**Section 1: Java Fundamentals**

# Subsection 1: Java’s headline features and components (5 min)

Oracle claims there are over 9 million Java developers in the world today. All of these developers made a conscious decision to learn Java, so let’s examine some of the reason why they made that decision.

**Java is easy to learn**

Java was intended to be quick and simple to learn and is therefore easier to code, compile, and debug than other programming languages such as C++. Java removed operator overloading and got shot of pointers, which are two very tricky aspects of other languages like C++.

**Java is free, mature, and stable**

There are no monetary costs involved in adopting Java. It’s free to download, world-class IDEs are available at no cost and fully featured servers are provided some of the biggest names in the industry, all at no cost.

The maturity of the Java platform and the stability that comes with age makes the choice of Java an easy one. It’s over 20 years old and still going strong. Code that was written when Java was young will still compile and execute on today’s JVM.

A large and cooperative community has grown up around Java. Developers have cooperated together to build a substantial collection of open source components that you can integrate into your application. You can even contribute to the further development of these components or open source your own for others to use.

**Java is object-oriented**

Java belongs to the class of programming languages known as object-oriented languages which models problems as objects rather than actions and acts on data rather than logic. It attempts to model real-life object making the code more natural and understandable. The developer can create modular programs and easily reusable code.

Java supports encapsulation by providing access modifiers that allow the programmer to hide variables and provide access through methods. This provides increased access control and prevents unintentional modification to variable values.

**Java is robust**

What this really means is that Java is reliable. A lot of importance is given to early checking for errors, thanks to the ability of the Java compiler to detect problems that would only show up during program execution time in other languages.

A significant feature that helps Java be robust is that it prevents memory leaks. It does this by managing its own memory and automatically collecting garbage. Other languages like C++ don’t have these features and often suffer from memory leaks.

**Java is platform-independent**

Probably the most important benefit of Java is its ability to move easily from one system to another. The Java Virtual Machine’s ability to run the same application on different systems is essential and allows environments to change. Rewriting and recompiling the code would be expensive, it’s much better to write code once and then be able to run it anywhere on any system. Java archives this through platform-independence at both the source code and byte code levels.  
  
**Java is network-centric**

Java is designed to make it easy to write networking applications that interact with resources in a distributed network. Networking is an inherent capability of the language. Working with multitier and server/client architectures is made much easier than in other languages and gives the developer a step up over the competition.

**Java has internationalization**

Java has internationalization at its centre and is the only common language to do so. Many other languages use only 8-bit characters to represent the alphabets of English and Western European languages; however, in contrast Java uses the 16-bit Unicode characters in order to represent the character sets and alphabets of the world. This feature is an integral part of the Java platform making it simple to develop applications in any language. Java is a truly world focused language.  
  
**Java is secure**

Security was baked into the platform right from the beginning and continues to be a key benefit of the platform. The language, runtime, interpreter and compiler were all developed with security in mind.

The Java platforms unique security model allows untrusted code to be downloaded over a network and to be executed in a secure environment which ensures that it cannot do any harm to the host.  
  
**Java is multithreaded**

Multithreading is the capability of an application to perform multiple jobs simultaneously. The multithreaded programming model in Java is well established when many other languages use operating system- specific features to enable multithreading.

**Java is community driven**

As already mentioned, there is a spirit of cooperation amount Java developers. Not only will you find this in the open source Java community but also in Oracle’s development of Java via the Java Community Process (JCP). The JCP is the way in which the technical specifications for the Java technology are developed. It is a transparent process that allows anyone to participate in reviewing and commenting on the Java Specification Requests (JSR), you can join as a JCP Member (https://www.jcp.org/en/participation/membership\_ind) and even participate on the Expert Groups of a JSR.

**Java is functional**

Java 8 saw one of the biggest changes in the language for nearly a decade: the integration of functional programming. It successfully blended object-oriented programming principles with functional programming constructs to produce an extremely potent language. Functional programming is particular well suited to event-driven and concurrent programming and therefore is idea for the many internet based applications that it is used to develop.

**Exercises**

Exercise 1: Which of the following are the key benefits of the Java language?

1. It is easy to learn
2. It is an immature language
3. It is robust and secure
4. It is procedural
5. It is platform-independent

Answers: 1, 3, 5

Exercise 2: Which of the following are not benefits of the Java language?

1. It is functional
2. Paid support is available
3. It is secure
4. It is community driven
5. It is object oriented

Answers: 2

# Subsection 2: The Java class structure defined (5 min)

Classes are a fundamental part of a Java program. They provide the blueprints for constructing objects.

Just like the blueprints that are used by architects to construct buildings, a Java program will use a class to construct an object.

These objects are used by the developer to model real-life things such as a bank account, a transaction, a cat, a dog, a book, list of your favourite books, and so on. In fact anything can be modelled using an object.

A Java developer will decide what objects are required by analysing the business problems that the application should solve. This means thinking about the component parts of the application. For a cash machine (ATM) application this might include a bank account, a transaction, a bank card, a pin, a keypad and a screen.

The developer would create a class that models the properties that each object has and the actions that it can perform.

For a bank account object the class would have properties such as the current balance, maximum overdraft amount and the customer’s name. It might have a bank card associated with it and the bank card would have an associated pin. Properties usually represent numerical and name properties of the object and are often objects themselves.

The actions that a bank account might perform might be to deposit an amount to the account, withdraw money from the account, and show the current balance.

We can clearly see from the discussion above that a class has two main characteristics: its properties and its actions. We call the actions that a class performs: **its methods**, and we call its properties: **its fields**.

**Classes, Fields and Methods**

**Classes**

We have already talked a little bit about what a class is; now let’s take a look at a simple example:

**public class BankAccount {}**

This class doesn’t do anything. It has no field or methods, but you can see the structure of a class definition.

**Class identifier**

A class can have any name as long as it’s unique within the package and follows the naming rules for legal Java identifiers. These rules are:

* It MUST start with a letter, currency ($,€,£ etc) or a connecting character (underscore \_ )
* It CANNOT start with a number, but can include a number
* It CANNOT include a space
* It CANNOT be a Java keyword (e,g, if, while, switch etc. See full keyword list below)
* It CANNOT be a literal (true, false, null)
* It CAN be of any length
* It CAN include a unicode characters, although note exception mentioned above
* It is case sensitive (Cat is not the same as cat)

Examples of perfectly legal, but not good, class names include:

* $greenback
* \_$
* \_\_\_\_\_\_2\_w
* a\_true\_thing
* £2000f
* \_Ö­\_
* שלום
* হ্যালো
* €\_forever

These are all perfectly legal names, and it is important to become familiar with what constitutes a valid class name; however you should also become familiar with the accepted naming convention for a good name.

The convention is that it should be singular, formatted in Pascal Case (every word starts with a capital) and should be a noun. Examples of good names are:

* BankAccount
* Window
* Item
* Driver
* Event
* Time

It is only by convention that these naming rules exist, although, to be a good Java citizen you are strongly encouraged to abide by them. To really understand the coding conventions that are used in the Java language down the Java Code Conventions PDF (<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>)

**Fields**

The fields of a class are referred to with various names: variables, class members and fields. These terms are used interchangeably.

Fields have an access modifier, a type and a name that identifies it. Look at the following example:

**private float balance;**

The responsibility of fields is to store values and the work done by the action of a method may act upon the values held by those fields.

The developer will code logic in a method that performs the function of the method.

For example, a method that deposits money into the bank account will perform a simple mathematical addition to the current balance of the account, and likewise the method that performs a withdrawal will implement logic that performs a simple subtraction from the current balance.

The **balance** field stores the account balance’s current state. The methods operate on the state and mutate it (change it) by adding or subtracting an amount.

**Methods**

As mentioned before, methods form part of the Java class structure and act on the state of the class.

A simple example of a method is as follows:

**public void withdraw(float amount){**

**// implement withdraw logic**

**}**

The data passed into a method via its parameters is acted upon by the method’s logic. In this case the logic would subtract the amount from the account’s balance.

**Comments**

An important part of developing is to comment on the code you have written. Comments give the developer an opportunity to communicate with the next developer (or oneself in the future) about the code. They are removed when the Java code is compiled, so they can be placed almost anywhere in the code.

There are three kinds of comments. The single-line comment starts with two slashes and is often used for short comments that may be temporary and designed to be replaced by code, as in the example above.

// Single-line comment for short or temporary comments

Multiline comments use slashes and asterisks to build blocks that allow more extensive comments to be made.

/\*

\* Multiple line comments for more

\* complex and detailed comments

\*/

/\*

An asterisk is only required for the

start and end of the comment block

\*/

Finally, there are Javadoc comments. These comments have a special structure so that tools like Javadoc can read them and produce detail documentations for developers. Javadoc comments are added just above the class and method definition and describe the function and parameters expected.

In the following example, you can see class level and method level comments.

/\*\*

\* This class defines a Bank Account

\* @author Alex Theedom

\* @version 1.0

\*/

public class BankAccount {

/\*\*

\* Increments account balance by given amount

\* @param amount the amount deposited

\*/

public void deposit(float amount){}

}

**Packages**

Java classes are organised into packages. These are created as directories in the file system and provide a unique namespace for the organisation of class files.

Like classes, packages follow a naming convention which consists of the same naming rules as for classes, but they must be lowercase. There is an accepted naming convention for package structures and that consists of the companies reverse domain name being used to form a hierarchal directory structure.

Let’s work through an example.

If you work for a company or client that uses the domain name [www.readlearncode.com](http://www.readlearncode.com) you would start the package folder structure **com** and then create a directory called **readlearncode**. It would look like this:

**-- com**

**|-- readlearncode**

The advantage of this naming convention is that it avoids naming collision with other organisations. Each domain name uniquely identifies a company, thus it ensures that class names do not conflict with those created by other companies.

Name collisions within an organisation can be resolved by using organisational specific characteristic such as region, department, operation, or whatever convention that suits the situation.

Therefore, an organisation whose domain name is readlearncode.com and creates software in the learning department that supports a Java course, would create a package directory structure as follows:

**-- com**

**|-- readlearncode**

**|-- learn**

**|-- java**

The developer can now create application-specific directories under the Java folder and organise classes within those directories.

A class declares its membership of a package by defining it on the first line of code with the **package** keyword. The package declaration can only ever be the first line of code (except for comments) in a class and looks like this:

**package com.readlearncode.learn.java;**

Note how the directories are separated by periods and the line terminates with a semicolon.

A class’s package also forms part of its fully qualified name. This means that a class named **Video.java** that declares membership of the **com.readlearncode.learn.java** package has the fully qualified name **com.readlearncode.learn.java.Video**.

Classes must be located in the directory that reflects the package that they declare, otherwise the code will not compile.

**-- com**

**|-- readlearncode**

**|-- learn**

**|-- java**

**|-- Video.java**

Classes that are part of the Java language itself are located in packages beginning with **java** or **javax**.

**Keywords**

Already we have met the **package**, **class**, **private** and **public** keywords; here is a list of all the keywords in the Java language.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Modifiers** | **Types** | **Flow Control** | **Not In Use** | **Classes** | **Other** |
| abstract | char | for | const | class | super |
| private | double | throws | goto | interface | synchronized |
| strictfp | int | else |  | extends | void |
| protected | boolean | if |  | implements | new |
| transient | long | break |  | import | this |
| public | byte | switch |  | package | enum |
| final | short | try |  |  | assert |
| native | float | continue |  |  |  |
| volatile |  | return |  |  |  |
| static |  | case |  |  |  |
| default |  | finally |  |  |  |
|  |  | catch |  |  |  |
|  |  | do |  |  |  |
|  |  | throw |  |  |  |
|  |  | while |  |  |  |
|  |  | instanceof |  |  |  |

You are not permitted to use any of these words as identifiers, which mean they cannot be used to name classes, packages, variables, and methods. They are reserved for exclusive use by the Java language.

You cannot have a method named **package**, but you can have a method named **copyPackage**. Keywords can form part of identifier names, they just cannot stand alone.

**Conclusion**

I have covered quite a lot of theory already, so I think that it’s time to get our hands dirty and create a class.

**Exercises**

Exercise 1: Which of the following are Java keywords?

1. switch
2. goto
3. native
4. true
5. null
6. default

Answer: 1, 2, 3, 6

Exercise 2: Select the invalid identifiers

1. \_$count
2. 2020vision
3. ^default
4. হ্যালো
5. true
6. “count balance”

Answer: 2, 3, 5, 6

Exercise 3: Select the correctly formatted comments

1. // This is not a comment
2. /\*\* Not another comment /
3. /\* No comment
4. ///\*\* More comments \*/
5. <!—Some comments -->
6. REM a remark

Answer: 1, 2, 4

Exercise 4: Which of the following are valid class name?

1. rinse-o-matic
2. \_Forever\_Young
3. 200Things
4. DefaultClass
5. $2000\_£4000
6. Large-Window

Answer: 2, 4, 5

Exercise 5: Select only the valid method signatures

1. public static void main(int count, String… args)
2. void nothing()
3. Public String printName(String name)
4. default void buffer()
5. private int true()
6. static Count count()

Answer: 1, 2, 4, 6

# Subsection 3: Program your first Java application (10 min)

Every Java application must have a class that contains a **main** method. You can think of it like an entry point to your application that attaches your application to the JVM process and keeps the application alive until it has finished executing.

A main method has a special signature that identifies it as a main method.

**public static void main(String… args){}**

The main method must be **public**, so that it can be accessed by the JVM from outside the package structure, **static** so that it can be executed by using the class name as a reference rather than an object instance (more on this concept later), it must be called **main** and must declare an array of arguments like so **main(String… args)**. Any deviation from this format will create a method that is not a main method, the JVM will not call it and your application will not launch.

Normally, an application will have only one main method which is located in a public class as follows:

public class Application {

public static void main(String… args){}

}

Now we have the bare bones of a Java application, it can be compiled and executed.

So let’s learn how to compile and execute a Java program.

**Compile Java Code**

You should ensure that you have downloaded and installed the latest version of Java SE 8 to your local machine. You can check this by opening up a command line window and executing the following command:

> java -version

You should see an output similar to this one:

java version "1.8.0\_91"

Java(TM) SE Runtime Environment (build 1.8.0\_91-b15)

Java HotSpot(TM) 64-Bit Server VM (build 25.91-b15, mixed mode)

If you don’t, you must visit the Oracle download page www.oracle.com/technetwork/java/javase/downloads and download the JDK for your operating system. Once it is downloaded, run the installation and accept all defaults. Reopen the command line prompt and type the version command again to verify that installation was successful.

Now that you have the JDK successfully installed, let’s start programming.

We are going to create a simple application in the video “Program your first Java application” that prints **Hello** to the console.

**Bytecode**

When a Java file is compiled it becomes bytecode. Bytecode is an intermediate step between the text file that contains your code and the native binary that runs on the computer’s operating system.

Every operating system is different and requires code to be compiled in the specific way that the operating system understands. You can imagine that given the multitude of operating systems out there, it would not be practical (or profitable) to compile your application for every possible operating system. Instead, the Java compiler generates what is called bytecode, this is understood by the JVM and the JVM understands how to execute the bytecode for the operating system it is running on.

So a developer only has to write and compile code once and, as long as there is a JVM running on the target machine, the application will launch and behave the same regardless of the underlying operating system and the hardware on which it’s hosted. This is where the slogan “Write once, run anywhere” derives from. It was created by Sun Microsystems to highlight the cross-platform advantages of Java and the JVM.

**Compiler Options**

So far we have seen how to compile a class using the **javac** tool, let’s dig a little deeper and look at this utility in more detail.

If you were to type **javac -help** at the command prompt you would be presented with a long list of possible options that configure the way the Java code is compiled.

It is not necessary to know how all these options affect the compilation of the code for the exam, so we will look at just those that are the most relevant for our purposes. They are the options that configure the class path (-cp or -classpath) and the option that configures the destination directory (-d).

**The class path**

The class path specifies the location of other files that are required by your application. This might be a JAR file (which is a Java ARchive file, it’s like a zip file that contains already compiled class files), resource files (such as database connection files)and any other file or directory locations. The option is either **-cp** or **-classpath** followed by a semicolon (for Windows) or colon (for Linux/Mac) separated list of locations, normally enclosed in double quotes.

On a Windows machine it might look like this:

**javac -cp “.;c:\lib\sourcefiles;c:\lib\extra.jar;\jars\\*” AClass.java**

On a Linux and Mac OS machine the same command would look like this:

**javac -cp “.:/lib/sourcefiles:/lib/extra.jar:/jars/\*” AClass.java**

The period signifies that you want to include the current directory in the class path and the asterisk wildcard matches all the JAR files in the specified directory.

**The destination directory**

By default, the location where the compiler saves the class files is at the same location in which the compile command was executed. This may not be desirable, so you can specify a destination location. This is done with the **-d** option flag.

**javac -d bin AClass.java**

This command will compile the **AClass** Java file into a directory called **bin**. The **bin** directory must exist before you execute the compile command.

**Compiling packaged code**

As you already know, classes are usually organised into packages and are also compiled into a package structure.

If the class **BankAccount.java** is organised in the package **com.readlearncode.bank.atm** it will be physically located in the directory **com/readlearncode/bank/atm/BankAccount.java** and when compiled it should be located in the directory **com/readlearncode/bank/atm/BankAccount.class**.

How do we ensure this happens? We do this by executing the compile command from the root of the project and optionally specifying a destination folder. This scenario might look like this:

Assuming the package structure is located in a project folder called **packtpub** and we are now in that location and execute the following command:

**javac -d bin com/readlearncode/bank/atm/HelloWorld.java**

This command compiles the **HelloWorld** Java file into the **bin** directory and creates a directory structure that reflects the class’s package. Its physical location will be:

**/packpub/bin/com/readlearncode/bank/atm/HelloWorld.class**

To execute the class file, assuming it has a main method, you should navigate to the **bin** directory and execute the Java command with the fully qualified location of the class file:

**cd bin**

**java com.readlearncode.bank.atm.HelloWorld**

**Execute classpath**

When executing a Java program you might need to specify a class path to files and JARs required by the program. You use the same options flag used by the compiler command and follow the same rules.

An example of an execute command might look as follows:

**java -cp “.;c:\lib\sourcefiles;c:\lib\extra.jar;\jars\\*” AClass**

**Gotchas:**

The order in which the options are placed does not matter:

javac -d bin AClass.java -cp .

javac -classpath . AClass.java -d bin

javac -d bin -cp . AClass.java

These are all the same and valid compile commands.

**Exercises**

Exercise 1: Which of the following compile commands are wrong?

1. javac -cp “.” AClass.java
2. javac -classpath . AClass.java -d bin
3. javac com/readlearncode/bank/atm/HelloWorld.java -d bin - ClassPath bin/
4. javac -d bin -classpath com/readlearncode/bank/atm/HelloWorld.java
5. javac -d bin com/readlearncode/bank/atm/HelloWorld.java -cp bin/
6. javac -destination bin HelloWorld.java -classpath .

Answer: 3, 6

Exercise 2: Choose the correctly formatted main methods

1. public static void main(String… args)
2. public static void main()
3. public void main(String args[])
4. public static void main(String[] args)
5. public static void main(String argumets)
6. public static void main(String args[])

Answer: 1, 4, 6

# Subsection 4: Import Java packages into your code (5 min)

The Java language allows you to import code, written by other developers, in to your application and use it as part of your program. When you downloaded and installed the JDK it included hundreds of packages containing thousands of classes written by developers at Oracle that enhance the language and save you from having to write a lot of low-level functionality. These packages contain classes that manipulate numbers, store and sort arrays of data, interact with the file system, and so much more. You will have a chance to learn all about these later on in the course.

To use a class from another package you must import it. You do this by using the **import** keyword followed by the fully qualified name of the class. The **import** statement is usually specified at the top of the class.

Here is an example of the import statement that imports the UUID class:

import java.util.UUID;

The UUID class has a method that generates a random universally unique identifier.

If I want to generate such an identifier I could write the code myself or I could simply import the UUID class, call the randomUUID() method and store the UUID value it returns back.

**Static imports**

A static import is a special variation of the import statement we have seen so far. It allows the developer to import the static methods and constants of another class. We talk about what static methods are later on, but for now it’s sufficient to know that they referred to by the class name rather than the instance name.

**import static java.lang.System.\*;**

**import static java.lang.Integer.\*;**

These import statements allow the developer to use the static methods of the **System** class and the **Constants** from the **Integer** class as if they were members of the same class.

In the example below the **MAX\_VALUE** of an integer is printed to the console and using the **print** method of the static constant **out**, then the static method **currentTimeMillis**, from the **System** class, is called and its return value printed to the console and finally the static method **gc**, from the **System** class, is called.

import static java.lang.System.\*;

import static java.lang.Integer.\*;

public class StaticImportExample {

public static void main(String... args){

out.print(MAX\_VALUE);

out.print(currentTimeMillis());

gc();

}

}

**Implicit Imports**

There are three packages that are implicitly imported:

* The package **java.lang**: this package contains the most essential classes required by a Java program such as **String**, **Object**, **Exception**, **System** and **Number** type classes. We will be using all these classes during this course;
* The default package: this is a package with no name. When you create a class and do not declare that, it belongs to a package. It will be put in a package that is implicitly called the default package;
* And finally, the package of the current class is imported without making an explicit import statement.

**Wildcards**

If you require more than one class from a package, instead of importing all classes one by one, you can use a wildcard and import all classes in the package. Such an import statement would look like this:

import java.util.\*;

This statement imports all classes in the **java.util** package.

It is important to note that a wildcard only matches on class names; it’s not possible to use it to import classes in a sub-package. In this case you would have to specify the sub-package followed by an asterisk.

So in the example above the import **java.util.\*** statement would only import the classes that belong directly to the **java.util** package. It would not import any classes that belong to a sub-package such as the **java.util.logging** package. To import these classes you would have to use the wildcard on this package like this:

import java.util.logging.\*;

**Redundancies**

Recall the three packages that are implicitly imported; they don’t require the programmer to use import statements in order to bring them into scope. Having said that, it is perfectly legal to explicitly import them:

**import java.lang.\*;**

**import com.readlearncode.\*;**

Although, it is really not necessary and creates a redundancy in the class’s code.

Think carefully before deciding which packages to import. If you are using an IDE to develop code you will most likely find that one of the IDE’s features is to optimise the class’s imports by automatically removing unused and duplicate imports.. This is one of the many productivity features that are common in IDEs.

**Class naming conflicts**

The advantage of using packages is that class naming conflicts are reduced. Even so, conflicts can occur if you want to use classes of the same name from different packages.

Consider this situation. There are two **Date** classes, one in the **java.sql** package and one in the **java.util** package. How can I use both of these two **Date** classes in the same class?

It’s not possible to import them both:

**import java.sql.Date;**

**import java.util.Date;**

This code is illegal and the compilation will fail with an **“a type with the same simple name is already defined by the single-type-import of Date”** error message.

You cannot use wildcards either, doing so results in a “**reference to Date is ambiguous**” error message.

In order to resolve this conflict you should use the fully qualified name of the class when making the declaration. Like so:

**java.sql.Date myDate;**

And by being explicit when declaring the variable, you don’t need to use the import statement at all.

So a class that uses both **Date** types might look like this:

**import java.util.\*;**

**public class TwoDateExample {**

**Date date;**

**java.sql.Date sqlDate;**

**}**

Consider the situation where you want to use classes from both the **java.util** and **java.sql** packages but only the **Date** class from the **java.sql** package. An ambiguity occurs and the compiler becomes confused and doesn’t know which **Date** class to use from which package.

This situation is resolved by explicitly stating in the import statements that **Date** class required. Such a class might look like this:

**import java.util.\*;**

**import java.sql.Date;**

**public class AmbiguityExample {**

**UUID uuid;**

**Date sqlDate;**

**}**

**Exercises**

Exercise 1: Which of the following imports are redundant? Assuming that you are developing code in the com.readlearncode package:

1. java.lang.\*;
2. java.util.UUID;
3. java.util.\*;
4. java.lang.System;
5. javax.lang.\*;
6. com.readlearncode.Hello;

Answer: 1, 2, 4, 6

Exercise 2: Which set of import statements are legal?

* 1. import java.util.\*;
  2. import java.sql.Date;
  3. import java.util.Date;
  4. import java.sql.Date;
  5. import com.readlearncode.\*;
  6. import java.sql.Date;
  7. import java.\*;
  8. import java.sql.Date;
  9. import java.sql.\*;
  10. import java.sql.Date;

1. import java.sql.\*;

class Example {

Date myDate;

java.util.Date Date;

}

Answers: 1, 3, 5, 6

# Subsection 5: Understand variables scope (5 min)

We have already talked about how methods act upon variables and change their state, so let’s look a little deeper at the concept of variable scope.

Variables store values, but they don’t store those values forever, they are stored for a limited period of time determined by the variables scope. The variables scope is determined by the structure of the code in which it is declared, and in Java the boundaries of a variable’s scope is defined by the open and close curly brackets ({}).

A variable defined within those brackets lives for as long as code is executing within those brackets, from the moment it is declared to the final end bracket. This is a very simplistic rule and some refinement is needed.

**Exercises**

Exercise 1: Which of the following statements are true?

1. Local scope variables go into scope when declared
2. Static variables have global scope
3. Variable declared inside a method are accessible in another methods
4. Method parameters have local scope
5. Class level variables have global scope
6. Variables defined inside a code block are accessible outside the block but only with the same method

Answers: 1, 2, 4