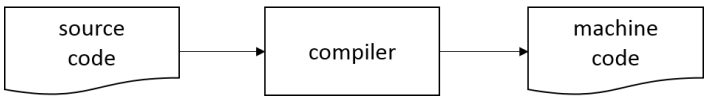


Figure 1: Source code to machine code compilation

Source code must be recompiled (using an operating system-specific compiler) for each different operating system the application is to be executed upon.

Java is different. Java source code is compiled into byte code. Operating systems do not understand byte code and so require software to translate the byte code into machine code. This process happens as the application is executed and so is referred to as interpretation.

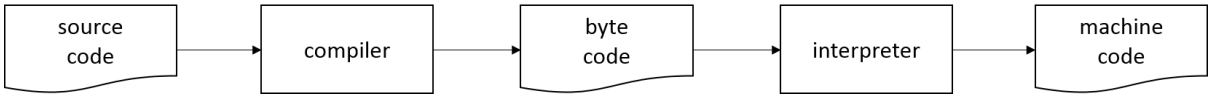


Figure 2: Source code to byte code compilation

The benefit of byte code is that Java source code does not have to be recompiled for each different operating system the application is to be executed upon. You may have heard the phrase 'write once, run anywhere' with regards to Java. There is a trade-off however. The appropriate interpreter must be installed before Java applications may be executed, but this need only happen once.

javac

The Java compiler is an executable file named **javac** and is used to compile Java source code into byte code.

```
$> javac MyClass.java
```

Many Java source code files may be compiled at once using the * wildcard.

```
$> javac *.java
```

NOTE

Java source code files must have a .java extension.

Provided there are no compilation errors, the compiler will produce a byte code file which has a .class extension.

In the event of a compilation error, the compiler will output an error message including the number of the line containing the error.

```
$> MyClass.java:3: error: invalid method declaration
   public statec void main(String[] args) {
                      ^
```

NOTE

With regards to compilation errors, the carat (^) does not necessarily point to the source of the problem. In the example given above, the compilation error is caused by the incorrect spelling (statec) of the keyword static.

java

The Java interpreter is an executable file named **java** and is used to execute byte code.

```
$> java MyClass
```

NOTE

The .class extension must be omitted when executing byte code files.

Exercises

We assume you're working on a machine running the Windows operating system. If that is not the case, ask your trainer to guide you through the first exercise – Exercise preparation.

Remember that Java is case-sensitive. MyClass is not the same as myClass, and public is not the same as Public. Indenting is crucial to the readability of your code. Be sure to indent properly. It demonstrates that you understand the structure of your code, and makes it easy to read and maintain.

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

1. Create a directory named JP1 in a location of your choosing in which you will store all the code you produce throughout the course.
2. Open a terminal window.
3. Run the Java interpreter:

   ```
   $> java
   ```

   You should see a large amount of output regarding the usage of the java command. This indicates that the Java Runtime Environment (JRE) is installed on your machine, and that you can execute Java applications. The JRE contains an interpreter (java) for executing Java applications and libraries (existing Java code) among other things.

4. Run the Java compiler:

   ```
   $> javac
   ```

   You should see a 'javac' is not recognised or similar message. This is because the operating system does not know where the javac executable file is located.

5. Open Windows Explorer and navigate to C:/Program Files/Java.

   You should see a directory whose name begins with jdk. The JDK (Java Development Kit) is like the JRE but it includes a compiler (javac) such that you can use it to develop your own Java applications.

6. Navigate into the jdk<version>/bin directory.

   This directory should contain both java and javac executables.

7. Copy the absolute path from the address bar.
8. In the terminal window, type the following at the command prompt but do not press Enter:

   ```
   $> set PATH=%PATH%;
   ```

9. Right click within the terminal window and select Paste.

   This should add the path to the JDK's bin directory to the set command.

10. Press Enter.

    You have set the operating system path (the set of locations in which the operating system looks for executables) to include the location of the javac command. Note that setting the path this way is only temporary. Search online for instructions about how to set the path permanently.

11. Run the Java compiler:

    ```
    $> javac
    ```

    You should now see a large amount of output regarding the usage of the javac command. You can now compile Java source code into byte code.

12. Do not close the terminal window. You will need it for the next exercise.

Hello world

It would not be a programming course without a Hello World application.

1. Open a text editor of your choosing.
2. Enter the following code.

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

3. Save the file in your JP1 directory as MyFirstApp.java.
4. In the terminal window, navigate to your JP1 directory.
5. Compile your source code:

   ```
   $> javac MyFirstApp.java
   ```

6. List the contents of the JP1 directory:

   ```
   $> ls
   ```

   You should see two files: MyFirstApp.java, and MyFirstApp.class. The class file is the byte code generated by the compiler. The byte code can be executed on any machine provided it has a JRE installed.

7. Execute your application:

   ```
   $> java MyFirstApp
   ```

   You should see Hello world output. Note that we omit the .class extension from the byte code file when executing it.

Eclipse

Eclipse is one of many Integrated Development Environments (IDEs) that may be used to write and test Java applications. It is, essentially, a text editor with a built-in compiler, interpreter, and console (like your terminal window). It enables you to write, compile, and execute code without having to switch between text editor and terminal window. In fact, Eclipse compiles your code on the fly. Eclipse does a lot more than this, of course, but that's all you need to know for now.

1. Open Eclipse.
2. When prompted, browse for and select your JP1 directory.
3. When Eclipse has started, select File | New | Other.
4. Select Java Project from the list of project types. If Java Project is not listed, type Java into the filter text box at the top of the dialog.
5. Click Next.
6. Enter 01 Applications in the Project name text box.
7. Click Finish.
8. If prompted to do so, open the Java perspective.
9. Expand the 01 Applications folder in Package Explorer.
10. Right click the src folder and select New | Class.
11. Enter MyFirstApp in the Name text box.
12. Click Finish.
13. Add the following code to the MyFirstApp class:

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

14. Select File | Save.
15. Select Run | Run.

16.

CHAPTER 2

Variables

Introduction

A variable is a named segment of memory that contains either some data, or a reference/pointer to another segment of memory containing some data.

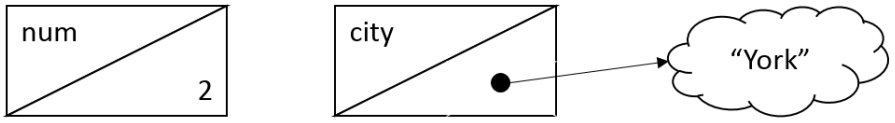


Figure 1: Variable types

Variables are so named because, typically, the data stored or referenced may be changed.

Java is a strongly-typed language which means that you must specify what type of data it is that you intend to store.

Data Types

There are two categories of data type in Java: primitive and reference.

Primitive

A primitive type variable is one in which the variable name and its associated data are stored together. Primitive data is both simple (it constitutes only one value) and small (the maximum size of a primitive variable is eight bytes).

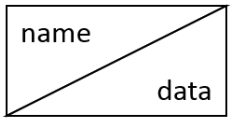


Figure 2: A primitive variable

There are eight fixed primitive data types as follows:

Type	Size	Values
boolean	1 bit	true or false
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	1.4E-45 to 3.4028235E38
double	8 bytes	4.9E-324 to 1.7976931348623157E308
char	2 bytes	U+0000 (NULL) to U+10FFFF (Noncharacter)

EXAM TIP

You don't need to know the size or range of each primitive type for the exam.

Reference

A reference type variable is one that contains a reference or pointer to some data elsewhere in memory. The referenced data (called an object) is usually complex (it may constitute many values) and large (relative to primitive data).

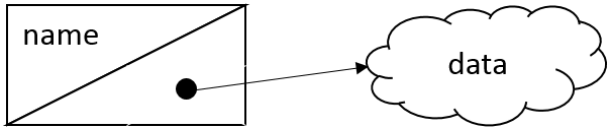


Figure 3: A reference variable

There are many reference data types in Java. In fact, you can create your own. Reference data type names should begin with a capital letter and so are easy to distinguish from primitive types.

Declaration

To declare a variable is to set aside and name a segment of memory for the storage of some data. Variable declaration takes the following form:

```
<data_type> <variable_name>;
```

To allocate memory for the storage of a whole number you might declare a primitive integer type variable as follows:

```
int num;
```

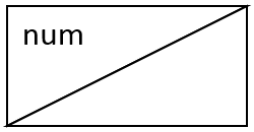


Figure 4: Primitive variable after declaration and before assignment

You can declare more than one variable of the same type on one line.

```
String city1, city2, city3;
```

NOTE

In the previous example the variables city1, city2 and city3 are reference type variables. We know this because the type name, String, begins with a capital letter.

Assignment

To assign a variable is to put some value against it. When a variable is first assigned some data it is said to have been initialised.

Primitive variable assignment takes the following form:

```
<variable_name> = <value>;
```

You might assign an integer variable as follows:

```
num = 2;
```

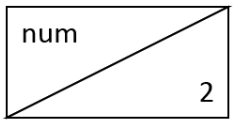


Figure 5: Primitive variable after declaration and assignment

Reference variable assignment takes the following form:

```
<variable_name> = new <data_type>(<parameters>);
```

You might assign a String variable as follows:

```
city = new String("York");
```

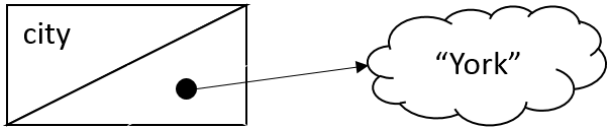


Figure 6: Reference variable after declaration and assignment

NOTE

String type variables are often mistaken for primitives because they may be assigned like a primitive.

```
String city = "York";
```

Nonetheless String is a reference data type, NOT a primitive data type.

Variables may be declared and assigned on one line.

```
int num = 2;
```

You can declare and assign more than one variable of the same type on one line.

```
String city1 = "York", city2 = "Bath", city3 = "London";
```

Naming Conventions

Variables should be named according to the following conventions:

- The name should begin with a lower-case letter, currency symbol or underscore
- The first letter of each new word should be capitalised (camel case)
- Digits may be used but not as the first character
- Currency symbols and underscores may be used

The following are legal variable names:

```
int num;
int myNumber;
int myNumber1;
int myNumber$;
int $num;
int ££££;
int _myNum;
int myNum_;
int ____;
```

The following are *illegal* variable names:

```
int 2num;
int my%num;
int num#1;
```

The following are legal but *unconventional* variable names:

```
int MyNumber;  // class names only should begin with a capital letter
int MY_NUMBER; // constant names only should be wholly capitalised
int my_number;
```

Literals

A literal is any boolean, number, character, string or object value. For example, the following variable declaration and assignment includes the character literal value a.

```
char letterA = 'a';
```

Underscores

Long numeric literals may include underscores so as to make them easier to read.

```
int bigNum = 200_000;
```

Underscores may not be placed at the beginning or end of a number, or adjacent to the decimal point. The following are examples of *illegal* underscore use:

```
int invalid1 = _123;
int invalid2 = 456_;
double invalid3 = 123_.456;
double invalid4 = 123._456;
```

Default types

Java assigns each numeric literal a default type as follows:

Literal	Example	Default type
Whole number	195	int
Floating point number	25.75	double

This has implications for assigning literals to variables. Consider the following:

```
byte b = 100;
```

The literal 100 is an int by default and it occupies four bytes of memory. The byte variable b has room for only one byte of data. And yet, this line of code compiles! This is because Java knows the number 100 will fit comfortably into one byte of memory (a byte can store values in the range -128 to 127).

Do you think the following line of code will compile?

```
short s = 35_000;
```

The literal 35,000 is an int by default and it occupies four bytes of memory. The short variable s has room for only two bytes of data. This line of code does not compile because Java knows the number 35,000 is too large to fit into two bytes of memory (a short can store values in the range -32,768 to 32,767 only).

What about this line of code?

```
long myLongNumber = 3_000_000_000; // three billion
```

At first glance it would appear that this line of code should compile. Long type variables can store values over nine quintillion (18 zeroes) and so can accommodate three billion comfortably. However, this line of code does not compile. That's because the literal 3,000,000,000 is an int by default and an int can only store values in the range -2,147,483,648 (-2.1 billion) to 2,147,483,647 (2.1 billion).

Finally, consider the following floating point example:

```
float f = 12.5;
```

The literal 12.5 is a double by default and it occupies eight bytes of memory. The float variable f has room for only four bytes of data. Unlike whole numbers, where Java will test to see if the literal will fit in the given variable, where floating point numbers are concerned any attempt to assign a double literal to a float variable will result in a compilation error.

Suffixes

When writing code with long, float or double literals, you can avoid most of the problems caused by the literal default types by adding a suffix to specify the type.

There are three literal suffixes as follows:

Suffix	Type	Example
l or L	long	long myLongNumber = 3_000_000_000L;
f or F	float	float f = 12.5f;
d or D	double	double d = 12.5d;

The suffix changes the type of the literal. The suffix l/L changes a default int literal to a long literal. The suffix f/F changes a default double literal to a float literal.

The double suffix may be used where the type of a numeric literal is ambiguous. Consider the following:

```
double d = 20;
```

The literal 20 is an int by default and so the line of code is somewhat ambiguous. You could make the code more readable in one of two of the following ways:

```
double d = 20.0; // or
double d = 20d;
```

Binary, octal and hexadecimal values

Binary, octal and hexadecimal numbers may be represented by prefixing a numeric literal with 0b, 0 or 0x respectively.

```
int binary = 0b110;  // decimal value 6
int octal = 0123;    // decimal value 83
int hex = 0x1c2;     // decimal value 450
```


Constants

A constant is a variable whose data is not changeable. A constant is created by adding the final keyword to the variable declaration.

```
final double TAX_RATE = 0.2;
```

Any attempt to reassign the variable will result in a compilation error.

```
TAX_RATE = 0.3; // compilation error
```

NOTE

By convention constant variable names should be wholly capitalised with underscores separating each word.

Scope

The scope of a variable refers both to its visibility and its lifecycle.

A variable declared inside a class but outside any method is visible to all methods in the class, and exists while ever the object or class (to which the variable belongs) exists.

```
class MyClass1 {

   int x = 1;

   void method1() {
      // x is visible here
   }

   void method2() {
      // x is visible here
   }
}
```

A variable declared inside a method is visible to the enclosing method only, and exists only while the method is executed.

```
class MyClass2 {

   void method1() {
      int y = 2;
   }

   void method2() {
      // y is not visible here
   }
}
```

A variable declared inside a block inside a method is visible to the enclosing block only, and exists only while the block is executed.

```
class MyClass3 {

   void method1() {
      while(true) {
         int z = 3;
      }
      // z is not visible here
   }

   void method2() {
      // z is not visible here
   }
}
```


Default Values

Variables declared inside a class but outside any method are assigned values by default as follows:

Data type	Default value
boolean	false
byte/short/int/long	0
float/double	0.0
char	U+0000 (NULL)
Reference (any)	null

With regards to reference types, null means the variable does not reference/point to anything.

NOTE

Default values are only assigned to variables declared inside a class but outside any method. Variables declared inside a method must be assigned a value explicitly. Failure to do so will result in a compilation error when the variable is accessed.

```
class MyClass {

   int x; // will be assigned 0 by default

   void method1() {
      int y;
      System.out.println(x); // will print 0 to the console
      System.out.println(y); // will result in a compilation error
   }
}
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Variables

1. Create a new Java Project in Eclipse named 02 Variables.
2. Create a new class named Variables.
3. Add a main method.
4. Declare and initialise variables to store the following data. Choose data types and variable names carefully. Use underscores and suffixes where appropriate.

Description	Value
Salary	£36,000.00
Grade	C
Age	27 years
Is registered	Yes
Name	Felicity
Distance from the sun	149,600,000,000m

5. Add a comment to identify the one reference variable.

   Comments are ignored by the compiler and may be inserted in one of two ways:

   ```
   // This is a single-line comment
   /*
    * This is a
    * multi-line
    * comment
    */
   ```

6. Print the value of each variable to the console with a String label, for example:

   ```
   System.out.println("Salary: " + salary);
   ```

   Used in this context, the + symbol concatenates the String literal with the variable.

7. Write two new lines of code to change the grade to B, and to print its value.
8. Save and execute your code.

CHAPTER 3

Operators

Introduction

An operator is a special symbol that can be applied to one or more variables and/or literals (referred to as operands) and that either changes the operand (unary operator) or evaluates to a value (binary operator).

An expression is a statement that evaluates to a value. The value is usually assigned to a variable. Consider the following:

```
int result = 9 – 3 * 2;
```

The part of the statement to the right of the assignment operator (=) is an expression. It evaluates to a value (3) that is subsequently assigned to the variable result.

Unary vs. Binary Operators

A unary operator is one that requires only one operand and changes its value. Consider the following:

```
int x = -1;
-x;
```

The minus symbol is (in this context) the unary operator negation. It changes the sign of the value of x such that the value of x is now 1.

A binary operator is one that requires two operands and evaluates to a value. Consider the following:

```
int x = 2;
int y = 3;
int z = x + y;
```

The plus symbol is (in this context) the binary operator addition. It adds the values of x and y (operands), and the evaluated value (5) is assigned to the variable z.

NOTE

It is important to understand the order in which the parts of the previous example statement are executed. First, the expression on the right side of the assignment operator (=) is evaluated.

```
x + y // 2 + 3 = 5
```

Then, the evaluated value is assigned to the variable z.

```
int z = 5
```


Order of Precedence

Operators follow an order of precedence. This is necessary for those expressions that comprise more than one operator. Consider the following:

```
int x = a + b * c;
```

Which part of the expression happens first? a + b or b * c?

The following table lists Java's operators in their order of precedence.

Operators	Symbols
Post-increment/decrement	<operand>++, <operand>--
Pre-increment/decrement	++<operand>, --<operand>
Other unary	+, -, !, ~
Multiplication/division/modulus	*, /, %
Addition/subtraction	+, -
Bit shift	<<, >>, >>>
Relational	<, >, <=, >=, instanceof
Equality	==, !=
Logical	&, |, ^
Short-circuit logical	&&, ||
Assignment	=, +=, -=, *=, /=, %=

Consider again the statement:

```
int x = a + b * c;
```

The order of precedence dictates that multiplication takes precedence over addition, therefore b is first multiplied by c, the result of which is then added to a.

Where an expression comprises two or more operators at the same level in the order of precedence, the expression is evaluated from left to right. Consider the following:

```
int x = a * b / c;
```

In this case neither operator takes precedence over the other, therefore a is first multiplied by b, the result of which is then divided by c.

Arithmetic Operators

An expression with an arithmetic operator yields a number. Consider the following:

```
float result = a / b;
```

The variable result is assigned the result of dividing a by b.

The following table lists Java's arithmetic operators:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (returns the remainder)

Numeric promotion

You may have noticed, from the example at the top of this page, that the result of dividing an integer by a float is a float. This is because the integer variable is automatically promoted to the float type before the expression is evaluated.

There are four numeric promotion rules as follows:

1. If the two operands are different types, Java will promote the smaller one to match the larger one. For example:

```
int myInt = 3;
long myLong = 4;
long result = myInt * myLong;
```

2. If one of the operands is an integral type (byte, short, int, long) and the other is a floating-point type, Java will promote the integral type operand to the floating-point type. For example:

```
int myInt = 3;
double myDouble = 2d;
double result = myInt + myDouble;
```

3. Operands of type byte, short and char are promoted to the int type whenever they are used with a binary arithmetic operator, even if neither operand is of type int. For example:

```
byte myByte = 1;
short myShort = 2;
int result = myByte – myShort;
```

4. After all promotion has occurred and the expression has been evaluated, the resulting value will be of the same type as the promoted operands. For example:


```
byte myByte = 2;
int myInt = 3;
float myFloat = 1.5f;
float result = myByte * myInt * myFloat;
```

In this example, myByte is first promoted to type int because it is used with a binary arithmetic operator (rule 3). The result of myByte * myInt (6) is then promoted to type float (6.0f) to be multiplied by myFloat (rule 2). Finally, the result of the expression (9.0f) is the same type as the promoted operands (rule 4) i.e. float.

Unary Operators

A unary operator is one that requires only one operand and changes its value (except not (!) which does not change the operand's value). Consider the following:

```
year++;
```

The value of the variable year is incremented by one.

The following table lists Java's unary operators:

Operator	Description
+	Indicates a number is positive (even if it isn't!)
-	Negation (changes the sign of the number)
++	Increment by one
--	Decrement by one
!	Not (inverts a boolean value)

The increment and decrement operators may be used in one of two different ways:

1. Pre-increment/decrement:

```
++x;
--x;
```

2. Post-increment/decrement:

```
x++;
x--;
```

When used in isolation there is no difference in the resultant behaviour. Crucially however, the behaviour does differ when used as part of an assignment statement. Consider the following:

```
int x = 1;
int y = ++x;
```

In this example, x is incremented before it is assigned to y. After these lines of code are executed both x and y contain the value 2.

```
int x = 1;
int y = x++;
```

In this example, x is incremented after it is assigned to y. After these lines of code are executed x contains the value 2, while y contains the value 1.

Assignment Operators

Assignment operators are used to assign a value to a variable. Consider the following:

```
int age = 39;
```

The variable age is assigned the numeric literal 39.

The following table lists Java's assignment operators:

Operator	Description
=	Assignment
+=	Compound assignment (addition)
-=	Compound assignment (subtraction)
*=	Compound assignment (multiplication)
/=	Compound assignment (division)
%=	Compound assignment (modulus)

Compound assignment operators are shorthand for simple assignment statements. Consider the following:

```
x = x + 2; // increment x by 2
```

This statement may be re-written using a compound assignment operator:

```
x += 2;
```

Relational Operators

An expression with a relational operator yields a boolean. Consider the following:

```
boolean result = a > b;
```

The variable result is assigned true if a is greater than b, else result is assigned false.

The following table lists Java's relational operators:

Operator	Description
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to

Testing for equality

There are three important rules associated with testing for equality as follows:

1. When comparing two numeric primitive types the rules of numeric promotion apply. For example:

   ```
   int x = 3;
   float y = 3.0f;
   boolean result = x == y;         // result is assigned true
   ```

   The int type variable x is promoted to float according to numeric promotion rule 2, therefore the result is assigned true.

2. A boolean value may only be compared with another boolean value. For example:

   ```
   boolean b1 = true == 2;          // compilation error
   boolean b2 = false != "Hello";   // compilation error
   boolean b3 = true == (2 > 1)     // b3 is assigned true
   ```

3. When comparing reference types, the references/pointers are compared, not the referenced data (objects). For example:

   ```
   String s1 = new String("Hello");
   String s2 = new String("Hello");
   boolean result = s1 == s2;       // result is assigned false
   ```

   It would appear that result ought to be assigned true, but the equality operator compares the values of the variables which, in this case, are references/pointers to the data (memory addresses of the data), not the data itself.

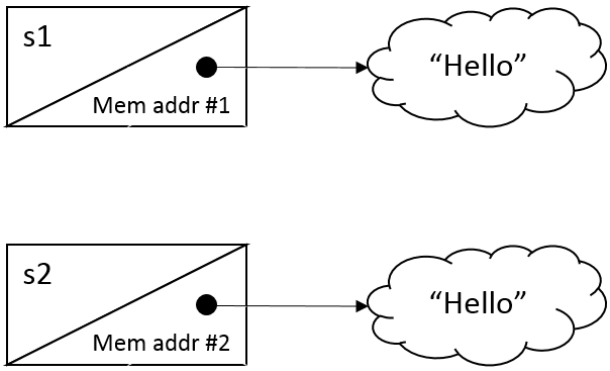


Figure 1: Two unequal reference variables

NOTE

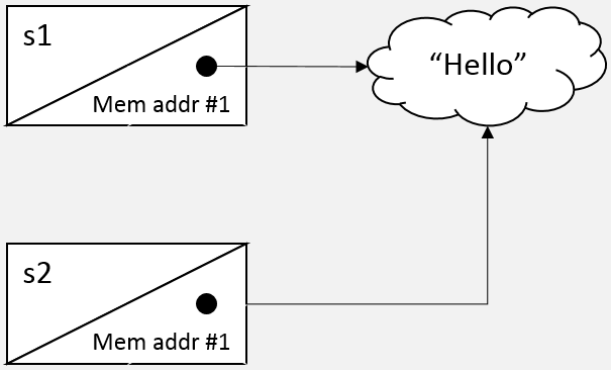
Whenever the new keyword is used in the assignment of a reference type variable, a new object is created with a new reference/pointer/memory address. However, when a String variable is declared like a primitive…

```
String name = "William";
```

...the behaviour is different. If a String object is created *without* the new keyword Java will attempt to reuse an existing String object in memory (assuming one exists). Therefore…

```
String s1 = "Hello";
String s2 = "Hello";
boolean result = s1 == s2;
```

…result will be assigned true! The variables s1 and s2 reference the same String object.



Logical Operators

Logical operators are used to construct compound boolean expressions (an expression composed of two or more expressions). Consider the following:

```
boolean result = a > b && b > c;
```

The variable result is assigned true if a is greater than b AND b is greater than c.

The following table lists Java's logical operators:

Operator	Description
&&	AND (short-circuit)
||	OR (short-circuit)
^	XOR (eXclusive OR)
&	AND
|	OR

The following rules apply with regards to logical operators:

- Where AND is used the compound expression evaluates to true only if both expressions evaluate to true.
- Where OR is used the compound expression evaluates to true if one or both of the expressions evaluates to true.
- Where XOR is used the compound expression evaluates to true only if one expression evaluates to true and the other expression evaluates to false.

Consider the following truth tables:

x & y (AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y (OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (XOR)

	y = true	y = false
x = true	false	true
x = false	true	false

Figure 2: Logical operator truth tables

When evaluating a compound boolean expression with a non-short-circuit logical operator, Java will always evaluate both expressions, but this is not always necessary. Consider the following:

```
boolean b1 = 1 > 2 & 6 > 4;
```

The first expression (1 > 2) evaluates to false and so b1 will be assigned false regardless of the second expression. Nonetheless, Java will evaluate the second expression (6 > 4) because the non-short-circuit logical operator AND (&) is used.

Short-circuit logical operators are designed to eliminate such redundant evaluations. Consider the following:


```
boolean b2 = 1 > 2 && 6 > 4;
```

The first expression (1 > 2) evaluates to false and so b1 will be assigned false regardless of the second expression. In this case, Java will ***not*** evaluate the second expression (6 > 4) because the short-circuit logical operator AND (&&) is used.

The following rules apply to short-circuit logical operators:

- Where AND (&&) is used, if the first expression evaluates to false, Java will not evaluate the second expression – the result must be false regardless.
- Where OR (||) is used, if the first expression evaluates to true, Java will not evaluate the second expression – the result must be true regardless.

NOTE

Careless use of short-circuit logical operators may introduce bugs into your code. Consider the following:

```
x <= 10 && (y = 10) > z
```

If x is greater than 10, y will not be assigned the literal value 10 because Java will not evaluate the second expression. Take care to use the appropriate logical operator when assigning variables within a compound boolean expression.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

Create a new Java Project in Eclipse named 03 Operators.

Operators

Assuming:

```
int a = 7, b = 3;
boolean truth = false;
```

Complete the table below by writing the result of each expression.

If you need to, create a class named Operators. Add a main method, and code each expression in a System.out.println statement. For example:

```
public class Operators {
   public static void main(String[] args) {
      int a = 7, b = 3;
      boolean truth = false;
      System.out.println(a + b);
      System.out.println(a – b);
      ...
   }
}
```

NOTE

Remember that unary operators change the value of the operand. You should assume a, b, and truth are reset to 7, 3, and false respectively after each expression.

If you're not sure why you get a particular result, ask your trainer for help. Add comments where appropriate.

Arithmetic	Result
a + b	
a - b	
a * b	
a / b	
a % b	

Unary	**Result**
+a	
-b	
++a	
--b	
a++	
b--	
!truth	
Relational	**Result**
a == b	
a != b	
a > b	
b >= 4	
a < b	
a <= 7	
Logical	**Result**
a > b && b < a	
a == b && b < a	
a < b && b > a	
a > b || b < a	
a == b || b < a	
a < b || b > a	
a > b ^ b < a	
a > b ^ b > a	
Order of precedence	**Result**
a + b * 2	
(a + b) * 2	
15 / b * a	
14 % b + b	
--a * b++	
a++ * 2 + --b	

Equality

1. Create a new class named Equality.
2. Add a main method.
3. Declare and initialise two int variables. Name one int1 and the other one int2, and assign each one the value 3.
4. Declare a boolean variable named isEqual and assign it the following expression:

```
int1 == int2
```

5. Print the value of isEqual to the console with a String label, for example:

```
System.out.println(3 is equal to 3: " + isEqual);
```

6. Declare a double variable named double1 and assign it the value 3.0d.
7. Assign isEqual the following expression:

```
int1 == double1
```

8. Print the value of isEqual to the console with a String label.
9. Declare and initialise two String variables as follows:

```
String string1 = "Hello";
String string2 = "Hello";
```

10. Assign isEqual the following expression:

```
string1 == string2
```

11. Print the value of isEqual to the console with a String label.
12. Declare and initialise two more String variables as follows:

```
String string3 = new String("world");
String string4 = new String("world");
```

13. Assign isEqual the following expression:

```
string3 == string4
```

14. Print the value of isEqual to the console with a String label.
15. Add a comment to explain the resultant output.
16. Save and execute your code.

CHAPTER 4

Decisions

Introduction

A decision or control statement is one in which the flow of execution is determined by the result of a boolean expression (that is, an expression that evaluates to true or false).

There are two principle decision statements in Java: if else and switch. And while these structures are semi-interchangeable, they are fundamentally different and are designed to be used in different scenarios.

The ternary operator is a third, simpler, form of decision statement used to assign a value to a variable.

If Else

A typical if else statement takes the following form:

```
if(<boolean_expression>) {
   // code to execute if expression evaluates to true
} else {
   // code to execute if expression evaluates to false
}
```

In the event only one line of code is to be executed in either block, the curly braces may be omitted.

```
if(<boolean_expression>)
   // one line of code to be executed if expression evaluates to true
else
   // one line of code to be executed if expression evaluates to false
```

NOTE

If omitting the curly braces, take care to indent properly. Incorrectly indented code can result in bugs, for example:

```
if(<boolean expression>)
  <statement 1>;
  <statement 2>;
```

While it appears that both statement 1 and 2 will be executed only if the expression evaluates to true, this is not the case. Statement 2 will be executed regardless of the decision statement.

One branch

A one branch if else statement comprises an if without an else.

```
if(age >= 18) {
   canVote = true;
}
// code here is executed in any case
```

Two branches

A two branch if else statement comprises both if and else.

```
if(age >= 18) {
   canVote = true;
} else {
   canVote = false;
}
```


Three or more branches

An if else statement with three or more branches comprises two or more boolean expressions.

```
if(birthYear <= 1945) {
   generation = "Silent/Traditionalist";
} else if(birthYear > 1945 && birthYear <= 1964) {
   generation = "Baby Boomer";
} else if(birthYear > 1964 && birthYear <= 1979) {
   generation = "X";
} else if(birthYear > 1979 && birthYear <= 1995) {
   generation = "Millenial";
} else {
   generation = "Z/Centennital/iGen";
}
```

NOTE

Only one branch of a multi-branch if else statement is ever executed, even if two or more boolean expressions evaluate to true.

Nesting

If else statements may be nested.

```
if(fileExists == true) {
   if(numRecords >= 1) {
      // code to execute if both expressions evaluate to true
   }
}
```

EXAM TIP

The exam is likely to contain one or more questions regarding decision statements with curly braces omitted, incorrect indenting, and nesting.

Switch

A switch statement is used to test if one value is equal to another. It may only be used to test values of type byte, short, char, int, String, and enum. Crucially, and unlike an if else statement, more than one branch may be executed.

A switch statement takes the following form:

```
switch(<value_to_test>) {
   case <value_#1>:
      // code to execute if <value_to_test> == <value_#1>
      // this segment may contain many lines
   case <value_#2>:
      // code to execute if <value_to_test> == <value_#2>
   case <value_#3>:
      // code to execute if <value_to_test> == <value_#3>
   default:
      // code to execute if <value_to_test> != <value_#1/2/3>
}
```

If the value to test matches value #1, then ALL the statements beneath the case will be executed including those associated with the other cases!

Consider the following:

```
int option  = 3;
switch(option) {
   case 1:
      System.out.println("Doing task 1…");
   case 2:
      System.out.println("Doing task 2…");
   case 3:
      System.out.println("Doing task 3…");
   case 4:
      System.out.println("Doing task 4…");
   default:
      System.out.println("Doing task 5…");
}
```

When executed, the code above will yield the following output:

```
$> Doing task 3…
$> Doing task 4…
$> Doing task 5…
```

A switch statement may be forced to behave like an if else statement by including the break keyword at the end of each set of case statements.

Consider the following:

```
int option  = 3;
switch(option) {
   case 1:
      System.out.println("Doing task 1…");
      break;
   case 2:
      System.out.println("Doing task 2…");
      break;
   case 3:
      System.out.println("Doing task 3…");
      break;
   case 4:
      System.out.println("Doing task 4…");
      break;
   default:
      System.out.println("Doing task 5…");
}
```

When executed, the code above will yield the following output:

```
$> Doing task 3…
```

NOTE

The break keyword forces the interpreter to jump to the end of the switch statement.

Ternary Operator

The ternary operator is used to assign one or another value to a variable depending on the result of a boolean expression without having to code a complete if else statement.

Assignment of a variable using the ternary operator takes the following form:

```
<var> = (<boolean_expression>) ? <value_if_true> : <value_if_false>
```

Consider the following:

```
int x = (5 > 3) ? 1 : 0;
```

This is equivalent to:

```
if(5 > 3) {
   x = 1;
} else {
   x = 0;
}
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

Create a new Java Project in Eclipse named 04 Decisions.

If else

1. Create a new class named IfElse.
2. Add a main method.
3. Add the following line of code which assigns a randomly generated number between 1 and 7 to an int variable named dayOfTheWeek.

```
int dayOfTheWeek = (int) (Math.random() * 7) + 1;
```

4. Add an if else statement as follows:

   If the day of the week is 1 – 4, print "It's a work day" to the console.

   If the day of the week is 5, print "It's nearly the weekend" to the console.

   If the day of the week is 6 or 7, print "It's the weekend!" to the console.

5. Save and execute your code.

Switch

1. Create a new class named Switch.
2. Add a main method.
3. Add the following line of code which assigns a randomly generated number between 1 and 3 to an int variable named carCleanOption.

```
int carCleanOption = (int) (Math.random() * 3) + 1;
```

4. Add a switch statement as follows:

   If the car clean option is 3, print "Wash", "Tyre shine", and "Vacuum" to the console each on separate lines.

   If the car clean option is 2, print "Wash" and "Tyre shine" to the console each on separate lines.

   If the car clean option is 1, print "Wash" to the console.

5. Save and execute your code.

CHAPTER 5

Loops

Introduction

A loop or iterative statement is one in which a block of code is repeated while the result of a boolean expression evaluates to true.

There are three principle loop statements in Java: for, while, and do. And while these structures are fully interchangeable (any one may be used in any scenario) each one is best suited to a given type of scenario.

The break and/or continue keywords may be used within a loop to disrupt the normal flow of execution. Break is used to force the interpreter to jump out of the loop, while continue is used to force the interpreter to the beginning of the next iteration.

For

A typical for loop takes the following form:

```
for(<init_ctrl_var>; <boolean_expression>; <incr/decr_ctrl_var>) {
   // code to be repeated
}
```

Consider the following:

```
for(int i = 0; i < 5; i++) {
   System.out.println("for: " + i);
}
```

The initialisation of the control variable i to 0 happens only once at the very start. Then at the beginning of each loop the boolean expression is evaluated. If the expression evaluates to true, the code within the for loop block is executed. At the end of each loop the control variable i is (in this particular case) incremented by one.

When executed, the code above will yield the following output:

```
$> for: 0
$> for: 1
$> for: 2
$> for: 3
$> for: 4
```

NOTE

The control variable needn't be incremented/decremented by one. For example, to print all odd numbers from 1 to 10:

```
for(int i = 1; i < 10; i += 2) {
   System.out.println("for :" + i);
}
```

Multiple control variables

You may include more than one control variable in your for loop.

```
for(int x = 0, int y = 1; x < 10; x += 2, y += 2) {
   System.out.println("for x: " + x + ", y: " + y);
}
```

Empty (infinite) loop

It is possible to code an empty and, consequently, infinite for loop by omitting code between the semicolons.

```
for( ; ; ) {
   // infinite loop code
}
```

Best used when…

For loops are best used to iterate over a collection or when the number of iterations is known and fixed.

While

A while loop takes the following form:

```
while(<boolean_expression>) {
   // code to be repeated
}
```

Consider the following:

```
while(currentLine != null) {
   System.out.println(currentLine);
   // read next line into currentLine variable
}
```

The code within the while loop block is never executed if the boolean expression evaluates to false the first time. Otherwise the code is executed repeatedly until the boolean expression evaluates to false.

In the example code above, currentLine is the control variable. It is crucial that that the control variable is changed within the while loop block to avoid infinite loops.

Consider the following:

```
int x = 0;
while(x < 5) {
   System.out.println("while: " + x);
}
```

When executed, the code above will yield an infinite loop – x is always less than 5.

Infinite loop

It is possible to code an empty and, consequently, infinite while loop by coding a boolean expression that always evaluates to true.

```
while(true) {
   // infinite loop code
}
```

Best used when…

While loops are best used when the number of iterations may be none, one or more.

Do

A do loop takes the following form:

```
do {
   // code to be repeated
} while(<boolean_expression>);
```

Consider the following:

```
int age = 0;
do {
   // prompt the user for his/her age
} while(age < 18 || age > 120);
```

The code within the do loop block is executed at least once, and then is repeated until the boolean expression evaluates to false.

In the example code above, age is the control variable. It is crucial that that the control variable is changed within the while loop block to avoid infinite loops.

Infinite loop

It is possible to code an empty and, consequently, infinite do loop by coding a boolean expression that always evaluates to true.

```
do {
   // infinite loop code
} while(true);
```

Best used when…

Do loops are best used when there must be at least one iteration.

Break

The break keyword is used to force the interpreter to jump out of a loop. Break is often used in conjunction with an if else statement nested inside the loop.

Consider the following:

```
int age = 0;
int numAttempts = 0;
do {
   if(numAttempts == 3) {
      break;
   }
   // prompt the user for his/her age
   numAttempts++;
} while(age < 18 || age > 120);
```

The interpreter will jump out of the loop if the user has failed to enter a valid age after three attempts.

Break should be used sparingly. In most cases a comprehensive boolean expression will mitigate the need for it. On the other hand, using break may improve code readability.

Continue

The continue keyword is used to force the interpreter to the beginning of the next iteration.

Consider the following:

```
// print all numbers from 1 to 10 excluding 6
for(int i = 1; i <= 10; i++) {
   if(i == 6) {
      continue; // skip the remaining loop code for this iteration
   }
   System.out.println("for: " + i);
}
```

When executed, the code above will yield the following output:

```
$> for: 1
$> for: 2
$> for: 3
$> for: 4
$> for: 5
$> for: 7
$> for: 8
$> for: 9
$> for: 10
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

Create a new Java Project in Eclipse named 05 Loops.

For, while and do

1. Create a new class named Loops.
2. Add a main method.
3. Add a for loop to print multiples of 3 from 3 to 15 to the console.
4. Repeat step 3 using a while loop.
5. Repeat step 3 using a do loop.
6. Save and execute your code.

Battleship

You're going to produce a very simple version of the board game Battleship. If you're not familiar with it, Battleship requires each player to attempt to sink the other's battleship by bombing coordinates chosen at random. In this case the second player is the computer, and it chooses its battleship position (a number from 1 to 10) at random. The player (user) then gets three attempts to guess the battleship's position before the game ends.

1. Create a new class named Battleship.
2. Add a main method.
3. Add the following line of code which assigns a randomly generated number between 1 and 10 to an int variable named battleshipPosition.

```
int battleshipPosition = (int) (Math.random() * 10) + 1;
```

4. Declare an int variable named guess and assign it the value 0;
5. Declare an int variable named numAttempts and assign it the value 0.
6. Add the following line of code which creates a Scanner object which you'll use to obtain input from the user.

```
java.util.Scanner keyboard = new java.util.Scanner(System.in);
```

7. Add a do loop such that the body of the loop will be executed while numAttempts is less than 3.
8. Inside the do loop:
   a. Print "Enter a position between 1 and 10: " to the console.
   b. Add the following line of code which reads the user input into the int variable named guess.


```
guess = keyboard.nextInt();
```

c. Add an if statement to test the user's guess is equal to the computer's battleship position. If they're equal, print "You sunk the computer's battleship! You win!" to the console and then break out of the loop.
d. Increment the numAttempts variable by 1.

9. Add an if statement to test the number of attempts is equal to 3. If they're equal, print "You lose!" to the console.
10. Add the following line of code to close the Scanner object.

```
keyboard.close();
```

11. Save and execute your code.

CHAPTER 6

Arrays

Introduction

An array is a fixed-size collection of non-unique, indexed values of the same data type. Arrays are commonly used when the number of values to be stored is known in advance and will not change.

NOTE

The size of an array cannot change once it is initialised.

An array can store primitive data or references/pointers. The array itself however is an object, and so the variable used to access it is a reference variable, regardless of the type of data stored.

Arrays may be one-dimensional or many-dimensional. A 2D array, for example, is an array in which each element of the array contains another array.

Declaration

Array declaration may take either of the following forms:

```
<data_type>[] <array_name>;
<data_type> <array_name>[];
```

Whilst both forms are valid, the second is unconventional.

Consider the following:

```
int[] numbers;
String[] names;
```

The variable numbers references an array that may be used to store primitive int values only. The variable names references an array that may be used to store String values only.

Multi-dimensional array declaration

The number of sets of square brackets determines the number of dimensions. Consider the following:

```
char[][] my2dArray;
```

The variable my2dArray references an array that may be used to store arrays of type char.

Assignment

Array assignment may take either of the following forms:

```
<array_name> = new <data_type>[<size>];
<array_name> = {<value_#1>, <value_#2>, <value_#3>, ...};
```

In the first instance, the array is initialised with a given size using the new keyword. In the second instance, the array is initialised with n values. The size of the array is, therefore, determined by the number of values specified.

Consider the following:

```
numbers = new int[5];
names = {“John”, “Paul”, “George”, “Ringo”};
```

When an array is initialised by specifying a size (as in the first instance above), each element of the array is assigned a default value as follows:

Data type	Default value
boolean	false
byte/short/int/long	0
float/double	0.0
char	U+0000 (NULL)
Reference (any)	null

Multi-dimensional array assignment

Assigning a value to a multi-dimensional array requires that you specify two or more indices. Consider the following:

```
char[][] my2dArray = new char[5][4];
```

The outer array contains five char arrays. Each inner array contains 4 primitive char values. To set the char value in the second slot of inner array four:

```
my2dArray[3][1] = ‘A’;
```

Setting and Getting

Setting elements

The setting of an element in an array takes the following form:

```
<array_name>[<index>] = <value>;
```

Consider the following:

```
names[0] = "John";
```

NOTE

Array indices always begin at 0. Therefore the first element is always accessed using index 0, and the length of an array (its size) is always one more than the largest index.

Getting elements

The getting of an element takes the following form:

```
<variable_name> = <array_name>[<index>];
```

Consider the following:

```
String thirdBeatle = names[2];
```

Traversing

Traversing an array may be achieved using any loop type, but the for loop is most common. Each array has a length property which is used in the boolean expression part of the loop. Consider the following:

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

EXAM TIP

The boolean expression uses the < (less than) operator as opposed to the <= (less than or equal to) operator when traversing an array because the length of the array is always one more than the largest index. Lookout for this common error in the exam.

Traversing multi-dimensional arrays

The traversal of a multi-dimensional array requires nested loops. Consider the following:

```
int rows = 3;
int cols = 3;
char[][] tictactoeBoard = new int[rows][cols];
tictactoeBoard[2][0] = 'X';
tictactoeBoard[0][0] = 'O';
tictactoeBoard[1][1] = 'X';
tictactoeBoard[0][1] = 'O';
tictactoeBoard[0][2] = 'X';
for(int row = 0; row < rows; row++) {
   for(int col = 0; col < cols; col++) {
      System.out.print(tictactoeBoard[row][col] + "\t");
   }
   System.out.println(); // print a new line
}
```

The code of the inner (cols) loop is executed three times for each execution of the outer (rows) loop. This is because each element of the array contains an array of size three.

When executed, the code above will yield the following output:

```
$> O  O  X
$>    X
$> X
```

EXAM TIP

You should expect multi-dimensional array questions on the exam.

The enhanced for loop

The enhanced for loop provides a simpler way to traverse an array and takes the following form:

```
for(<data_type> <variable_name> : <array_name>) {
   // do something with each element of the array
}
```

This is equivalent to for each or for in, in other programming languages.

Consider the following:

```
for(String name : names) {
   System.out.println(name);
}
```

NOTE

While simpler than a traditional for loop, the enhanced for loop does not provide access to the element index.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Exercise preparation

Create a new Java Project in Eclipse named 06 Arrays.

Arrays

1. Create a new class named Arrays.
2. Add a main method.
3. Declare a four element array of Strings named vanHalen.
4. Add the following values:

   "Eddie Van Halen"

   "David Lee Roth"

   "Alex Van Halen"

   "Michael Anthony"

5. Traverse the array, printing each name to the console.
6. Replace David Lee Roth with Sammy Hagar.
7. Print the second element only to the console.
8. Save and execute your code.

Command line arguments

The main method (the one that makes your class executable) is configured to accept an array of Strings named args. When you run a Java application on the command line, any values that follow the class name are copied into an array of Strings and passed to the main method. Consider the following command:

```
$> java MyApp Hello world
```

The words Hello and world will be copied into the args array and made available in the main method, thus making the application interactive.

1. Create a new class named PatronSaint.
2. Add a main method.
3. Add an if statement. If the length of the args array is not 1, print "Usage: PatronSaint <country>" to the console and terminate the application using the following line of code:

   ```
   System.exit(1);
   ```

4. Declare a String variable named country and assign it args[0].

5. Add an if else statement as follows. Note that you should not use == to test for equality of String values. You should instead use the String class's equals method, for example country.equals("England") which will return a boolean value.

   If the country is "England", print "George" to the console.

   If the country is "Wales", print "David" to the console.

   If the country is "Scotland", print "Andrew" to the console.

   If the country is "Ireland", print "Patrick" to the console.

   If the user enters anything else, print "Invalid country" to the console.

6. Save your code.
7. Open a terminal window and navigate to your JP1 directory, then into 06 Arrays.
8. List the contents of the current directory.

   Note that the 06 Arrays project directory contains two sub-directories: src and bin. The src directory contains your source code (.java) files. The byte code (.class) files are saved in the bin directory by default.

9. Navigate into the bin directory.
10. Run your application as follows:

```
$> java PatronSaint Scotland
```

11. The word Scotland is a program argument, and will be added to the args array before the main method is invoked. The output should be:

```
$> Andrew
```

12. Test your application with other country names.

CHAPTER 7

Methods

Introduction

A segment of code that occurs repeatedly within your program is a candidate method. A method is a named set of statements that may be invoked/called using the method name only. This means that many lines of code may be replaced by one. Methods reduce code duplication and may be reused by many applications.

Methods may require input (parameters/arguments) and/or produce output (return value). In Java, any combination is allowed:

- no input or output
- input only
- output only
- input and output

In other programming languages, what is known to a Java developer as a method, may be referred to as a function, procedure, or subroutine.

Declaration

A method declaration is the specification of the set of statements and its associated name that may be invoked/called later. A simple method declaration takes the following form:

```
<return_type> <method_name>(<parameters>) {
   // statements to be executed
}
```

Consider the following:

```
void printHello() {
   System.out.print("Hello");
   System.out.print(" ");
   System.out.print("World");
}
```

The method name is printHello and it comprises three statements. It neither requires input – there is nothing between the round brackets – nor does it produce output – the return type is void.

Invocation/Call

To invoke/call a method is to execute the statements specified in the method declaration. A method invocation/call takes the following form:

```
<method_name>(<arguments>);
```

Consider the following:

```
printHello();
```

This one line of code instructs the interpreter to execute the statements specified in the method declaration. When executed, the code above will yield the following output:

```
$> Hello World
```

Parameters

A parameter is an item of data that a method needs to do its job. A method may require zero, one or more parameters, each separated by a comma. Parameters are local to the method, that is, they exist only for as long as the method is executing. Consider the following:

```
void printGrade(int mark) {
   if(mark >= 90) {
      System.out.println('A');
   } else if(mark >= 80) {
      System.out.println('B');
   } else if(mark >= 70) {
      System.out.println('C');
   } else if(mark >= 60) {
      System.out.println('D');
   } else if(mark >= 50) {
      System.out.println('E');
   } else {
      System.out.println('F');
   }
}
```

This method requires an int value to do its job. Now consider the method invocation:

```
printGrade(77);
```

The int value 77 is copied into the parameter named mark so it may be used by the statements in the method declaration. The int value 77 is, in the context of a method invocation/call, an argument.

Now consider a second invocation/call of the same method:

```
int myMark = 63;
printGrade(myMark);
```

In the same way, the myMark variable value is copied into the parameter named mark.

Now consider a third invocation/call of the same method:

```
int mark = 91;
printGrade(mark);
```

Now we have two variables named mark. One that exists outside of the method declaration, and the method parameter. They are not the same variable. Here again the variable value is copied into the parameter named mark.

NOTE

Regardless of the argument type, Java always copies the argument value into the parameter. For a primitive type argument, this means copying some data. For a reference type argument, this means copying a reference/pointer.

Return Type

A return type is a specification of the type of data to be output (returned) by a method. Consider the following:

```
char getGrade(int mark) {
  char grade = 'N';
  if(mark >= 90) {
     grade = 'A';
  } else if(mark >= 80) {
     grade = 'B';
  } else if(mark >= 70) {
     grade = 'C';
  } else if(mark >= 60) {
     grade = 'D';
  } else if(mark >= 50) {
     grade = 'E';
  } else {
     grade = 'A';
  }
  return grade;
}
```

The method specifies that it will output/return a char type value. The return keyword is used to output/return a value of the specified type. Failure to return a value of the specified type will result in a compilation error. Now consider the method invocation/call:

```
char grade = getGrade(77);
```

The value returned by the method is copied into to a char variable named grade. Note that this variable is distinct from that declared inside the method, despite their having the same name. The grade variable declared inside the method is local, that is, it exists only while the method is executing.

NOTE

Only one value may be returned by a method, but the value itself may be an object, array, or some other collection containing or referencing many items.

Overloading

Overloading is the specification of two or more methods with the same name. This is only allowed on the condition that each version of the method specifies a different parameter list. This enables the interpreter to determine which version of the method to execute based on the arguments specified when the method is invoked/called. Consider the following method declarations:

```
void sayHello() {
   System.out.println("Hello");
}

void sayHello(String name) {
   System.out.println("Hello " + name);
}

void sayHello(String firstName, String lastName) {
   System.out.println("Hello " + firstName + " " + lastName);
}
```

Each method has the same name, but a different parameter list. The sayHello method is said to be overloaded. Now consider the method invocation/call:

```
sayHello("Sarah");
```

The interpreter can determine which version of the method to execute since the method was invoked with one String argument. When executed, the code above will yield the following output:

```
$> Hello Sarah
```

NOTE

Only the parameter list is considered in method overloading, the return type is not. The following will yield a compilation error:

```
void sayHello() {
   // code to execute
}

String sayHello() {
   // code to execute
}
```

Output as Input

The output of one method may serve as the input to another. Consider the following:

```
char myGrade = getGrade(83);
System.out.println("My grade is " + myGrade);
```

These two lines of code may be refactored such that the output of the getGrade method serves as part of the input to the println method:

```
System.out.println("My grade is " + getGrade(83));
```

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Methods

1. Create a new Java Project in Eclipse named 07 Methods.
2. Create a new class named Methods.
3. Add a main method.
4. Add a static method named printAllOddNumbers that accepts an int but does not return any value. As the name suggests, the method should print each odd number from 0 to the number passed as an argument. The method must be static so that it may be invoked/called from the main method. More about static later. The method signature should look like this:

```
static void printAllOddNumbers(int number) {
      // TODO
}
```

5. Test your printAllOddNumbers method by invoking/calling it in the main method.
6. Save and execute your code.
7. Add a second static method named getCircumference that accepts a double representing the radius of a circle, and returns a double representing the circumference.

   Hint: the formula to calculate the circumference of a circle is 2 x Pi (3.14) x radius.

8. Test your getCircumference method by invoking/calling it in the main method. Print the result to the console with a String label.
9. Save and execute your code.
10. Add a third static method named isPrime that accepts an int and returns a boolean. If the number is prime, it should return true else it should return false.

    Hint: add a for loop that starts at 2 and increments by 1 while the loop counter is less than the argument. If the remainder of dividing the argument by the loop counter is 0, the number is not prime and you should return false. If the loop completes, the number is prime and you should return true.

11. Test your isPrime method by invoking/calling it in the main method. Print the result to the console with a String label.
12. Save and execute your code.

CHAPTER 8

Objects

Introduction

An object is a complex data structure that may comprise many properties. A property may be data or a method. A reference variable references/points to an object.

Objects are fundamental to Java. After all, it is an object-oriented language. Objects are good because they provide for the encapsulation, into one data structure, of the data and methods (behaviour) associated with that data.

Creation and Assignment

Object creation often takes the following form:

```
new <data_type>();
```

Consider the following:

```
new Account();
```

Of course, the newly created object is usually assigned to a reference variable so as to be accessible later.

```
Account myAccount = new Account();
```

NOTE

You might be inclined to say that myAccount is an Account object (many developers would!), but that is not accurate. myAccount is a reference variable that references/points to an Account object.

Unlike primitive types, of which there are only eight, there are hundreds (if not thousands) of reference types included in the platform libraries for you to use. You have already encountered one – String. Indeed, you can create your own reference types as you will see in the next chapter.

Instance Fields

An instance field is a property of an object which contains some data. An object's state is determined by the values of its instance fields.

An Account object might comprise the following instance fields:

- String name
- int number
- double balance

Getting

The getting of the value of an instance field takes the following form:

```
<variable_name> = <object_name>.<field_name>;
```

Consider the following:

```
String accountName = myAccount.name;
```

The value of the Account object's name field is copied into the accountName variable.

Setting

The setting of the value of an instance field takes the following form:

```
<object_name>.<field_name> = <value>;
```

Consider the following:

```
myAccount.name = "Johnson";
```

The value of the Account object's name field is changed to "Johnson".

Instance Methods

An instance method is a method that is a property of an object. An object's behaviour is determined by the set of its instance methods.

An Account object might comprise the following instance methods:

- void deposit(double amount)
- void withdraw(double amount)

Invoking/calling

The invoking/calling of an instance method takes the following form:

```
<object_name>.<instance_method_name>();
```

Consider the following:

```
myAccount.deposit(100d);
```

Instance methods typically either use or change one or more of the object's instance fields. In this case, the value of the balance instance field will be changed by the invocation of the deposit method.

Chaining

Where a method returns a reference to another object, methods may be chained together. Consider the following:

```
Account davies = bank.getAccount("Davies");
double daviesBalance = davies.getBalance();
```

The bank object's getAccount method returns a reference to an Account object. If we need only to use the account object once, there is no need for the Account reference variable. Instead, we can chain the getBalance method onto the end of the getAccount method. Consider the following:

```
double daviesBalance = bank.getAccount("Davies").getBalance();
```


References as Parameters

When a method is invoked/called, the values of any arguments are **copied** into the associated method parameters. Consider the following:

```
void changeIt(int num) {
   num += 2;
}
int x = 1;
changeIt(x);
System.out.println("x is " + x);
```

When the changeIt method is invoked/called, the value of x is copied into num. The parameter num is then incremented by two. But this has no effect on the variable x. Indeed, the parameter num only exists while the changeIt method is executing. When executed, the code above will yield the following output:

```
$> x is 1
```

Consider the following:

```
void changeIt(Account account) {
   account.name = "Williams";
}
Account myAccount = new Account();
myAccount.name = "Simpson";
changeIt(account);
System.out.println("Account name is " + myAccount.name);
```

When the changeIt method is invoked/called, the value of myAccount is copied into account. Note however that myAccount is a reference variable and its value is a reference/pointer to an account object. This means that, following the copy, the parameter account references/points to the same Account object as does myAccount. Therefore, any changes to the Account object using account will be reflected by myAccount too. When executed, the code above will yield the following output:

```
$> Account name is Williams
```

Lifecycle

Consider the following:

```
myAccount = null;
```

The null keyword means that the reference variable references/points to nothing. Setting a reference variable to null has the effect of dereferencing the object in memory. But this does not mean that the object is deleted from memory. In other languages, this would be known as a memory leak. The dereferenced object occupies some memory that is:

1. not accessible
2. cannot be used by other data

Fortunately, we do not have to code the reclamation of such memory. The JVM includes a daemon called the Garbage Collector which runs from time to time freeing up memory occupied by dereferenced objects.

The Garbage Collector does its work when the JVM deems it necessary, and you cannot dictate (in code at least) when the Garbage Collector runs.

NOTE

It is possible (indeed it is common) that one object is referenced by many variables. Setting one variable to null does not necessarily mean that the object is eligible for garbage collection.

Consider the following:

```
String s1 = new String("Hello");
String s2 = s1;
s1 = null;
```

The variable s1 references/points to a String object containing the value "Hello". On the second line, the value of s1 is copied into s2. That is, the reference/pointer is copied. Therefore, the variable s2 now references/points to the same String object as does s1. On the third line, s1 is set to null, but the String object is not eligible for garbage collection because it is referenced by s2.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Objects

1. Create a new Java Project in Eclipse named 08 Objects.
2. Create a new class named Objects.
3. Add a main method.
4. Declare a String variable named myString and assign it a String object as follows:

   ```
   String myString = new String("Hello world");
   ```

5. Add a comment noting that myString is a reference/pointer to a String object containing "Hello world". The part of the statement on the left side of the assignment operator (=) is the declaration of a String reference variable, and the part of the statement on the right side of the assignment operator is the creation of a new String object.

   We'll now use the myString reference to access some of the String object's properties.

6. Type the following on a new line (don't press Enter):

   ```
   myString.
   ```

   Eclipse should list all the instance fields and methods associated with the String object. If it does not, and with the cursor positioned immediately after the dot, press Ctrl + Space.

   Note that there are no instance fields listed, only instance methods. This is not uncommon and is the result of encapsulation, of which more later.

   The first instance method listed is likely to be length because it is the most commonly used. Note that the length method does not accept any arguments, and returns an int – length() : int.

7. Use your arrow keys to select the length method and press Enter.
8. As the length method returns an int, declare an int variable named myStringLength and assign it the method invocation:

   ```
   int myStringLength = myString.length();
   ```

9. Print the value of myStringLength to the console.
10. Save and execute your code.

Each String object has a substring method that accepts an int index, and returns a reference to a new String object. Note that a String object is actually an array of chars. In fact, every object can be deconstructed to a set of primitives. The int argument passed to the String object's substring method is the array index from which the characters are to be copied into a new String object.

11. Invoke the String object's substring method to extract the word world into a new String object:

```
String mySubstring = myString.substring(6);
```

12. Print the value of mySubstring to the console.
13. Save and execute your code.

NOTE

If mySubstring is a reference/pointer to an object, and not itself an object, why does the following line of code print the word world to the console?

```
System.out.println(mySubstring);
```

Surely it should print a memory address, after all, that's what the variable contains. When you print a reference variable in this way, Java automatically invokes the object's toString method. The above line of code is the same as:

```
System.out.println(mySubstring.toString());
```

The String object's toString method is written in such a way as to return the value of the object referenced, and not the reference itself. This is not always the case.

14. Typically, you should not declare a variable if it's only used once – as is the case here with both myStringLength and mySubstring. Refactor your code such that the behaviour is the same, whilst removing the two aforementioned variables.
15. Save and execute your code.

CHAPTER 9

Classes

Introduction

Whilst the Java platform libraries contain hundreds of reference types that enable you to do complex things quickly and easily, you will not find reference types that represent business-specific entities such as bank accounts, employees, blogs, or social media profiles. In these such cases, you will want to develop your own reference types.

Each reference type is a class. A class is a template from which objects are created. A class is used to specify what fields and methods each object of that type should have.

Many programming languages allow for the creation of objects without a template, but there are advantages to the Java approach. If each object of a given type is created using a template, then you can be confident each object will have certain properties, and that each object will exhibit certain behaviours. And this makes processing large quantities of objects easier than might otherwise be the case.

NOTE

The word instance, as in instance field or instance method, is synonymous with object. An object is an instance of a class.

Structure

A class typically contains the specification of some/all of the following:

- instance fields
- constructor(s)
- instance methods

A simple class declaration takes the following form:

```
class <class_name> {
   // instance fields
   // constructor(s)
   // instance methods
}
```

NOTE

Whilst it doesn't matter to the compiler/interpreter in what order the fields, constructors, and methods appear in the class, it is conventional to specify fields at the top, followed by constructors, with methods at the bottom.

Instance Fields

An instance field is a property of an object which contains some data. In the context of a class, it is a variable that is declared inside the class but outside of any method.
Consider the following:

```
class Account {
   // instance fields
   String name;
   int number;
   double balance;
   // constructor(s)
   // instance methods
}
```

Each Account object created using this class will have a name, number, and balance.

Each instance field is assigned a default value as follows:

Data type	Default value
boolean	false
byte/short/int/long	0
float/double	0.0
char	U+0000 (NULL)
Reference (any)	null

You can override these default values inline as follows:

```
class Account {
   // instance fields
   String name = "N/A";
   int number = -1;
   double balance = 0d;
   // constructor(s)
   // instance methods
}
```

NOTE

It is preferable to initialise instance fields with default values in a constructor than doing so inline as described above.

Instance Methods

An instance method is a method that is a property of an object. In the context of a class, it is a method that is declared inside the class. Consider the following:

```
class Account {
   // instance fields
   String name;
   int number;
   double balance;
   // constructor(s)
   // instance methods
   void deposit(double amount) {
      balance += amount;
   }
   void withdraw(double amount) {
      if(balance >= amount) {
         balance -= amount;
      } else {
         System.out.println("Insufficient funds");
      }
   }
}
```

Each Account object created using this class will have deposit and withdraw functionality. Note that each method changes the state of the object in the form of the balance field.

Constructors

A constructor is the mechanism used to create an object. Consider the following:

```
Account myAccount = new Account();
```

Account() is the invocation of a constructor. A constructor is like a method but with some very important differences:

- The name of a constructor must match the name of the class precisely
- A constructor cannot return any value
- If you do not add a constructor to your class, the compiler will add one for you

NOTE

The default constructor added by the compiler does not specify any parameters.

You should use the constructor to dictate how an object of a given type is created. In the case of an Account object, one might argue that it does not make sense to create an Account object without a name and number. Consider the following:

```
class Account {
   // instance fields
   String name;
   int number;
   double balance;
   // constructor(s)
   Account(String name, int number) {
      this.name = name;
      this.number = number;
      balance = 0d;
   }
   // instance methods
   void deposit(double amount) {
      balance += amount;
   }
   void withdraw(double amount) {
      if(balance >= amount) {
         balance -= amount;
      } else {
         System.out.println("Insufficient funds");
      }
   }
}
```

The constructor requires String and int arguments that will be used to set the name and number instance fields. Note that both the instance fields and parameters have the same name. The *this* keyword is used to distinguish the instance field from the parameter. To be precise, the *this* keyword refers to the current object. Note too that the balance instance field is initialised inside the constructor. It is good practice to initialise all your instance fields inside the constructor.

NOTE

The compiler does not add a default constructor if you add one of your own. In the example code above, this means that an Account object may only be created when a name and number is provided at creation time. This is a good thing. It ensures that business rules are applied in code.

It is often the case that business rules allow for objects to be created in multiple ways. For example, business rules might dictate that an account may be opened with an initial balance. You can achieve this by overloading the constructor. Consider the following:

```
class Account {
   // instance fields
   String name;
   int number;
   double balance;
   // constructor(s)
   Account(String name, int number) {
      this.name = name;
      this.number = number;
      balance = 0d;
   }
   Account(String name, int number, double balance) {
      this(name, number); // invoke the other constructor
      this.balance = balance;
   }
   // instance methods
   void deposit(double amount) {
      balance += amount;
   }
   void withdraw(double amount) {
      if(balance >= amount) {
         balance -= amount;
      } else {
         System.out.println("Insufficient funds");
      }
   }
}
```

The constructor is overloaded to enable the creation of Account objects with or without an opening balance. In the case of the second constructor, the first line is an invocation of the first constructor. This is known as constructor chaining and it reduces code repetition.

Assuming the Account class as specified above, an Account object might be created as follows:

```
Account account1 = new Account("Harrison", 6675); // or...
Account account2 = new Account("Kim", 5190, 50d);
```


Initialiser Blocks

An initialiser block is an unnamed, unlabelled block of code within a class that may be added to reduce code repetition within overloaded constructors. Any code within the initialiser block is added to the top of each constructor. Consider the following:

```
class InitBlockTest {
   // constructor
   InitBlockTest() {
      System.out.println("Inside the constructor");
   }
   // init block
   {
      System.out.println("Inside the init block");
   }
}
```

Now consider the creation of an InitBlockTest object as follows:

```
InitBlockTest ibt = new InitBlockTest();
```

Execution of the code about will yield the following output:

```
$> Inside the init block
$> Inside the constructor
```

EXAM TIP

Initialiser blocks are unlikely to appear on the exam.

Naming Conventions

Classes should be named according to the following conventions:

- The name should begin with an upper-case letter
- The first letter of each new word should be capitalised
- Digits may be used but not as the first character
- Currency symbols and underscores may be used

NOTE

A class name should reflect the thing it represents and is typically a noun.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Classes (with code)

1. Create a new Java Project in Eclipse named 09 Classes.
2. Create a new class named TrainingCourse.

```
class TrainingCourse {

}
```

3. Add instance fields to represent the course name, duration (days), price per person per day, delegates (the names of those people booked to attend), and number of delegates currently booked to attend.

```
String name;
int duration;
double pricePerPersonPerDay;
String[] delegates;
int numDelegates;
```

4. Add a constructor that accepts a name and duration only.

```
TrainingCourse(String name, int duration) {
   this.name = name;
   this.duration = duration;
   pricePerPersonPerDay = 250d;
   delegates = new String[5];
   numDelegates = 0;
}
```

   Note that the pricePerPersonPerDay, delegates, and numDelegates instance fields are also initialised in the constructor. The default price is £250/person/day, delegates is assigned a new String array of size 5, and the number of delegates currently booked to attend is set to 0.

5. Add an instance method that calculates and returns the course's total price.

```
double getTotalPrice() {
   return duration * pricePerPersonPerDay * numDelegates;
}
```

6. Add an instance method for assigning a delegate to the course. This method requires a String name, and must add the name to the array of delegates and increment the number of delegates by 1, unless the course is fully booked.

```
void assignDelegate(String name) {
   if(numDelegates < 5) {
      delegates[numDelegates] = name;
```

```
            numDelegates++;
            System.out.print("Successfully assigned " + name);
        } else {
            System.out.println("Can't assign " + name
                + ", the course is fully booked");
        }
    }
```

7. Add an instance method that prints to the console the name of each delegate currently booked to attend.

```
void printDelegates() {
    System.out.println("Assigned delegates:");
    for(int i = 0; i < numDelegates; i++) {
        System.out.println(delegates[i]);
    }
}
```

8. Save your code.
9. Create a new class named TrainingCourseTest.
10. Add a main method.
11. Declare a TrainingCourse variable named java and assign it a new TrainingCourse object. Pass the constructor the values "Java" and 5.

```
TrainingCourse java = new TrainingCourse("Java", 5);
```

12. Invoke the assignDelegate method on your TrainingCourse object 6 times.

```
java.assignDelegate("John");
java.assignDelegate("Sarah");
java.assignDelegate("Tim");
java.assignDelegate("Grace");
java.assignDelegate("Theo");
java.assignDelegate("Fran");
```

13. Invoke the getTotalPrice method on your TrainingCourse object and print the return value to the console with a String label.

```
System.out.println("Total course price: " + java.getTotalPrice());
```

14. Invoke the printDelegates method on your TrainingCourse object.

```
java.printDelegates();
```

15. Save and execute your code.

Case Study – Part 1

1. Create a new Java project in Eclipse named Case Study.
2. Create a new class named Book (of the Library variety).
3. Add instance fields to represent the book's name, author, whether the book is currently on loan, and its popularity (a number representing the number of times the book has been loaned out).

4. Add a constructor that accepts a name and author only. Initialise the on loan instance field to false, and the popularity instance field to 0.
5. Add instance methods to check the book out and in. A book that is currently out on loan cannot be checked out. Each time a book is checked out its popularity should be incremented by 1. Print success and failure messages to the console.
6. Save your code.
7. Create a new class named Library.
8. Add instance fields to represent the library's name, its collection of books (use an array for this), and the number of books currently in the collection.
9. Add a constructor that accepts a name only. Initialise the array of books as you see fit, and the number of books currently in the collection to 0.
10. Add an instance method to find a book. This method requires the name of a book, and should iterate over the library's collection of books searching for a match. If a match is found, the method should return a reference to the book, which can be used to check it out/in. If no match is found, the method should return null.
11. Add an instance method for adding a book to the library's collection. This method requires a reference to a Book object, and should add the book only if:
    a. it does not already exist in the collection, and…
    b. the library is not full.

    Print success and failure messages to the console.
12. Save your code.
13. Create a new class named LibraryTest.
14. Add a main method.
15. Declare a Library variable named myLibrary and assign it a new Library object.
16. Invoke the add method on your Library object 3 or 4 times. In each case, pass the add method an anonymous Book object, for example:

    ```
    myLibrary.add(new Book("Decline and Fall", "Evelyn Waugh"));
    ```

    The Book object is anonymous because we did not declare a Book variable to reference it. That's okay, because the Book object is accessible via the Library object referenced by myLibrary.
17. Invoke the find method on your Library object, passing the name of a book you've added, and assign the return value to a Book variable named myBook.
18. Use the myBook variable to check the book out.
19. Print the popularity of the Book object referenced by myBook to the console.
20. Save and execute your code.
21. Time permitting, add more code to your LibraryTest class to test the adding of an existing book; the adding of a book to a full library; the finding of a book that doesn't exist and; the checking out of a book that is already out on loan.

CHAPTER 10

Static Members

Introduction

The static keyword is a source of misunderstanding for many Java developers. This may be due, in part, to the fact that the word static suggests something that does not change. But this is not what static means in Java.

The static keyword may be applied to fields and methods (members) of a class, and effectively means that the member belongs to the class (template) and not to each object created using that template. A field or method with the static keyword applied is not an instance member. Rather it is a static or class member.

When an object is created, Java first loads the class needed to create the object into memory. Once the class is loaded into memory, it may be used to create any number of objects of that particular type. Each object gets its own set of instance members. Static members, however, are attached to the class and so regardless of the number of objects created, there will only ever be one instance of each static member.

Static Fields

A static field is a variable declared inside a class but outside of any method, and has the static keyword applied. A static field declaration takes the following form:

```
static <data_type> <variable_name>;
```

A field should be made static if its value is to be shared by/will be the same for all objects of the given type. Let's assume you've been tasked with creating a class to represent a tweet. The class might look something like this:

```
class Tweet {
   // instance fields
   Author author;
   LocalDateTime dateTime;
   String content;
   int maxCharacters;
}
```

Which of the Tweet class's fields should be made static?

The maxCharacters field should be made static because its value will be shared by/the same for all Tweet objects. Let's refactor the class as follows:

```
class Tweet {
   // instance fields
   Author author;
   LocalDateTime dateTime;
   String content;
   // static/class fields
   static int maxCharacters;
}
```

The maxCharacters field now belongs to the class, and not to each object created using the class. Regardless of the number of Tweet objects created, there will only ever be one maxCharacters field shared by them all.

Accessing static fields

Static fields belong to the class and so should be accessed using the class name.

```
Tweet.maxCharacters = 140;
System.out.println(Tweet.maxCharacters);
```

Unfortunately, static fields may be accessed using an object reference too.

```
Tweet myTweet = new Tweet();
myTweet.maxCharacters = 150;
```


NOTE

While legal, it is very bad practice to access a static field using an object reference. This is for two reasons: 1. it gives the impression that the field being accessed is an instance field when it is not, and 2. changes to the static field using an object reference will affect all objects, not just the one used to change the field.

Constants

Constants are typically both final (cannot be changed) *and* static (only one shared by all objects). In the code example above, maxCharacters should probably be a constant unless the business rules allow for the maximum number of characters to change. Let's refactor the class once more as follows:

```
class Tweet {
   // instance fields
   Author author;
   LocalDateTime dateTime;
   String content;
   // static/class fields
   static final int MAX_CHARACTERS;
}
```

Note that, by convention, constants should be named using upper case letters and underscores.

Static Methods

A static method is a method that is declared inside the class and has the static keyword applied. A static method declaration takes the following form:

```
static <return_type> <method_name>(<parameters>) {
   // statements to be executed
}
```

A method should be made static if it neither uses nor changes one or more instance fields. Let's assume you've been tasked with creating a class to represent a route from one point to another (think Google Maps). The class might look something like this:

```
class Route {
   // instance fields
   Point[] points;
   // instance methods
   float getTotalDistance() {
      // TODO
      return 0f;
   }
   float convertToKms(float miles) {
      // TODO
      return 0f;
   }
}
```

Which of the Route class's methods should be made static?

The convertToKms method should be made static because it neither uses nor changes one or more instance fields. To put it another way, the convertToKms method has no relationship with the object state. The getTotalDistance method, conversely will use the information contained in the points array instance field to calculate the total distance. Let's refactor the class as follows:

```
class Route {
   // instance fields
   Point[] points;
   // instance methods
   float getTotalDistance() {
      // TODO
      return 0f;
   }
   // static/class methods
   static float convertToKms(float miles) {
      // TODO
      return 0f;
   }
}
```


Accessing static methods

Static methods belong to the class and so should be accessed using the class name.

```
float distanceInKms = Route.convertToKms(100f);
```

NOTE

A static method does not require that an object be created to use it. Beware of creating static methods out of convenience. Always apply the rule described above. Overuse of static methods may indicate that you are coding procedurally.

Unfortunately, static methods may be accessed using an object reference too.

```
Route myRoute = new Route();
float distanceToKms = myRoute.convertToKms(100f);
```

NOTE

While legal, it is very bad practice to access a static method using an object reference. This is because it gives the impression that the method being accessed is an instance method when it is not.

Static Context

Statements that are executed inside a static method are said to be so in a static context. You cannot access non-static members from a static context. Consider the following:

```
class StaticContext {
   // instance fields
   String name;
   // instance methods
   void printName() {
      System.out.prinltn("Static context: " + name);
   }
   // static/class methods
   static void myStaticMethod() {
      printName(); // compilation error
   }
}
```

The attempted invocation of the instance method printName inside the static method myStaticMethod will yield a compilation error. The reason for this is that the class (and by implication all static members) will exist in memory *before* any objects created using that class will. That is, there may not be any StaticContext objects on which to invoke the printName method. Even if there are StaticContext objects in memory, how does the static method know on which one to invoke printName? It doesn't, and so consequently non-static members cannot be accessed from a static context.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Static Fields

1. Create a new Java Project in Eclipse named 10 Static Members.
2. Create a new class named Product (as in something to sell).
3. Add instance fields to represent the product description and its price.
4. Add a static field to represent VAT (value-added tax) and set its value to 0.2.
5. Add a constructor that accepts a description and a price.
6. Save your code.
7. Create a new class named ProductTest.
8. Add a main method.
9. Print the value of the VAT static field to the console.
10. Save and execute your code.

    Note that you don't need a Product object to access the VAT static field. It belongs to the class (template) and exists in memory before any Product objects are created.

11. Declare a Product variable named bat and assign it a new Product object. Pass the constructor the values "Bat" and 20d.
12. Declare a second Product variable named ball and assign it a new Product object. Pass the constructor the values "Ball" and 3d.
13. Change the price of the bat from 20d to 25d:

```
bat.price = 25d;
```

14. Print the price of each Product object to the console:

```
System.out.println("Bat price: " + bat.price);
System.out.println("Ball price: " + ball.price);
```

15. Save and execute your code. The output should be:

    Bat price: 25.0
    Ball price: 3.0

    Note that price, like description, is an instance field. Each Product object get its own price and description.

16. Change the VAT static field using the variable ball:

```
ball.vat = 0.25; // not good practice
```

17. Print the vat for each Product object to the console:

```
System.out.println("VAT for the bat: " + bat.vat);
System.out.println("VAT for the ball: " + ball.vat);
```

18. Save and execute your code. The output should be:

    VAT for the bat: 0.25
    VAT for the ball: 0.25

    As VAT is a static field, there is only one instance of it, regardless of the number of Product objects. All Product objects share the same VAT static field. For this reason, you should only ever access static members using the class name, e.g. Product.vat.

Case Study – Part 2

In the first part of the case study you created a Library class comprising an array of Books. It is very likely that you will have set the size of said array using an integer literal. This is not good practice. You have probably used the same integer literal in the add instance method to test the library is not full. If business rules change, you (or some other unfortunate developer) will have to change all occurrences of the integer literal.

1. Add and initialise a constant (static final) field to represent the maximum number of books the library can maintain and name it appropriately.
2. Use the constant you just added to initialise the books array in the constructor.
3. Use the constant you just added to test the library is not full in the add instance method.
4. Save your changes and execute LibraryTest to ensure the application still works as expected.

CHAPTER 11

Enums

Introduction

The range of values that may be assigned to an instance field may, in some cases, have to be limited. Consider the following:

```
class SocialMediaProfile {
   // instance fields
   String relationshipStatus;
   ...
}
```

The range of valid values that may be assigned to the relationshipStatus instance field is likely to be limited, but the field may be set to any valid String value, and consequently may lead to bugs or, at the very least, require a significant amount of validation.

One solution is to create a class that contains a fixed set of static constants. Consider the following:

```
class Status {
   // static/class fields
   static final Status SINGLE = new Status("SINGLE");
   static final Status IN_A_RELATIONSHIP = new Status("IN_A_RELATIONSHIP");
   static final Status MARRIED = new Status("MARRIED");
   static final Status DIVORCED = new Status("DIVORCED");
   ...
}
```

We could now refactor the SocialMediaProfile class as follows:

```
class SocialMediaProfile {
   // instance fields
   Status relationshipStatus;
   ...
}
```

This means that the relationshipStatus instance field may only be set to one of a fixed set of valid values. But there is a better, simpler way, and that is to use an enum.

An enum is a special type of class that contains a fixed set of static constants.

Structure

A simple enum takes the following form:

```
enum <enum_name> {
   <CONSTANT_1_NAME>, <CONSTANT_2_NAME>, <CONSTANT_3_NAME>, ...;
}
```

Yes, it's that simple. Consider the following:

```
enum RelationshipStatus {
   SINGLE, IN_A_RELATIONSHIP, MARRIED, DIVORCED;
}
```

This looks like a specification of constant names only. But what is the data type? And what is the value of each constant?

Each constant listed within an enum is an object of the given type. In the example code above, each constant is an object of type RelationshipStatus (an enum is compiled into a class by the compiler). The value of each constant is its name.

NOTE

Each constant specified in an enum is static and final by default.

Usage

Enum values are accessed in the same way a static constant is accessed:

```
<variable_name> = <enum_name>.<constant_name>;
```

Consider again the SocialMediaProfile class; this time, however, the relationshipStatus instance field is of enum type RelatioshipStatus:

```
class SocialMediaProfile {
   // instance fields
   RelationshipStatus relationshipStatus; // enum type
   ...
}
```

Now consider the setting of the relationshipStatus instance field:

```
SocialMediaProfile myProfile = new SocialMediaProfile();
myProfile.relationshipStatus = RelationshipStatus.IN_A_RELATIONSHIP;
```

And the getting:

```
System.out.println(myProfile.relationshipStatus);
```

Execution of the code above will yield the following output:

```
$> IN_A_RELATIONSHIP
```

EXAM TIP

An enum may be far more complex than that described here. It is, after all, a class in disguise. For the exam, however, you need only to be able to create and use enums at this level.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Enums

1. Create a new Java project in Eclipse named 11 Enums.
2. You've been tasked with developing an application for a used car dealership.

   The attributes of a car in which we're interested are make, model, type, and colour. All of these are candidate enums, but for the sake of the exercise, you're going to create a type and colour enum only.
3. Create a new enum named CarType comprising the types: saloon, hatchback, and SUV. Create another new enum named CarColour comprising the colours: white, black, grey, sliver, and red.
4. Save your code.
5. Create a new class named Car.
6. Add instance fields to represent the car's make, model, type, and colour.
7. Add a constructor that accepts a make, model, type, and colour.
8. Save your code.
9. Create a new class named CarTest.
10. Add a main method.
11. Declare a Car variable named myCar and assign it a new Car object. Pass the constructor the values "Ford", "Focus", CarType.HATCHBACK, and CarColour.BLACK.
12. Print the value of each instance field to the console with a String label.
13. Save and execute your code.
14. Try assigning the String literal "BLUE" to the Car object's colour field.

Case Study – Part 3

Your library client has decided she wants to categorise each book.

1. Create a new enum named Genre comprising the genres: fiction, travel, science, and history.
2. Save your code.
3. Add an instance field to the Book class to represent the book's genre.
4. Refactor the Book class constructor to accept a name, author, and genre.
5. Save your code.
6. Refactor the LibraryTest class to account for the changed Book constructor.
7. Save and execute your code.

CHAPTER 12

Encapsulation

Introduction

Encapsulation is one of the three core principles of object-oriented programming (the other two being inheritance and polymorphism). Encapsulation dictates that an object's instance fields should not be accessible directly. This is to ensure that an object's state is not placed in an invalid state. Consider the following:

```
class Employee {
   // instance fields
   String name;
   double salary;
   ...
}
```

It is very likely that business rules will place limitations on the values that may be assigned to an Employee object's name and salary. For example, the name must not be empty, and the salary must be in the range 10-90 thousand. Encapsualtion enables the enforcement of such business rules in code.

NOTE

The word encapsulation sounds a bit like capsule, and it might help to think of a class as a capsule containing a specification of instance fields and instance methods. All class members should be relevant to the thing you're trying to represent, and each instance method should either use or change one or more instance field.

Access Modifiers

An access modifier is a keyword that may be prepended to a class, field, or method to dictate which other classes may access it.

There are four access modifiers as follows:

Access modifier	Member accessible by…
public	any class in any package
protected	any class in the same package or by any child class in any package
<none>	any class in the same package
private	other members in the same class only

NOTE

A package is effectively a folder of classes used to organise code and to avoid conflicts between classes with the same name.

Consider the following filesystem of Java classes:

- Package1
  - ClassA
  - ClassB
- Package2
  - ClassC
  - ClassD (child of ClassB)

First, let's assume ClassA contains a public instance method. That method is accessible by any class in any package – in this case, ClassB, ClassC and ClassD.

Second, let's assume ClassB contains a protected instance method. That method is accessible any class in the same package, or by any child class in any package – in this case, ClassA and ClassD.

Third, let's assume ClassC contains an instance method with no access modifier. That method is accessible by any class in the same package – in this case, ClassD.

Finally, let's assume ClassD contains a private instance method. That method is only accessible to other members of the same class.

Encapsulating a Class

To encapsulate a class and thereby protect it from being placed in an invalid state:

1. Make the instance fields private
2. Add public getter and setter methods
3. Where appropriate, add validation code to setter methods

NOTE

Not all instance fields should have both getter and setter methods. For example, IDs are often generated internally and therefore should not be settable by the user.

Getter (accessor) methods

A getter or accessor method is a public method that enables the getting of a private instance field. They are typically very simple – they have a return type that matches the type of the instance field to get, and no parameters. Consider the following:

```
class Employee {
   // instance fields
   private String name;
   private double salary;
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
}
```

NOTE

By convention, getter methods should be named using the word get followed by the instance field name with the first letter capitalised. Many Java frameworks like Spring and Hibernate require that getter methods are so named.

Setter (mutator) methods

A setter or mutator method is a public method that enables the setting of a private instance field. They are typically very simple – they don't return anything, and have one parameter, the type of which matches the type of the instance field to be set. Consider the following:

```
class Employee {
   // instance fields
   private String name;
   private double salary;
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public void setName(String name) {
      this.name = name;
   }
   public void setSalary(double salary) {
      this.salary = salary;
   }
}
```

NOTE

By convention, setter methods should be named using the word set followed by the instance field name with the first letter capitalised. Many Java frameworks like Spring and Hibernate require that setter methods are so named.

Note that the name of the setter method parameter is the same as the instance field in each case. This is very common. The *this* keyword is used to distinguish the instance field from the parameter. To be precise, the *this* keyword refers to the current object.

We've added a significant amount of code to the class, but we've not protected objects of this type from being placed in an invalid state. To do that, we must add some validation code to the setter methods.

Setter method validation

Each setter method should be written to ensure the business rules are adhered to. In the introduction, we suggested that business rules might dictate that an Employee object's name must not be empty, and that the salary must be in the range 10-90 thousand. Let's refactor the code to enforce these rules:

```
class Employee {
   // instance fields
   private String name;
   private double salary;
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
```



```
    }
    public void setName(String name) {
       if(name != null && !name.equals("")) {
          this.name = name;
       }
    }
    public void setSalary(double salary) {
       if(salary >= 10_000 && salary <= 90_000) {
          this.salary = salary;
       }
    }
}
```

NOTE

When testing a String for empty values, you must use the equals instance method. When used with reference variables, the equality operator (==) tests the references/pointers for equality, not the values of the objects.

The example code above will ensure an Employee object's name/salary is not set to an invalid value, but what happens in the event the user does attempt to set the object's state using an invalid value? The answer is, as it stands, nothing. Exception handling provides the mechanism to deal with anticipatable problems such as this, as you will see.

Objects of an Encapsulated Class

Let's now look at the setting and getting of an Employee object's state. Note that the Employee class may now be considered encapsulated.

```
// create an Employee object
Employee simon = new Employee();
// set the Employee object's state
simon.setName("Simon");
simon.setSalary(30_000);
// get the Employee object's state
System.out.println("Name: " + simon.getName());
System.out.println("Salary: " + simon.getSalary());
```

Might an Employee object be placed in an invalid state? What is missing from the Employee class that we should add?

When an Employee object is first created, it is in an invalid state. The String name is, by default, null, and the double salary is, by default, 0.0d. We can solve this problem by adding a constructor and invoking the setter methods from within it.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Case Study – Part 4

Our Book and Library classes are not encapsulated inasmuch as the instance fields are not private, and they've no getters and setters.

1. Encapsulate the Book class so as to enforce the following business rules:
   a. A book's name, author, loan status, popularity, and genre may be accessed.
   b. A book's name may be changed.
   c. A book's name must contain at least one character.
   d. A book's author may be changed.
   e. A book's loan status must only be changed on check out/in.
   f. A book's popularity must only be changed on check out.
   g. A book's genre may be changed.
   h. Book objects may be created by any class in any package.
   i. A book's check in and check out behaviour should be available to any class in any package.
2. Save your code.
3. Encapsulate the Library class so as to enforce the following business rules:
   a. A library's name, books, and number of books may be accessed.
   b. A library's name may be changed.
   c. A library's collection must only be changed on the adding of a book.
   d. A library's number of books must only be changed on the adding of a book.
   e. Library objects may be created by any class in any package.
   f. A library's find and add behaviour should be available to any class in any package.
4. Save your code.
5. Refactor the Library class's find and add instance methods to account for the encapsulation of the Book class.
6. Save your code.
7. Refactor the LibraryTest class to account for the encapsulation of the Book and Library classes.
8. Save and execute your code.

CHAPTER 13

Inheritance & Polymorphism

Introduction

Inheritance and polymorphism are two of the three core principles of object-oriented programming (the other being encapsulation).

Inheritance is the principle that allows for a class to inherit its fields and methods from another. Inheritance specifies an IS-A relationship between two classes. For example, audio track IS-A specific type of media; tablet IS-A specific type of communications device, and so on.

The class from which fields and methods are inherited may be referred to as any one of:

- Super class
- Base class
- Parent

The class that inherits fields and methods may be referred to as any one of:

- Sub class
- Derived class
- Child

A class may be both super and sub simultaneously. For example, ClassB inherits its fields and methods from ClassA, while ClassC inherits its fields and methods from ClassB.

NOTE

Java only allows for single inheritance. That is, a given class may have only one super class. This is because two or more super classes may each have members with the same name. In this event, there is no simple way for the sub class to determine which of the identically named members it should inherit.

Inheritance offers two main benefits as follows:

1. Reduces code duplication
2. Enables polymorphism (this is more important than 1)

Polymorphism is a by-product of inheritance. The word means many forms and is the principle that one object may be of many types. That is, an object may contain fields and methods that meet the specification of more than one class.

An AudioTrack object meets the specifications of the AudioTrack class *and* the Media class (assuming the AudioTrack class inherits from the Media class). This is powerful because wherever a Media object is expected (think of a method that requires a Media object to be passed as a parameter), an AudioTrack object may be substituted. This enables you to build scalable applications.

Inheriting from a Super Class

The extends keyword is used in the class declaration to specify that one class should inherit from another. Consider the following:

```
public class AudioTrack extends Media {
   // no code here
}
```

Let's assume the Media class looks like this:

```
public class Media {
   // instance fields
   private String title;
   private float length;
   // constructor(s) - TODO
   // instance methods
   public void play() {
      System.out.println("Playing generic media: " + title);
   }
   public void pause() {
      System.out.println("Pausing generic media: " + title);
   }
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public float getLength() {
      return length;
   }
   public void setLength(float length) {
      this.length = length;
   }
}
```

The AudioTrack class will inherit the fields and methods of the Media class. The following, therefore, is valid code making use of the AudioTrack class:

```
AudioTrack track1 = new AudioTrack();
track1.setTitle("Overture – Marriage of Figaro");
track1.setLength(4.35f);
track1.play();
```


Sub Class Differentiation

There is little point in coding one class to inherit from another if nothing distinguishes the sub class from the super class. Typically, you will implement inheritance when you want to *extend* the super class. That is, when you want build on top of the fields and methods specified in the super class.

NOTE

It is very important that the IS-A rule is maintained. Inheritance should only be implemented when the one class IS-A specific type of the other.

There are two ways to differentiate the sub class from the super class:

1. Add new fields and/or methods
2. Override one or more super class methods

Adding new fields and/or methods

Adding new fields and/or methods to the sub class is simple. Consider the following additions to the AudioTrack class:

```
public class AudioTrack extends Media {
   // instance fields
   private String artist;
   // instance methods
   public String getArtist() {
      return artist;
   }
   public void setArtist(String artist) {
      this.artist = artist;
   }
}
```

The AudioTrack class now specifies three instance fields:

- title [inherited from Media]
- length [inherited from Media]
- artist [specified in AudioTrack]

And eight instance methods:

- play [inherited from Media]
- pause [inherited from Media]
- getTitle [inherited from Media]
- setTitle [inherited from Media]
- getLength [inherited from Media]
- setLength [inherited from Media]
- getArtist [specified in AudioTrack]
- setArtist [specified in AudioTrack]

Overriding Super Class Methods

In some cases, the way in which a super class method is implemented is not appropriate for the sub class. That is, *what* the super class method does is appropriate for the sub class, but *how* it does it, is not. In these such cases, you must override the method implementation in the sub class.

To properly override a super class method in the sub class, you must code the method with the same signature (same return value, name, and parameters) and you should add the @Override annotation.

NOTE

An annotation is meta data. That is, data about the code. Annotations may be added immediately above class, field, constructor, or method declarations. The @Override annotation is used by the compiler to ensure the method correctly overrides a method with the same name in the super class.

Consider the following:

```
public class AudioTrack extends Media {
   // instance fields
   private String artist;
   // instance methods
   @Override
   public void play() {
      System.out.println("Playing audio track: " + title);
   }
   @Override
   public void pause() {
      System.out.println("Pausing audio track: " + title);
   }
   public String getArtist() {
      return artist;
   }
   public void setArtist(String artist) {
      this.artist = artist;
   }
}
```

EXAM TIP

Don't confuse overriding with overloading. Overloading is the specification of two or more methods with the same name and different parameter lists. Overriding only applies to inheritance and is the specification of a super class method in the sub class with the same return value, name, and parameter list.

Sub Class Constructors

Constructors are not inherited. This is primarily because the name of a constructor must match the name of the class. For example, it would not make sense for the AudioTrack class to inherit a constructor named Media.

The sub class must include a constructor that matches a constructor in the super class. This is because the sub class IS-A specific type of the super class and so must be created in the same way. For example, if a Media object must be created with a title and a length, then so must an AudioTrack object.

Consider the following refactored Media class:

```
public class Media {
   // instance fields
   private String title;
   private float length;
   // constructor(s)
   public Media(String title, float length) {
      setTitle(title);
      setLength(length);
   }
   // instance methods
   public void play() {
      System.out.println("Playing generic media: " + title);
   }
   public void pause() {
      System.out.println("Pausing generic media: " + title);
   }
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public float getLength() {
      return length;
   }
   public void setLength(float length) {
      this.length = length;
   }
}
```

This change will break the AudioTrack class. Currently, it has only a default constructor added by the compiler which does not specify any parameters. Let's refactor the AudioTrack class:

```
public class AudioTrack extends Media {
   // instance fields
   private String artist;
   // constructor(s)
   public AudioTrack(String title, float length) {
      super(title, length);
   }
   // instance methods
   @Override
   public void play() {
      System.out.println("Playing audio track: " + title);
   }
   @Override
   public void pause() {
      System.out.println("Pausing audio track: " + title);
   }
   public String getArtist() {
      return artist;
   }
   public void setArtist(String artist) {
      this.artist = artist;
   }
}
```

The super keyword refers to the super class. When used in this context it is an invocation of the super class constructor. The first statement in every constructor is an invocation to the identical constructor in the super class, even if you don't code it explicitly (the compiler will add it if you don't). This ensures that the fields inherited from the super class are properly initialised.

NOTE

Many Java developers believe that invoking a super class constructor results in the creation of a super class object. It does not. It simply means that the statements specified in the super class constructor will be executed.

Polymorphism

As stated earlier, polymorphism is a by-product of inheritance. The word means many forms and is the principle that one object may be of many types. That is, an object may contain fields and methods that meet the specification of more than one class.

An AudioTrack object is of both types: AudioTrack and Media. This is because the AudioTrack class inherits from the Media class and, consequently, meets the specification of the Media class. Consider the following:

```
Media track1 = new AudioTrack("O Fortuna", 3.50f);
```

This is a polymorphic declaration, and statements like this occur frequently in production Java code. We will discuss why you might do this shortly but, for now, let's analyse this statement.

track1 is a reference variable of type Media. It references/points to an object of type AudioTrack. This is legal because an AudioTrack object has all the fields and methods a Media object does. Indeed, an AudioTrack object IS-A specific type of Media object.

Here is the critical part:

- The **reference type** dictates **what** can be done
- The **object type** dictates **how** it will be done

As the reference variable is of type Media, only those fields and methods specified in the Media class may be accessed. The following will yield a compilation error:

```
track1.setArtist("Carl Orff"); // setArtist is specified in AudioTrack
```

As the object is of type AudioTrack, the versions of the play and pause methods that will be executed on invocation will be those specified in the AudioTrack class. Invoking the play method using the track1 variable will yield the following output:

```
$> Playing audio track O Fortuna
```

You can exploit polymorphism to build scalable applications. Consider the following:

```
public class Playlist {
   // instance fields
   private AudioTrack[] tracks;
   private int numTracks;
   // static/class fields
   private static final int MAX_TRACKS = 10;
   // constructor
   public Playlist() {
      tracks = new AudioTrack[MAX_TRACKS]; // set array size to 10
      numTracks = 0; // number of tracks in playlist initially
   }
   ...
```

```
    // instance methods
    public void addTrack(AudioTrack track) {
       if(numTracks < MAX_TRACKS) {
          tracks[numTracks++] = track;
       } else {
          System.out.println("Error: playlist full");
       }
    }
    public void playAll() {
       for(int i = 0; i < numTracks; i++) {
          tracks[i].play();
       }
    }
    ...
}
```

There is nothing wrong with this class syntactically, but it does not exploit polymorphism, and it is tightly-coupled with/cannot be separated from the AudioTrack class. Most importantly, it is not scalable. A Playlist object can only deal with AudioTrack objects. What if your client decides she wants to be able to add videos to a playlist too?

Let's refactor the Playlist class to exploit polymorphism:

```
public class Playlist {
    // instance fields
    private Media[] tracks;
    private int numTracks;
    // static/class fields
    private static final int MAX_TRACKS = 10;
    // constructor
    public Playlist() {
       tracks = new Media[MAX_TRACKS]; // set array size to 10
       numTracks = 0; // no tracks in playlist initially
    }
    // instance methods
    public void addTrack(Media track) {
       if(numTracks < MAX_TRACKS) {
          tracks[numTracks++] = track;
       } else {
          System.out.println("Error: playlist full");
       }
    }
    public void playAll() {
       for(int i = 0; i < numTracks; i++) {
          tracks[i].play();
       }
    }
    ...
}
```

The Playlist class now has an array of Media objects. Thanks to polymorphism, AudioTrack objects may be substituted where Media objects are expected. Consider the following use of the Playlist class:


```
AudioTrack track1 = new AudioTrack("Money for Nothing", 3.55f);
AudioTrack track2 = new AudioTrack("Brothers in Arms", 4.20f);
Playlist playlist = new Playlist();
playlist.addTrack(track1);
playlist.addTrack(track2);
playlist.playAll();
```

It works! But what is better, the application is now scalable. A Playlist object can now deal with any Media sub class type object. Even those that have not yet been conceived! And because the **object type** always dictates **how** something is done, different Media sub class type objects may be played in different ways, without making any changes to the Playlist class.

There is, of course, a drawback. As the **reference type** dictates **what** can be done, you cannot directly access sub class specific fields and methods from a reference variable declared polymorphically. In the code example above, you could not directly access the artist field or its associated getter and setter methods using the elements of the tracks array. Each one is of type Media and knows nothing about artists.

Upcasting

Upcasting is the referencing of a sub class type object by a super class type reference variable. Consider again:

```
Media track1 = new AudioTrack("O Fortuna", 3.50f);
```

This is an example of an upcast, and is legal because an AudioTrack object IS-A specific type of Media.

Downcasting

Downcasting is the referencing of a super class type object by a sub class type reference variable. Consider the following:

```
AudioTrack track2 = new Media("O Fortuna", 3.50f); // compilation error
```

This is an example of a downcast, and will result in a compilation error. This is because a Media object is not necessarily an AudioTrack. It might be a Video.

Sometimes downcasting is necessary. For example, when playing a playlist, you might want to display artist information for those Media objects that are AudioTracks.

An explicit cast is a mechanism for forcing the interpreter to perform a downcast. Consider the following:

```
Media myMedia = new AudioTrack("Day for Night", 7.25f);
AudioTrack myAudioTrack = (AudioTrack) myMedia; // explicit cast
myAudioTrack.setArtist("Spock's Beard");
```

The myMedia reference variable type is effectively changed from Media to AudioTrack. It works in this particular case because the object referenced by myMedia is indeed an AudioTrack.

Explicit casting is risky. Consider the following:

```
Media myMedia = new Video("Iron Man", 90.50f);
AudioTrack myAudioTrack = (AudioTrack) myMedia; // runtime error
myAudioTrack.setArtist("Black Sabbath");
```

If the reference variable you are attempting to change does not reference/point to an object of the appropriate type, your application will fail at runtime.

instanceof

The instanceof operator is used to perform explicit casting safely. It may be used to test the type of the referenced object before downcasting the reference variable. Consider the following:

```
if(myMedia instanceof AudioTrack) {
   AudioTrack myAudioTrack = (AudioTrack) myMedia;
   // AudioTrack specific statements
}
```

NOTE

You should always check the type of the referenced object using instanceof before performing an explicit cast.

Abstract Classes and Methods

The Media class includes an implementation of the play and pause methods, but how realistic is this? Can we really specify how Media is played? The answer is no. It depends on the type of Media in question. The Media class should therefore specify *what* the sub classes must do, but *not how* they must do it. The abstract keyword enables just such a relationship.

When applied to a method declaration, the abstract keyword dictates that:

- The method declaration must not include a method body (statements)
- Sub classes must override the method

An abstract method declaration takes the following form:

```
<access_modifier> abstract <return_type> <method_name>(<parameters>);
```

Note that there are no curly braces or statements to be executed. Consider the following refactoring of the Media class.

```
public abstract class Media {
   // instance fields
   private String title;
   private float length;
   // constructor(s)
   public Media(String title, float length) {
      setTitle(title);
      setLength(length);
   }
   // instance methods
   public abstract void play();  // must be overridden by sub classes
   public abstract void pause(); // must be overridden by sub classes
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public float getLength() {
      return length;
   }
   public void setLength(float length) {
      this.length = length;
   }
}
```

The play and pause methods **must** be overridden in the classes that extend Media.

If one or more methods in a class is made abstract, then the class itself must also be made abstract (as is the case in the example code above). An abstract class is one that cannot be instantiated, that is, it cannot be used to create objects. Instead, it must be sub-classed.

There are some rules associated with abstract classes as follows:

- An abstract class cannot be instantiated
- An abstract class may contain none, one, or more abstract methods
- An abstract class may contain none, one, or more concrete/non-abstract methods
- If a class contains one or more abstract methods, the class must be made abstract

NOTE

You may feel that making methods abstract defeats the purpose of inheritance (that is, to inherit behaviour and reduce code duplication), but inheritance as the enabler of polymorphism is far more important. Therefore, you should make methods and classes abstract whenever it is logical to do so. And it makes for good class design to prevent the instantiation of classes that are primarily enablers of polymorphism.

Interfaces

Consider again the Playlist class. Each Playlist object has an array of Media objects, each of which may be played. This is good class design that exploits polymorphism and yields a scalable application. But it is limited in that a Playlist object can only deal with Media sub class objects. What if there are other playable things that don't fall under the Media classification such as an ImageDirectory or a PowerPointShow, both of which extend the File class.

NOTE

Remember that a class may only inherit from one super class.

What we now have is a range of classes in two different groups that we'd like to be able to deal with in a Playlist. As it stands, the only way we can do this is to add another instance field to the Playlist class as follows:

```
private File[] files;
```

But this is bad class design because, presumably, not all files can be played. What we must do is to focus on the behaviour the Playlist object expects of each of its items, rather than the type of item. An interface is the solution to this problem.

An interface is a special type of class that, for the most part, specifies only abstract methods. It is a mechanism for imposing behaviour on a many classes across a range of classifications.

Interface declaration

An interface declaration takes the following form:

```
public interface <interface_name> {
   // abstract methods here
}
```

The Playlist class is interested in play behaviour. Therefore, we need an interface that specifies this required behaviour. Consider the following:

```
public interface Playable {
   // abstract methods
   public abstract void play();
}
```

NOTE

Methods in an interface are both public and abstract by default and so the keywords are typically omitted from methods in the interface declaration.

Any class that implements this interface must exhibit play behaviour, either by overriding the abstract play method, or by inheriting a concrete play method.

Implementing an interface

To implement an interface is to commit to exhibiting the behaviours specified therein. The implements keyword is used in the class declaration to specify that one class should implement an interface. Consider the following:

```
public class PowerPointShow extends File implements Playable {
   // instance methods
   @Override
   public void play() {
      System.out.println("Playing slideshow " + filename);
   }
}
```

A class can implement **one or more** interfaces in addition to inheriting from a super class. If more than one interface is to be implemented, they must be listed after the implements keyword, and separated using commas.

NOTE

The fact that a class can both inherit from a super class and implement interfaces, appears to break Java's single inheritance rule. But this is not so. Java only allows single inheritance to avoid conflicts between identically named fields and methods in two or more super classes. But interfaces cannot have instance fields, and all the methods inherited from an interface are abstract and must be overridden in any case.

Interface polymorphism

The implementation of an interface, like the extending of a super class, represents an IS-A relationship. For example, a PowerPointShow IS-A File AND a Playable. This means that interfaces may be used to exploit polymorphism too. Consider the following:

```
Playable myPlayable = new PowerPointShow("Strategy.pptx");
```

This is a polymorphic declaration. myPlayable is a reference variable of type Playable. It references/points to an object of type PowerPointShow. This is legal because an PowerPointShow object exhibits all the behaviour specified in the Playable interface. Indeed, a PowerPointShow object IS-A specific type of Playable.

So how does all this help with the Playlist problem. Let's refactor each of the classes AudioTrack, Video, ImageDirectory, and PowerPointShow to implement the Playable interface as follows:

```
public class AudioTrack extends Media implements Playable {
   // this class already specifies/inherits a concrete play method
   ...
}
```



```
public class Video extends Media implements Playable {
   // this class already specifies/inherits a concrete play method
   ...
}
public class ImageDirectory extends File implements Playable {
   // instance methods
   @Override
   public void play() {
      System.out.println("Playing images in dir " + dir);
   }
}
public class PowerPointShow extends File implements Playable {
   // instance methods
   @Override
   public void play() {
      System.out.println("Playing slideshow " + name);
   }
}
```

Now, let's refactor the Playlist class to deal with Playables:

```
public class Playlist {
   // instance fields
   private Playable[] tracks;
   private int numTracks;
   // static/class fields
   private static final int MAX_TRACKS = 10;
   // constructor
   public Playlist() {
      tracks = new Playable[MAX_TRACKS]; // set array size to 10
      numTracks = 0; // no tracks in playlist initially
   }
   // instance methods
   public void addTrack(Playable track) {
      if(numTracks < MAX_TRACKS) {
         tracks[numTracks++] = track;
      } else {
         System.out.println("Error: playlist full");
      }
   }
   public void playAll() {
      for(int i = 0; i < numTracks; i++) {
         tracks[i].play();
      }
   }
   ...
}
```

Now a Playlist object can deal with anything that can be played, Media or otherwise. This type of class design is often referred to as coding to the interface.

The Object Class

Unless you explicitly specify a super class, your class will implicitly inherit state and behaviour from a class named Object. The Object class exists in the java.lang package and is the root class for all other classes.

Earlier in this chapter we described a Media super class and an AudioTrack sub class. But what is the Media class's super class? The answer is the Object class. The Object class is the only class in the Java API not to have a super class.

The state and behaviour specified in the Object class is indirectly inherited by every Java class ever written. Take the Media and AudioTrack classes for example. The Media class inherits state and behaviour from the Object class. And the AudioTrack class inherits state and behaviour from the Media class – which include that inherited from Object.

Whilst the name – Object – may seem confusing at first, it actually makes good sense. Recall that inheritance describes an IS-A relationship. Given that every Java class inherits state and behaviour from the Object class (either directly or indirectly), every object created IS-A Object (naturally).

The Object class doesn't have any instance fields, static fields, or static methods. It only has instance methods.

Object class instance methods

The Object class has the following instance methods:

- clone() : Object
- **equals(Object obj) : boolean**
- finalize() : void
- getClass() : Class<?>
- **hashCode() : int**
- notify() : void
- notifyAll() : void
- **toString() : String**
- wait() : void
- wait(long timeout) : void
- wait(long timeout, int nanos) : void

Of these, you will typically override the equals, hashCode, and toString methods in every class modelling a real world thing that you write.

The clone method may be used in conjunction with the Cloneable interface to clone objects, but this is beyond the scope of this course.

The finalize method is invoked on an object prior to its being garbage collected.

NOTE

It is possible an object that is eligible for garbage collection is never actually collected, and so the finalize method may never be invoked.

The getClass method is used in reflection and is beyond the scope of this course.

The notify and wait methods are used only in multi-threaded applications and are beyond the scope of this course.

equals and hashCode

The equals method is used to compare two object references.

NOTE

The equals method as inherited from the Object class compares two object references using the equality operator (==). This compares memory addresses, not object content.

The equals method should be overridden to test if the content of two objects is the same. Business rules will dictate what constitutes two equivalent objects of a particular type. Consider the following:

```
Account a1 = new Account("Harrison", 531420);
Account a2 = new Account("Thompson", 531420);
```

Are the two Account objects referenced by a1 and a2 equal? It depends on the business rules. If business rules dictate that both account names and numbers must match to constitute equivalent accounts, then the two objects are not equal. If, however, business rules dictate that only the account numbers must match to constitute equivalent accounts, then the two object are equal. It is for this reason that you should override the equals method in every class modelling a real world thing that you write.

The hashCode method calculates and returns a seemingly random integer that represents the object.

NOTE

The hashCode method as inherited from the Object class calculates and returns an integer based on the object's memory address. If you override the equals method, you must also override the hashCode method.

The hashCode of a given object should be calculated using the same fields used to test for equality. Using the example above, if two Account objects are to be considered equal if the account numbers match, then the hashCode for an Account object should be calculated using the account number only. To put it another way, two Account objects that are equal should yield the same hashCode.

Both the equals and hashCode methods are used by hashing collections such as HashMap and Hashtable. Hashing collections are beyond the scope of this course but you should know that if you fail to override these methods in the appropriate classes, your application may not work as expected.

You can have Eclipse override these methods for you in a standard way by following these simple steps:

1. Right-click anywhere in your class
2. Select Source, then Generate hashCode() and equals()…
3. In the resultant dialog, select the fields that should be used to test for equality and to calculate the hash code, and then click OK

NOTE

Remember that business rules must dictate which fields you select when having Eclipse automatically generate the hashCode and equals methods for you.

toString

The toString method returns a String representation of an object.

NOTE

The toString method as inherited from the Object class returns the object's memory address as a String.

The toString method is invoked automatically by Java when you write or print an object reference. Consider the following:

```
Account a1 = new Account("Harrison", 531420);
System.out.println(a1); // same as System.out.println(a1.toString());
```

It is common practice to override the toString method to return a String that represents the content of the object, and not its memory address.

You can have Eclipse override the toString method for you in a standard way by following these simple steps:

1. Right-click anywhere in your class
2. Select Source, then Generate toString()…
3. In the resultant dialog, select the fields and or methods that should be included in the return value, and then click OK

NOTE

By convention, you would only select fields, and not methods, to be included in the toString method's return value.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Case Study – Part 5

In addition to books, the library maintains collections of CDs and children's toys for loan. One way to represent this in code would be to add instance fields to the Library class. Consider the following:

```
public class Library {
   // instance fields
   private String name;
   private Book[] books;
   private CD[] cds;
   private Toy[] toys;
   ...
```

This solution tightly couples the Book, CD, and Toy classes with the Library class. This is not necessarily a problem, but consider the following:

- What if the library decides not to stock children's toys any more?
- What if the library decides to stock DVDs?

Both scenarios require changes to be made to the Library class. Inheritance and polymorphism can help to protect against such changes. But is traditional inheritance the solution in this case? Do books, CDs, and toys share common state and behaviour? If so, what is the super class? You might argue that they are all things, but that's too generic. Everything is a thing! In the context of a library, each book, CD, and toy is loanable. It is the check out and check in behaviour that is common.

1. Create a new, fully encapsulated class named CD.
2. Add instance fields to represent the CD's title, artist, and release year.
3. Add a constructor that accepts a title, artist, and release year.
4. Add getters and setters for each instance field.
5. Have Eclipse override the equals, hashCode, and toString instance methods inherited from the Object class. Two CDs are equal if they have the same title and artist.
6. Save your code.
7. Create a new, fully encapsulated class named Toy.
8. Add instance fields to represent the toy's description and age appropriateness.
9. Add a constructor that accepts a description and age.
10. Add getters and setters for each instance field.
11. Have Eclipse override the equals, hashCode, and toString instance methods inherited from the Object class. Two toys are equals if the have the same description and age appropriateness.
12. Save your code.

13. Create a new abstract class named Loanable.
14. Add the following instance field:

```
protected boolean onLoan;
```

15. Add the following abstract instance method:

```
String getIdentifier();
```

16. Copy the checkOut, checkIn, and isOnLoan methods from the Book class into the Loanable class.
17. Replace the references to the name instance field with invocations of the getIdentifier method.
18. Save your code.
19. Change the Book class such that it extends the abstract Loanable class.
20. Remove the onLoan instance field, checkIn and isOnLoan methods.
21. Have Eclipse override the equals, hashCode, and toString instance methods inherited from the Object class. Two books are equal if they have the same name and author.
22. Override the inherited getIdentifier instance method such that it returns the book's name.
23. Save your code.
24. Change the CD class such that it extends the abstract Loanable class.
25. Override the getIdentifier instance method such that it returns the CD's title.
26. Save your code.
27. Change the Toy class such that it extends the abstract Loanable class.
28. Override the getIdentifier instance method such that it returns the toy's description.
29. Save your code.
30. Change the Library class such that it has an array of Loanables named items, instead of an array of Books named books.
31. Change the name of the instance field numBooks to numItems.
32. Change name of the constant MAX_BOOKS to MAX_ITEMS.
33. Change the find instance method such that it accepts a String named identifier and returns a Loanable.
34. Change the add instance method such that it accepts a Loanable named item.
35. Change the name of the getBooks instance method to getItems, and change its return type from Book[] to Loanable[].
36. Change the name of the getNumBooks instance method to getNumItems.
37. Save your code.
38. Change the LibraryTest class to account for the changes made to the Library class.
39. Add some CDs and toys to your library. Ensure that you can find, check out, and check in all Loanable types.
40. Save and execute your code.

CHAPTER 14

Exception Handling

Introduction

An exception is an error that, if not dealt with, will crash your application. Exception handling is a mechanism for dealing with such exceptions. That is, it provides a way for you to respond when something goes wrong and to recover. What is more, exception handling allows you to choose the point at which to respond, which needn't be the point at which the exception was generated.

Checked and Unchecked Exceptions

Checked Exceptions

A checked exception is one that must be dealt with. That is, you must either throw it on to the calling method to deal with, or catch it and handle it in the current method. Checked exceptions represent errors that result from bad user input or changes to the application environment. The following scenarios are likely to yield checked exceptions:

- Attempting to parse user input where the user was expected to enter a number, but instead entered non-numeric characters
- Attempting to read a file on which permissions have been changed
- Attempting to connect to a remote machine that has been disconnected from the network or switched off

None of these errors are the fault of the developer, that is, there may be nothing wrong with the code, and yet the application still crashes. But these types of errors are **anticipatable**. It is the developer's responsibility to consider all the possible scenarios and to code accordingly to protect against application failure.

Checked exceptions are so named because the compiler will check lines of code that may yield such exceptions for associated exception handling code.

NOTE

The Exception class and all its sub classes are checked exception types.

Unchecked exceptions

An unchecked exception is one that needn't be dealt with. Unchecked exceptions represent errors that result from developer mistakes. The following scenarios are likely to yield unchecked exceptions:

- Attempting to invoke a method on a null reference
- Attempting to read or write to non-existent array indices
- Attempting an illegal downcast

These errors are the fault of the developer. The solution is to fix the code, not to add exception handling code.

Unchecked exceptions are so named because the compiler will not check lines of code for such exceptions. Indeed, it is impossible for the compiler to anticipate such errors.

NOTE

The RuntimeException class and all its sub classes are unchecked exception types.

Throwing Exceptions

To throw an exception is to:

1. Create an object of type Exception (or one of its subclasses)
2. Pass the Exception object to the calling method

The throwing of an exception from within a method takes the following form:

```
throw new <exception_type>(<error_message>);
```

The method signature must also reflect the fact that an exception may be thrown by appending the throws keyword and the exception type to the end of the method signature. Consider the following:

```
public String[] readLines(String filename) throws IOException {
   // code here may throw an IOException object
   ...
}
```

It is appropriate to throw an exception when:

- Something unexpected happens, and…
- The problem can't be dealt with in the current method

Consider the setter methods in the following Employee class:

```
public class Employee {
   // instance fields
   private String name;
   private double salary;
   // constructor
   public Employee(String name, double salary) {
      setName(name);
      setSalary(salary);
   }
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public void setName(String name) {
      if(name != null && !name.equals("")) {
         this.name = name;
      }
   }
   public void setSalary(double salary) {
      if(salary >= 10_000 && salary <= 90_000) {
         this.salary = salary;
      }
   }
}
```

The setter methods only set the instance fields if the arguments are valid. Now consider the following usage of the Employee class:

```
public class EmployeeTest {
   public static void main(String[] args) {
      // prompt the user to enter a name and a salary
      Employee myEmployee = new Employee(name, salary);
      // code that makes use of myEmployee
      ...
   }
}
```

The main method calls the Employee class constructor, which in turn calls the setName and setSalary methods. Let's assume the user sets the name to "David" and the salary to 7000. The name argument – "David", is valid and so is used to set the name instance field of the new Employee object. The salary argument – 7000, is invalid (salary must be in the range 10,000 – 90,000) and so is not used to set the salary instance field. What, therefore, is the value of the new Employee object's salary instance field? Remember that instance fields are assigned default values, and each numeric type instance field is assigned zero by default. This means that, by not setting the salary instance field, it has a value of zero – which is invalid! We must prevent the creation of an Employee object if either name or salary argument is invalid.

We could generate or return an error message from within the setter method, but that is not satisfactory for two reasons:

1. It will not prevent the creation of an Employee object
2. Return values should never be used for error messages

The problem cannot be dealt with in the setter method. It must instead be dealt with at the point at which the user enters the name and salary. In this case, that means dealing with the error in the main method.

First, we must refactor the Employee class to throw exceptions:

```
public class Employee {
   // instance fields
   private String name;
   private double salary;
   // constructor
   public Employee(String name, double salary) throws Exception {
      setName(name);
      setSalary(salary);
   }
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   ...
```



```
    public void setName(String name) throws Exception {
       if(name != null && !name.equals("")) {
          this.name = name;
       } else {
             throw new Exception("Name cannot be empty");
       }
    }
    public void setSalary(double salary) throws Exception {
       if(salary >= 10_000 && salary <= 90_000) {
          this.salary = salary;
       } else {
             throw new Exception("Salary must be in the range 10-90K");
       }
    }
}
```

In the event the setName or setSalary methods are called with invalid arguments, each will throw an Exception object to the calling method.

NOTE

The constructor is also configured to throw Exception objects. This is because it calls the setName and setSalary methods.

Effectively the Employee class is passing the responsibility for ensuring valid values outside of the class to whichever class is making use of it.

Catching Exceptions

To catch an exception is to:

- Wrap the code that may yield an exception in a try block
- Add code to deal with the exception in a catch block associated with the try block

The catching of an exception takes the following form:

```
try {
   // call a method/constructor that may throw an exception
} catch(<exception_type> <variable_name>) {
   // code to deal with the exception
}
```

Let's refactor the EmployeeTest class to catch exceptions:

```
public class EmployeeTest {
   public static void main(String[] args) {
      // prompt the user to enter a name and a salary
      Employee myEmployee = null;
      try {
         myEmployee = new Employee(name, salary);
      } catch(Exception e) {
            System.out.println(e.getMessage());
            System.exit(1); // terminate the application
      }
      // code that makes use of myEmployee
      ...
   }
}
```

Let's consider the code step-by-step:

- The myEmployee reference variable is declared outside of the try block and initialised to null. This is necessary such that the Employee object is accessible outside of the try block.
- The code that may throw an exception – the calling of the Employee constructor – is wrapped in a try block.
- Code to deal with the exception (if indeed an exception is thrown) is added to a catch block immediately following the try block. The catch block is configured to catch exception objects of type Exception.
- If an exception is thrown, the code inside the catch block is executed.
- The variable named e references the Exception object thrown by the Employee class.
- Every object of type Exception or one of its subclasses has an instance method named getMessage. This method may be used to access the error message associated with the Exception object – in this case the error message is printed to the console.

- In this case, following the printing of the error message to the console, the application is terminated by invoking the static exit method of the System class. This is because we do not have a valid Employee object, and carrying on with an invalid object is likely to cause problems elsewhere in the application.
- If no exception is thrown, the catch block is skipped and the code beneath the catch block is executed as normal.

NOTE

Terminating the application is a last-resort way of dealing with an exception. Alternatively, we could have added a loop and prompted the user to enter the employee's name and salary again.

One try block, many catch blocks

It is possible to associate many catch blocks with a try block. This may be necessary if you want to deal with different types of exceptions in different ways. Consider the following:

```
String[] lines = null;
try {
   lines = readLines("stocks.csv");
} catch(FileNotFoundException fnfe) {
   // code to deal with file not found error
} catch(IOException ioe) {
   // code to deal with read error
}
```

NOTE

Catch blocks must be ordered from most specific to most least specific. In the example code above, FileNotFoundException is a sub class of IOException.

finally!

A finally block may be added after the catch block(s). Code inside the finally block will be executed regardless of whether an exception is thrown. Historically, the finally block was used to close resources such as file handles and network and database connections. Consider the following:

```
try {
   // code to connect to a remote machine
} catch(Exception e) {
   // exception handling code
} finally {
   // code to close the connection
```

```
    }
```

Unfortunately, it is often the case that methods to close resources themselves may throw exceptions. This meant developers had to nest a try catch block inside the finally block! Thankfully this is no longer the case, as you will see in the next section.

Try with Resources

With regards to exception handling, a resource is an object that is associated with something external to the application, like a file or a database. Resources enable you to write code to read/write from files and databases but must be closed when you're finished with them.

Prior to Java version 7, it was considered good practice to close resources in a finally block, so that regardless of whether the read/write operation resulted in an exception, the resource would be closed. For many resources, however, the close operation itself may throw an exception and must therefore be handled. This resulted in some rather ugly code. Consider the following:

```
FileInputStream in = null;
try {
   in = new FileInputStream("stocks.csv");
   int data = in.read();
   while(data != -1) {
      System.out.print((char) data);
      data = in.read();
   }
} catch(FileNotFoundException fnfe) {
   System.out.println(fnfe.getMessage());
} catch(IOException ioe) {
   System.out.println(ioe.getMessage());
} finally {
   try {
      in.close();
   } catch(IOException ioe) {
      // ignore
   }
}
```

Note that the closing of the FileInputStream object within the finally block is enclosed within a try catch block.

The try with resources construct was added to Java in version 7. It provides for the automatic closing of resources. A try with resources block takes the following form:

```
try(<resource declaration and initialisation>) {
   // call a method that may throw an exception
} catch(<exception_type> <variable_name>) {
   // code to deal with the exception
}
```

The reading of a file using a FileInputStream resource as per the example code above might be refactored using the try with resources construct as follows:

```
try(FileInputStream in = new FileInputStream("stocks.csv")) {
   int data = in.read();
   while(data != -1) {
      System.out.println((char) data);
      data = in.read();
   }
} catch(FileNotFoundException fnfe) {
   System.out.println(fnfe.getMessage());
} catch(IOException ioe) {
   System.out.println(ioe.getMessage());
}
```

Because the resource is declared and initialised as part of the try block, Java will close it automatically regardless of whether an exception is thrown.

NOTE

Only those resources (classes) that implement the AutoCloseable interface may be used in a try with resources block.

Custom Exceptions

It is not good practice to throw exceptions of type Exception. The Exception class is the root class of all checked exceptions and is too generic in most cases. You should instead throw an exception that is specific to the problem. That is, the exception class name should specifically and accurately describe what went wrong.

Sometimes you will want to create your own exception classes to describe business specific problems. Consider again the example used previously in this section:

```
public class Employee {
   // instance fields
   private String name;
   private double salary;
   // constructor
   public Employee(String name, double salary) throws Exception {
      setName(name);
      setSalary(salary);
   }
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public void setName(String name) throws Exception {
      if(name != null && !name.equals("")) {
         this.name = name;
      } else {
            throw new Exception("Name cannot be empty");
      }
   }
   public void setSalary(double salary) throws Exception {
      if(salary >= 10_000 && salary <= 90_000) {
         this.salary = salary;
      } else {
            throw new Exception("Salary must be in the range 10-90K");
      }
   }
}
```

Note that the constructor and setter methods may throw exceptions of type Exception. Let's assume this class forms part of a large application where the other developers working on the project are oblivious to the presence of the Employee type. If an exception is thrown, the cause of the error is not immediately obvious. This could be overcome, however, if the Employee class were coded to throw exceptions of type EmployeeException – a custom type.

To create a custom exception, simply create a class that extends Exception and add a constructor. Consider the following example:

```
public class EmployeeException extends Exception {
   // constructor
   public EmployeeException(String message) {
      super(message);
   }
}
```

NOTE

By convention, custom exception classes should be named to end with the word Exception.

The Employee class may now be refactored to throw exceptions of type EmployeeException, thus making it easier for your fellow developers to identify the source of any problem.

```
public class Employee {
   // instance fields
   private String name;
   private double salary;
   // constructor
   public Employee(String name, double salary) throws EmployeeException {
      setName(name);
      setSalary(salary);
   }
   // instance methods
   public String getName() {
      return name;
   }
   public double getSalary() {
      return salary;
   }
   public void setName(String name) throws EmployeeException {
      if(name != null && !name.equals("")) {
         this.name = name;
      } else {
            throw new EmployeeException("Name cannot be empty");
      }
   }
   public void setSalary(double salary) throws EmployeeException {
      if(salary >= 10_000 && salary <= 90_000) {
         this.salary = salary;
      } else {
            throw new EmployeeException("Salary must be in the range 10-90K");
      }
   }
}
```


Guidelines

The following guidelines apply to the writing of exception handling code:

- Don't throw exceptions of type Exception. Use a more specific exception class or create a custom exception class.
- Always throw an exception of the type most specific to the problem.
- If the code within your try block may throw more than one type of exception, arrange the catch blocks from most specific type to least specific type.
- Always include a useful String error message when creating an exception object.
- Never throw exceptions from the main method. Doing so is tantamount to allowing your application to crash.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Case Study – Part 6

One of your colleagues has developed a GUI for the library app you're working on. It should work well with the classes you've developed provided the user doesn't do anything 'unexpected'.

1. Delete LibraryTest.java.
2. Your trainer will provide you with the LibraryUI.java source code file. It has a main method and uses the classes you've developed – Loanable, Book, CD, Toy, and Library. Add it to your project and execute it.
3. Create a new book – Item | New | Book – as follows:
   a. Name: Sapiens
   b. Author: Yuval Noah Harari
   c. Genre: History
4. Add the new book to the library – Item | Save.
5. Create a new CD – Item | New | CD – as follows:
   a. Title: Villains
   b. Artist: Queens of the Stone Age
   c. Release year: 2017
6. Add the new CD to the library – Item | Save.
7. Create a new toy – Item | New | Toy – as follows:
   a. Description: Building blocks
   b. Age: 1
8. Add the new toy to the library – Item | Save.
9. Select the CD you added in the list of items and check it out – Item | Check out.
10. Select Item | Find, enter Sapiens in the dialog, and then click OK.
11. Select the CD you added in the list of items and check it in – Item | Check in.

    All appears to be working well. But we've not tested the application rigorously. That is, we've assumed the end user knows the business rules. He/she does not.

12. Create a new book – Item | New | Book – but don't enter any values.
13. Add the new book to the library – Item | Save.

    A new book is added to the library with a null name and empty author. This should not have happened. The Book class constructor accepts a name, author, and genre, and invokes the associated setter methods. Only the setName method contains validation code, but does nothing if the name is invalid.

14. Create a new class named BookException that inherits from Exception.
15. Refactor the Book class's setName and setAuthor instance methods such that if the name or author parameter is null or empty, a BookException with a

meaningful message is thrown. Throw the exception from the Book class's constructor too.

16. Save your code.

    Your changes above will have broken the LibraryUI code. The creation of a Book object may generate a BookException, and so must be handled. The LibraryUI class is the best place to handle user input related errors, as it is from here that you can prompt the user to try again.

17. Scroll through the LibraryUI.java source code until you find the line (no. 57) that no longer compiles. Surround it with a try catch block.
18. In the catch block, display a dialog as follows:

    ```
    JOptionPane.showMessageDialog(this, e.getMessage());
    ```

    Note that we are displaying the error message contained within the BookException object – e.getMessage().

    It is not enough to simply surround the non-compiling line of code with a try catch block. Remember that the interpreter carries on executing the code that follows the catch block. In this case, the next three lines are:

    ```
    library.add(book);
    populateItemList();
    clearAllFields();
    ```

    But we don't want any of this to happen if a Book object was not created due to invalid user input.

19. Move the three lines of code listed above into the try block. This way, if an exception is thrown creating a Book object, these three lines will be skipped.
20. Save and execute your code.
21. Try saving a new book with invalid values. It should not be possible.
22. Time permitting, add code to throw exceptions of type LibraryException from the Library class's add instance method. Add appropriate exception handling code to the LibraryUI class, and then save and test your changes.

CHAPTER 15

Packages and Imports

Introduction

Most Java applications are composed of many classes. In some cases, an application may even comprise two classes with the same name. For example, the development team working on the HR module might create a class to represent an employment contract named Contract, while the team working on the Sales module might create an identically named class to represent the terms of an agreement with a client.

Packages and imports are the constructs that provide for the organisation and use of classes within and between Java applications.

Packages

A package is a directory containing Java classes. All classes within the package must reference the package name using the package keyword and this statement must be the first in the source code file (excluding comments). Consider the following:

```
package sales;
public class Contract {
   // TODO
}
```

The Contract class must exist inside a directory named sales. Let's assume the Contract class has a main method and so is executable. Consider the following command:

```
$> java Contract
```

If executed within the sales directory this command will yield an exception as follows:

```
$> Error: Could not find or load main class Contract
```

This is because the package name now forms part of the way the class is accessed. That is, the class's fully-qualified name is sales.Contract. Consider the following command:

```
$> java -classpath <sales_dir_parent_dir> sales.Contract
```

To execute a class within a package you must set the classpath to the parent directory of the package, and then reference the class using its fully-qualified name.

NOTE

By default, the classpath is set to the current directory. Note too that the classpath may comprise many distinct locations each separated by a semicolon e.g.

```
$> java -classpath dir1;dir2;dir3 package.Class
```

Naming conventions

By convention, packages should incorporate your company's domain name in reverse. Consider the following:

```
package com.stayahead.sales;
```

This is to distinguish all Java classes globally and enables the sharing of code between companies without having to worry about naming conflicts.

Each part of the example package name above is a separate directory. The sales directory exists inside the stayahead directory which in turn exists inside the com directory. This creates a hierarchical structure of logically grouped code.

Java vs. third-party packages

Packages that begins with the word java are accessible to your application natively. That is, you don't have to add any files/libraries to your project to use them. These packages form part of the Java Runtime Environment (JRE) which is needed to execute a Java application.

Packages that are developed by third parties (those that do not begin with the word java) are not accessible to your application natively. That is, you must add the files/libraries to your project to use them. Furthermore, you must make the third-party files/libraries accessible on the classpath.

EXAM TIP

The use of third party libraries does not form part of the exam.

The Java API

The Java API (application programming interface) is the suite of packages and classes available as part of the JRE. It includes classes for building collections, performing mathematical computations, file I/O, database access, networking, security, building GUIs and lots more. Documentation for the Java 8 API is available at: https://docs.oracle.com/javase/8/docs/api/

Imports

Let's assume you're working on the Contract class within the com.stayahead.sales package and you want to make use of the CurrencyExchange class which exists within the com.stayahead.utils package. One way to do this is to reference the class using its fully-qualified name. Consider the following:

```
package com.stayahead.sales;
public class Contract {
   ...
   // instance methods
   public double getPriceInLocalCurrency() {
      double localPrice =
         com.stayahead.utils.CurrencyExchange.convert(price, GB, US);
      return localPrice;
   }
}
```

The static convert method in the CurrencyExchange class may be accessed using the class's fully-qualified name, but this requires writing a significant amount of additional code, particularly if the class in question is used many times. Consider the instantiation of a CurrencyExchange object:

```
com.stayahead.utils.CurrencyExchange exchange =
   new com.stayahead.utils.CurrencyExchange();
```

The import construct enables the use of classes in other packages without having to reference them using their fully-qualified names. An import statement must appear after the package declaration and before the class declaration, and takes the following form:

```
import <fully_qualified_class_name>;
```

Let's refactor the code above to include an import statement:

```
package com.stayahead.sales;
import com.stayahead.utils.CurrencyExchange;
public class Contract {
   ...
   // instance methods
   public double getPriceInLocalCurrency() {
      double localPrice = CurrencyExchange.convert(price, GB, US);
      return localPrice;
   }
}
```

The CurrencyExchange class may now be accessed using the class name only.

NOTE

Contrary to its name, an import statement doesn't result in the import of anything. It simply helps the compiler/interpreter to locate the code used by the class in question.

A class may contain many import statements. Indeed, there is no theoretical limit on the number of classes you may import.

Wildcards

A wildcard (*) may be used if you want to access many classes in the same package. Consider the following:

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

This import statement allows for the use of any class within in the java.util package and can save writing many import statements.

Whilst the wildcard can save typing, it makes for code that is harder to maintain. Consider the following:

```
package com.stayahead.sales;
import java.util.*;
import java.text.*;
import java.math.*;
public class Contract {
   ...
   // instance methods
   public String getFormattedPrice() {
      DecimalFormat decimalFormat = new DecimalFormat("#0.##");
      return decimalFormat.format(price);
   }
}
```

There is no way of knowing (unless you're very familiar with the Java libraries) which package contains the DecimalFormat class.

NOTE

Some developers oppose the use of the wildcard in import statements because they believe, falsely, that it bloats the project with lots of unused classes. But recall that an import statement simply helps the compiler/interpreter to locate the code used by the class in question. It has no effect on the overall size of the project, even when wildcards are used.

java.lang

You may have noticed that within the classes you've written in the course thus far, you've used several Java classes without adding an import statement. Consider the following:

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

This simple class uses the String and System classes without an import statement. This is because every Java application includes access to the classes within the java.lang package by default (without an import statement). And it so happens that the String and System classes reside in the java.lang package.

Static Imports

A static member (field or method) is one that belongs to the class (template) and not to each object created using that template. Static members should be accessed using the class name. Consider again the following:

```
package com.stayahead.sales;
import com.stayahead.utils.CurrencyExchange;
public class Contract {
   ...
   // instance methods
   public double getPriceInLocalCurrency() {
      double localPrice = CurrencyExchange.convert(price, GB, US);
      return localPrice;
   }
}
```

We know that the convert method is static because it is accessed using the class name – CurrencyExchange – and not an object of said class.

The number of classes in the Java API containing many static methods has increased significantly in the last few versions (the reasons for which are beyond the scope of the course). Static imports enable the invocation of static methods without having to reference the class name. This can be useful where a static method is used over and over in your code.

A static import takes the following form:

```
import static <fully_qualified_class_name>.<static_method>;
```

Or, to enable access to all static methods in the class:

```
import static <fully_qualified_class_name>.*;
```

Let's refactor the Contract class to use a static import:

```
package com.stayahead.sales;
import static com.stayahead.utils.CurrencyExchange.convert;
public class Contract {
   ...
   // instance methods
   public double getPriceInLocalCurrency() {
      double localPrice = convert(price, GB, US);
      return localPrice;
   }
}
```

Note that the import statement is static and includes the static method name, and that the static method convert is now accessed without referencing the class name.

NOTE

Overuse of static imports make your code hard to maintain insofar as it makes it difficult to differentiate instance methods from static methods. You should only use a static import if you intend to use a static method of an imported class many times.

CHAPTER 16

Selected APIs

Introduction

This section describes the small set of Java API classes which appear in the Oracle Java SE 8 Programmer I exam.

Wrapper classes play an important role in the storage of numbers in collections and are the mechanism by which primitive numeric variables are derived from String objects.

The String class is probably the most commonly used data type in the Java language. It includes a suite of useful methods for String manipulation.

The StringBuilder class is used to concatenate Strings and other data types in a manner that is more memory efficient than the concatenate (+) operator.

Introduced in Java version 8, the classes in the java.time package allow for the representation of dates and times and supersede the date and time classes in the java.util package.

The ArrayList class is just one of dozens of collection classes but is one of the most commonly used. An ArrayList is like an array but can grow and shrink as needed.

Wrapper Classes

Each primitive data type has an associated wrapper class as follows:

Primitive type	Wrapper class
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
boolean	java.lang.Boolean
char	java.lang.Character

Each wrapper class has an instance field of the associated primitive type. In effect, a wrapper class object *wraps* a primitive variable in an object. Consider the following:

```
public class Integer {
   // instance fields
   private int i;
   // constructor
   public Integer(int i) {
      this.i = i;
   }
   ...
}
```

NOTE

Collection classes, unlike arrays, do not allow for the storage of primitive variables. To store simple numbers in a collection, they must be wrapped in an object.

Autoboxing and autounboxing

Java allows for the automatic conversion between primitive variable and wrapper class object. Consider the following:

```
Double wrapperDouble = 1.5d;
```

The primitive literal 1.5d is automatically converted to a Double wrapper object. This process is known as autoboxing. The reverse, autounboxing, also works:

```
double primitiveDouble = wrapperDouble;
```


Converting Strings to numbers

Each wrapper class includes a static parse method to derive a primitive variable from a String object. This is very useful given that most UI data is read as a string of characters, regardless of what the user enters. Consider the following:

```
import java.util.Scanner;
public class AgeCalculator {
   public static void main(String[] args) {
      System.out.print("Enter your year of birth: ");
      try(Scanner keyboard = new Scanner(System.in)) {
         String stringBirthYear = keyboard.nextLine();
         int intBirthYear = Integer.parseInt(stringBirthYear);
         int age = 2017 – intBirthYear;
      } catch(NumberFormatException e) {
            System.out.println("You did not enter a valid number");
            System.exit(1);
         }
      System.out.println("You are " + age + " years old?");
      keyboard.close();
   }
}
```

The nextLine method of the Scanner object returns a String. The static parseInt method of the Integer class is used to derive a primitive int variable from the String object.

NOTE

The parseInt method will throw a NumberFormatException if a primitive int variable cannot be derived from the String object.

String

The String class is probably the most commonly used data type in the Java language and is rather unique because:

- A String variable can be initialised like a primitive
- A subset of Java memory is reserved just for String objects
- String objects are immutable

Initialisation

A String variable may be initialised in one of two ways. Consider the following:

```
String s1 = new String("Hello World");
String s2 = "Hello World";
```

Each line of code above yields a String object. For many Java developers, the second line is simply a shorthand version of the first. But this is quite wrong.

When a String object is created using the new keyword, the object exists in normal Java memory and will be eligible for garbage collection when dereferenced. Consider the following:

```
String name = new String("David");
...
name = null;
```

When the variable name is set to null, the String object it previously referenced is eligible for garbage collection. That is, the memory it occupies may be made available for use by some other object. This applies to all objects, not just those of type String.

When a String object is created without using the new keyword, the object exists in a special subset of Java memory reserved just for Strings known as the String Pool. Objects in the String Pool are not garbage collected in the usual way. Instead, the objects in the String Pool are retained in memory on the assumption that they will be used again at some point. Consider the following:

```
String language = "Java";
...
language = null;
```

When the variable language is set to null, the String object it previously referenced is retained in the String Pool on the assumption it will be used again.

If the String object you create will be used over and over in your code, initialise the String variable without the new keyword. This will save Java having to create and subsequently garbage collect many objects.

If the String object you create will only be used once, initialise the String variable with the new keyword. The will preserve memory, that is, the String object will not be retained in the String Pool indefinitely.

NOTE

Developers new to Java often mistake String variables as primitives because a String object can be created without the new keyword. But it doesn't matter how you initialise your String variable, it will reference a String object.

Immutability

Consider the following:

```
String name = "Stephen";
name = "Steve";
```

The String object referenced by the variable name was changed from "Stephen" to "Steve", right? Wrong. String objects are immutable. This means that once a String object is created it can never be changed. In the example code above, the String variable name is initially set to reference a String object containing "Stephen". On the next line, the String variable name is set to reference another String object containing "Steve".

NOTE

The String objects containing "Stephen" and "Steve" may have been created or, given the initialisation does not include the new keyword, may have been reused from the String Pool.

String objects are immutable for several reasons, one of which is that a String is comprised of an array of characters. As you know, arrays are fixed size and so changing a String object would, in most cases, require the creation of a new array.

NOTE

Any String method that appears to change a String object does not. Instead, it creates a new String object and returns a reference to it.

Methods

The String class includes many useful instance methods for manipulating String objects. The table below lists some of the more commonly used String methods with example code assuming a String variable initialised as follows:

```
String myString = "Hello World";
```

method(parameters) : return	Example usage
charAt(int index) : char	char c = myString.charAt(4); // c = 'o'
contains(String s) : boolean	boolean b = myString.contains("World"); // b = true
endsWith(String suffix) : boolean	boolean b = myString.endsWith("Java"); // b = false

method(parameters) : return	Example usage
indexOf(String str) : int	int i = myString.indexOf("World"); // i = 6
length() : int	int i = myString.length(); // i = 11
replace(String a, String b) : String	String s = myString.replace("World", "Java"); // s = "Hello Java"
split(String regex) : String[]	String[] s = myString.split(" "); // a = ["Hello", "World"]
startsWith(String prefix) : boolean	boolean b = myString.startsWith("Hello"); // b = true
substring(int begin, int end) : String	String s = myString.substring(1, 4); // s = "ell"
toLowerCase() : String	String s = myString.toLowerCase(); // s = "hello world"
toUpperCase() : String	String s = myString.toUpperCase(); // s = "HELLO WORLD"

Concatenation

Strings may be joined together using the concatenate operator (+). Consider the following:

```
String firstName = "John";
String lastName = "Smith";
System.out.println(firstName + lastName); // John Smith
```

As you know, the + operator is also used for addition. Java knows if the + operator should be used to concatenate or add depending on the context. If either operand is a String, then Java will concatenate the two operands. Consider the following:

```
String s = "The meaning of life is ";
int x = 42;
System.out.println(s + x); // The meaning of life is 42
```

You will recall that expressions are evaluated from left to right. Consider the following:

```
System.out.println(5 + 5 + "5");
```

The expression passed to the println method is evaluated from left to right. Java first evaluates 5 + 5 (addition) and then concatenates the result with the String literal "5" yielding "105".

NOTE

Anything concatenated with a String yields a String.

Concatenation of Strings can result in many String objects in memory. If you must concatenate many variables and or literals, consider using an object of type StringBuilder (see the next section) instead.

StringBuilder

StringBuilder objects are like String objects, but mutable. That is, a StringBuilder object may be changed. You should use StringBuilder where a String is to be constructed in stages/concatenated using many parts.

The StringBuilder class includes many useful instance methods, but you will need to be familiar with the append and insert methods for the Java SE 8 Programmer 1 exam. Each of these methods is overloaded to enable the appending/inserting of practically anything. Consider the following:

```
String name = "David";
int age = 18;
StringBuilder builder = new StringBuilder();
builder.append("Hello, my name is ");
builder.append(name);
builder.append(". I'm ");
builder.append(age);
builder.append(" years old.");
```

The insert method requires an int offset argument in addition to the data to be inserted. Consider the following:

```
builder.insert(10, "first ");
```

This will insert the String literal "first " at offset/index 10.

The toString method of the StringBuilder class returns a String representation of the StringBuilder object. Consider the following:

```
System.out.println(builder.toString());
```

This line of code will yield the following output:

```
$> Hello, my first name is David. I'm 18 years old.
```

NOTE

The StringBuilder class is included in the java.lang package and so does not require an import statement.

java.time Classes

The java.time package was added to Java in version 8. It includes a suite of classes for representing dates and times. The classes in the java.time package supercede those in the java.util package including java.util.Date and java.util.Calendar.

All java.time class objects are constructed using static methods and are immutable (like String objects). The reason for this is beyond the scope of this course, but it does mean that any method that appears to change a java.time object does not. Instead, the method will create a new object and return a reference to it.

LocalDate

The LocalDate class is used to create objects representing a date (day, month, and year), and may be constructed as follows:

```
LocalDate today = LocalDate.now(); // yyyy-MM-dd
LocalDate magnaCarta = LocalDate.of(1215, 6, 15);
```

The LocalDate class includes many useful instance methods, some of which are listed in the table below.

method(parameters) : return	Example usage
getDayOfWeek() : DayOfWeek (enum)	DayOfWeek d = magnaCarta.getDayOfWeek(); // d = MONDAY
getDayOfMonth() : int	int i = magnaCarta.getDayOfMonth(); // i = 15
getDayOfYear() : int	int i = magnaCarta.getDayOfYear(); // i = 166
getMonth() : Month (enum)	Month m = magnaCarta.getMonth(); // m = JUNE
getYear() : int	int i = magnaCarta.getYear(); // i = 1215
isAfter(LocalDate other) : boolean	boolean b = magnaCarta.isAfter(today); // b = false
isBefore(LocalDate other) : boolean	boolean b = magnaCarta.isBefore(today); // b = true
isLeapYear() : boolean	boolean b = magnaCarta.isLeapYear(); // b = false
minusDays(long days) : LocalDate	LocalDate l = magnaCarta.minusDays(20); l = 1215/05/26
plusDays(long days) : LocalDate	LocalDate l = magnaCarta.plusDays(20); l = 1215/07/05

NOTE

In addition to minusDays and plusDays, the LocalDate class includes similarly named methods for adding and subtracting weeks, months, and years.

LocalDateTime

The LocalDateTime class is used to create objects representing a date and time (day, month, year, hour, minute, and second), and may be constructed as follows:

```
LocalDateTime thisMinute = LocalDateTime.now(); // yyyy-MM-ddTHH:mm:ss
LocalDateTime newYear2018 = LocalDateTime.of(2018, 1, 1, 0, 0, 0);
```

The LocalDateTime class includes many useful instance methods, many of which are identical to those in the LocalDate class. Where the LocalDate class includes methods for getting, adding, and subtracting days, weeks, months, and years, the LocalDateTime class includes methods for getting, adding, and subtracting seconds, minutes, hours, and days.

LocalTime

The LocalTime class is very similar to LocalDate and LocalDateTime, except that it represents a time in hours, minutes, and seconds only.

DateTimeFormatter

The default format of a LocalDate object is yyyy-MM-dd. The DateTimeFormatter class is used to format a LocalDate/LocalDateTime/LocalTime objects in any manner you choose.

NOTE

The DateTimeFormatter class is included in the java.time.format package.

A DateTimeFormatter object is typically constructed using the static ofPattern method which requires a String argument representing the desired format. Consider the following:

```
DateTimeFormatter formatter =
   DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

The DateTimeFormatter class includes many instance methods, the most commonly used, however, is the format method. The format method is passed a LocalDate/LocalDateTime/LocalTime object and returns a String representation of the date/datetime/time. Consider the following:

```
String stringDate = formatter.format(magnaCarta);
System.out.println(stringDate);
```

This line of code will yield the following output:

```
$> 15/06/1215
```

Period

The Period class is used to represent a period of time between two dates. A Period object may be obtained in one of two ways. Consider the following:

```
LocalDate iPhone1 = LocalDate.of(2007, 6, 29);
LocalDate iPhone7 = LocalDate.of(2016, 9, 16);
Period p1 = iPhone1.until(iPhone7);
Period p2 = Period.between(iPhone1, iPhone7);
```

The period objects, p1 and p2, are identical.

The Period class includes many instance and static methods, the most commonly used, however, are those that return the number of years, months, and days that constitute the given period. Consider the following:

```
int p1Years = p1.getYears();     // 9
int p1Months = p1.getMonths();   // 2
int p1Days = p1.getDays();       // 18
```

NOTE

If the first date precedes the second, one or more of the Period object's get methods will return a negative number.

ArrayList

You will recall that an array is a fixed-size collection of non-unique, indexed values of the same data type. An array is a good choice when the number of values to be stored is known in advance and will not change.

An object of type ArrayList is like an array, except that its size is not fixed. ArrayList is one of many Java collection classes and is included in the java.util package.

Declaration and initialisation

Like most Java collection classes, ArrayList is generic. This means that you must specify the type of data you intend to store when you declare and initialise an ArrayList object. Consider the following:

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

The variable names references an ArrayList object that is configured to store String references. Since Java 7, the type parameter (String in this case) may be omitted from the invocation of the ArrayList constructor. Consider the following:

```
/*
 * The type of data to be stored by the ArrayList object under
 * construction may be implied from the variable type.
 * Note that the diamond operator <> is required nonetheless.
 */
ArrayList<String> names = new ArrayList<>();
```

NOTE

The names ArrayList object may store references of type String (or any of its subclasses) only. Any attempt to add a reference of any other type will result in a compilation error.

When initialising an ArrayList object, it is good practice to specify an initial size that is an estimation of the number of objects you intend to store. This is because an ArrayList object stores its data in – you might have guessed it – an array! When the underlying array reaches its capacity, a new, larger, array is created and the existing elements are copied into it. Setting an initial size for your ArrayList object can help to minimise this costly copying of elements. Consider the following:

```
/*
 * The integer literal 100 passed to the ArrayList constructor is
 * used to set the size of the underlying array.
 */
ArrayList<String> names = new ArrayList<>(100);
```

Adding elements

Elements are added to an ArrayList object using the add method. Consider the following:

```
names.add("David");
names.add("Sarah");
names.add("Fiona");
names.add("Simon");
```

The add method is overloaded to accept a second parameter indicating the index at which the specified element is to be inserted into the list. Existing elements from the specified index forward are moved along such that element at index n becomes element at index n + 1. Consider the following:

```
names.add("Tracy", 2);
```

This resultant ArrayList object looks like this:

0. > David
1. > Sarah
2. > Tracy
3. > Fiona
4. > Simon

Removing elements

Elements are removed from an ArrayList object using the remove method. The remove method is overloaded to accept either an index, or object representing the element to be removed. Consider the following:

```
names.remove(1); // or...
names.remove("Sarah");
```

In addition to removing the specified element from the list, the remove method returns the removed object to the calling method. Consider the following:

```
String removed = names.remove(1); // removed is "Sarah"
```

Removing an element causes existing elements from the specified index forward to be moved along such that element at index n becomes element at index n – 1.

NOTE

You should avoid using indices to reference ArrayList elements given that adding and removing elements almost always changes the indices of some or all others.

Getting elements

A copy of an element in an ArrayList object may be obtained using the get method. Consider the following:

```
String thirdName = names.get(2);
```

Unlike the remove method, the get method is not overloaded to accept an object representing the element to be obtained.

Changes to an object, the reference to which was obtained from an ArrayList object using the get method, will be reflected in the list. Consider the following:

```
ArrayList<Person> people = new ArrayList<>();
people.add(new Person("Fiona", 31)); // index 0
...
Person person = people.get(0);
person.setName("Harry");
System.out.println(people.get(0).getName()); // prints Harry
```

The variable person refers to the same Person object as does element 0 of the people ArrayList object. Therefore, changes to said object using the variable person will be reflected in the ArrayList object too.

Setting elements

The object to which an ArrayList object element refers may be changed using the set method. Consider the following:

```
names.set(2, "Harry");
```

The first argument represents the index of the element to be set, and the second argument is the object to be referenced.

NOTE

The set method changes the element reference, not the object referred to.

Sizing

The number of elements stored in an ArrayList object may be obtained using the size method. Consider the following:

```
int numberOfNames = names.length;
```

NOTE

The size method returns the number of elements currently in the ArrayList object, not the length of the underlying array.

Traversing the elements

There are various ways to traverse the elements of an ArrayList object.

Consider the following traversal using a traditional for loop:

```
for(int i = 0; i < names.size(); i++) {
   System.out.println(names.get(i));
}
```

Note the use of the size method to control the number of iterations. This method of traversing the elements of an ArrayList object is appropriate when you need to know the index (i in this example) of each element.

Introduced in Java version 5, the enhanced for loop makes traversing over a list simpler than using a traditional for loop. The enhanced for loop take the following form:

```
for(<data_type> <variable_name> : <list>) {
   // code to be executed for each element in the list
}
```

The enhanced for loop is comparable to for each/for in loops in other programming languages. However, there is no each or in keyword in Java, instead the colon (:) is used. Consider the following:

```
for(String name : names) {
   System.out.println(name);
}
```

This method of traversing the elements of an ArrayList object is simpler than using a traditional for loop insofar as you don't have to use the size and get methods. Note however that you do not have access to each element's index.

Every collection class in the Java API must be iterable (it's what defines a collection). This is enforced by the implementation of the Iterable interface, and means that every collection class must provide a method that returns an Iterator object. The Iterator object may be used to traverse the elements of the collection. Consider the following:

```
Iterator<String> it = names.iterator();
While(it.hasNext()) {
   System.out.println(it.next());
}
```

The Iterator interface, like the ArrayList class, is generic and so requires that you specify the data type of the elements stored in the associated ArrayList object.

Exercises

If you don't know how to perform any of the tasks listed, or you do not get the result you expected, ask your trainer for help.

Strings

1. Create a new Java project in Eclipse named 16 Selected APIs.
2. Your trainer will provide you with the users.csv file and Strings.java source code file. The CSV file contains comma separated data as follows:

   ```
   1,Grayce,Treacher,gtreacher0@cloudflare.com,Female,195.87.120.26
   2,Arch,Skotcher,askotcher1@spotify.com,Male,192.184.231.186
   ```

   The Strings.java source code file has a main method with code to read the users.csv file line by line. Add the users.csv file to your project's root directory, and the Strings.java source file to your project's src directory.
3. Your task is to process each line and print the result to the console so as to produce the following output:

   ```
   TREACHER:cloudfare.com:195.87.120.0
   SKOTCHER:spotify.com:192.184.231.0
   ```

Case Study – Part 7

Your client at the library wants to be able to tell customers when a book, currently out on loan, is due to be returned.

1. Add an instance field to the Book class to represent the loan date.
2. Initialise the loan date to null in the constructor.
3. Add a getter method for the loan date field, but not a setter.
4. Set the loan date to the current date in the checkOut instance method only if the book is not already out on loan.
5. Add a constant to represent the loan duration in days and set it to 7.
6. Add an instance method named getDueDate to calculate and return the date the book is due to be returned. The method should return a String representation of the due date in the format – <day of week>, <month> <day of month>, e.g. Fri, Sep 1. If the loan date is null, or the book is not currently out on loan, the method should return null.
7. Save your code.
8. In the LibraryUI class, replace the TODO comment line in the checkOut instance method with the following:

   ```
   if(item instanceof Book) {
      Book book = (Book) item;
      onLoanLabel.setText(book.getDueDate());
   }
   ```

9. Replace the TODO comment line in the showBookPanel(Book book) instance method with the following:

```
onLoanLabel.setText(book.getDueDate());
```

10. Save and execute your code.
11. Add a new book and then check it out. Its due date should appear in the box labelled Available?

Case Study – Part 8

A new, larger premises has been built for the library, and your client no longer wants to limit the number of items that can be stored to an arbitrary number. Rather, she wants your library app to be able to store an unlimited number of items.

1. Change the Library class such that it has an ArrayList of Loanable items, instead of an array. This will require changes to:
   a. The instance field named items
   b. The instance field named numItems (it is no longer needed)
   c. The static field named MAX_ITEMS (it is no longer needed)
   d. The constructor
   e. The find, add, and getItems instance methods
   f. The getNumItems instance method (it is no longer needed)
2. Save your code.
3. Your changes above will break the LibraryUI class. It expects the Library object's getItems method to return an array of Loanables. Rewrite the populateItemList method as follows:

```
Loanable[] items = new Loanable[library.getItems().size()];
items = library.getItems().toArray(items);
itemList.setListData(items);
```

   The getItems instance method now returns a reference to an ArrayList object. Each ArrayList object has a toArray method which copies the list elements into the array referenced by the method parameter.

4. Save and execute your code.
5. Test your library application works as before. Its behaviour will not have changed, except inasmuch as now it can store an unlimited number of items.

CHAPTER 17

Lambda Expressions

Introduction

A lambda expression is an anonymous function, that is, a function without a name. Lambda expressions are a key component in functional programming – a programming paradigm that prioritises functions over objects.

Before Java version 8, methods could only be written to accept primitives and or object references. Consider the following:

```
public void withPrimitive(int number) { ... }
public void withObjectRef(Order order) { ... }
```

Now consider a method that will be used to filter a list. What type of thing should such a method accept as an argument? Ideally we would pass some code – an if statement to be executed for each element in the list. Previously, the only way to do this was to wrap the code in a class, and pass an object of that type. Lambda expressions enable the passing of behaviour (code segments) as method arguments.

Outer Class

Let's begin by solving the list filter problem in the traditional way. Consider the following class representing a Bank and comprising a list of Account objects.

```
public class Bank {
   // instance fields
   private ArrayList<Account> accounts;
   ...
   // instance methods
   public ArrayList<Account> filter(AccountPredicate predicate) {
      ArrayList<Account> filteredAccounts = new ArrayList<>();
      for(Account account : accounts) {
         if(predicate.test(account) == true)
            filteredAccounts.add(account);
      }
      return filteredAccounts;
   }
}
```

The filter method accepts an object reference of type AccountPredicate. AccountPredicate is an interface as follows:

```
public interface AccountPredicate {
   boolean test(Account account);
}
```

Why an interface? Because a Bank object's list of Accounts might be filtered in any number of ways. Consider the following example implementations of the AccountPredicate interface:

```
public class BigClientAccountPredicate implements AccountPredicate {
   @Override
   public boolean test(Account account) {
      return account.getBalance() >= 500_000d;
   }
}

public class InDebtAccountPredicate implements AccountPredicate {
   @Override
   public boolean test(Account account) {
      return account.getBalance() < 0;
   }
}
```


So, to filter a Bank object's list of Accounts we must first create an AccountPredicate object and then pass it to the Bank object's filter method:

```
public class BankTest {
   public static void main(String[] args) {
      Bank bank = new Bank();
      // code to populate the Bank object with accounts
      ...
      AccountPredicate predicate = new BigClientAccountPredicate();
      ArrayList<Account> bigClients = bank.filter(predicate);
      for(Account bigClient : bigClients) {
         System.out.println(bigClient.getName());
      }
   }
}
```

There is nothing wrong with this code. It is flexible insofar as a Bank object's accounts may be filtered in any way, limited only by the number of AccountPredicate implementations. But for many developers, there is too much code here to do something relatively simple. For each new way of filtering Account objects, the developer must write a new class and create a new object. This is tedious, particularly when the filter code is simple. In this example, there is only one part of one line reflecting business logic:

```
account.getBalance() >= 500_000d
```

We can go some way to reducing the amount of code needed by using anonymous inner classes.

Anonymous Inner Class

If the code that is to be used to filter the list is unlikely to be reused, then it is not necessary to create a named class. In this, and other such cases, an anonymous inner class may be used instead. An anonymous inner class is a class without a name that is implemented inline. That is, the class definition is inserted at the point at which you would typically insert an object reference. Consider the following:

```
public class BankTest {
   public static void main(String[] args) {
      Bank bank = new Bank();
      // code to populate the Bank object with accounts
      ...
      ArrayList<Account> bigClients = bank.filter(
         new AccountPredicate() {
            @Override
            public boolean test(Account account) {
               return account.getBalance() >= 500_000d;
            }
         });
      for(Account bigClient : bigClients) {
         System.out.println(bigClient.getName());
      }
   }
}
```

The code that follows new AccountPredicate() is an anonymous implementation of the AccountPredicate interface. Note that it is identical to the code inside the BigClientAccountPredicate class.

This solution is not really any better than using an outer class. There is almost as much code; it is, arguably, harder to read; and the interface implementation is not reusable. But it is a step closer to a lambda expression.

Lambda Expression

A lambda expression is an anonymous function and may be used in place of an object whose class implements a functional interface.

NOTE

A functional interface is an interface that contains only one abstract method. AccountPredicate is an example of a functional interface.

Consider the following:

```
public ArrayList<Account> filter(AccountPredicate predicate) { ... }
```

The filter method accepts an object of type AccountPredicate (as it's an interface the object must be of a type that implements AccountPredicate). As AccountPredicate is a functional interface a lambda expression may be used when invoking the filter method.

A lambda expression may take many forms, as you will see, but the long form is:

```
(<data_type> <arg_name>, ...) -> { <method body> }
```

The lambda code is the implementation of the functional interface's one abstract method. The part surrounded by round brackets is the argument list, and the part surrounded by curly braces is the method body. Note that the method's access modifier and name are omitted. Now consider the following lambda expression passed to the filter method:

```
bank.filter((Account account) -> {
   return account.getBalance() >= 500_000d;
});
```

The code passed to the filter method represents an anonymous implementation of the AccountPredicate interface's test method. (Account account) is the argument list, and the code inside the curly braces is the method body.

A number of rules apply to lambda expressions that enable the minimisation of the code:

- If the method body contains only one line of code then the curly brackets, semicolon, and return keyword may be omitted, e.g.:

  ```
  bank.filter((Account account) -> account.getBalance() >= 500_000d);
  ```

- Argument types may be implied, e.g.:

  ```
  bank.filter((account) -> account.getBalance() >= 500_000d);
  ```

- If there is only one argument then the round brackets may be omitted, e.g.:

  ```
  bank.filter(account -> account.getBalance() >= 500_000d);
  ```

The code now passed to the Bank object's filter method is stripped of almost everything bar the business logic for filtering Accounts.

NOTE

Lambda expressions may only be used when invoking a method that accepts a functional interface as an argument.

Functional Interfaces

In the previous sections we made use of a functional interface named AccountPredicate. As you might have guessed, the concept of a predicate is used in filtering generally, not just with regards to bank accounts.

There have been functional interfaces in the Java API since version 1 (though they were not called functional interfaces back then), but a whole new suite of them were introduced in Java version 8 to coincide with the introduction of lambda expressions.

The java.util.function package contains many functional interfaces, one of which is named Predicate. It contains one abstract method as follows:

```
boolean test(T t);
```

It looks very similar to the test method we included in AccountPredicate, only the data type is T, not Account. Predicate is a generic interface, meaning that the data type of the object you intend to test must be specified when you create an implementing class. Consider the following Predicate implementation:

```
public class MyPredicate implements Predicate<String> {
   public boolean test(String t) {
      // test code
   }
}
```

Of course, as Predicate is a functional interface, you may not create an implementing class, but instead use a lambda expression.

Finally, consider again the Bank class' filter method, this time refactored to accept an argument of type java.util.function.Predicate:

```
public ArrayList<Account> filter(Predicate<Account> predicate) {
   ArrayList<Account> filteredAccounts = new ArrayList<>();
   for(Account account : accounts) {
      if(predicate.test(account) == true)
         filteredAccounts.add(account);
   }
}
```

The invocation of the filter method using a lambda expression does not change:

```
bank.filter(account -> account.getBalance() >= 500_000d);
```


StayAhead Training Limited
6 Long Lane
London
EC1A 9HF

020 7600 6116

www.stayahead.com