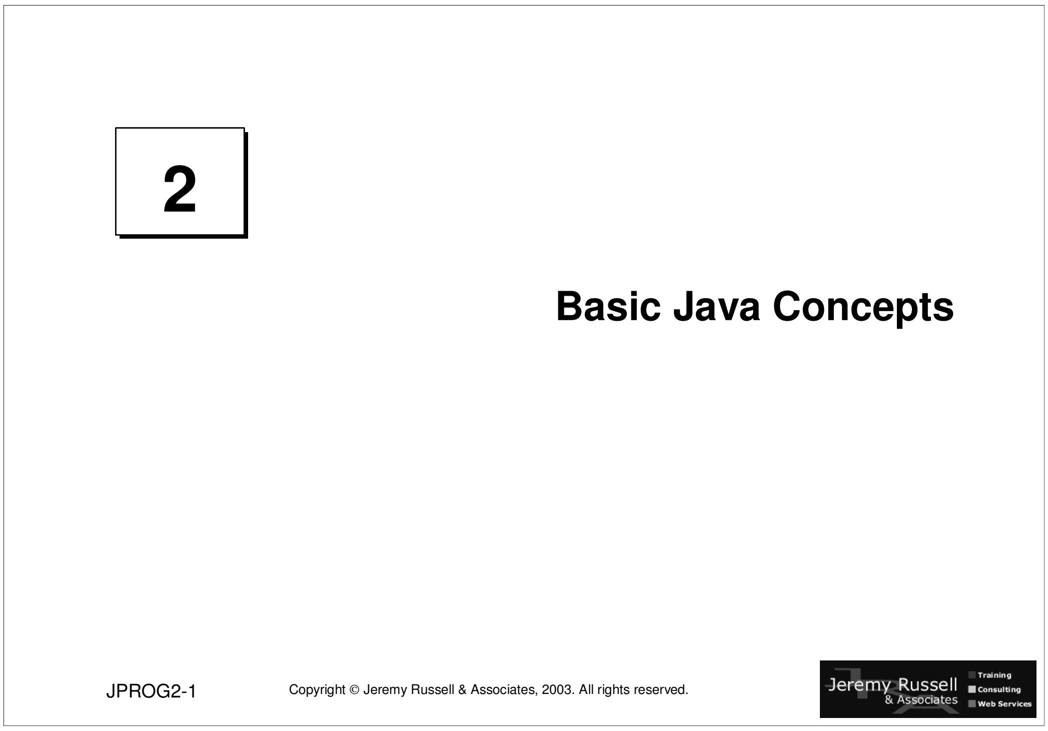


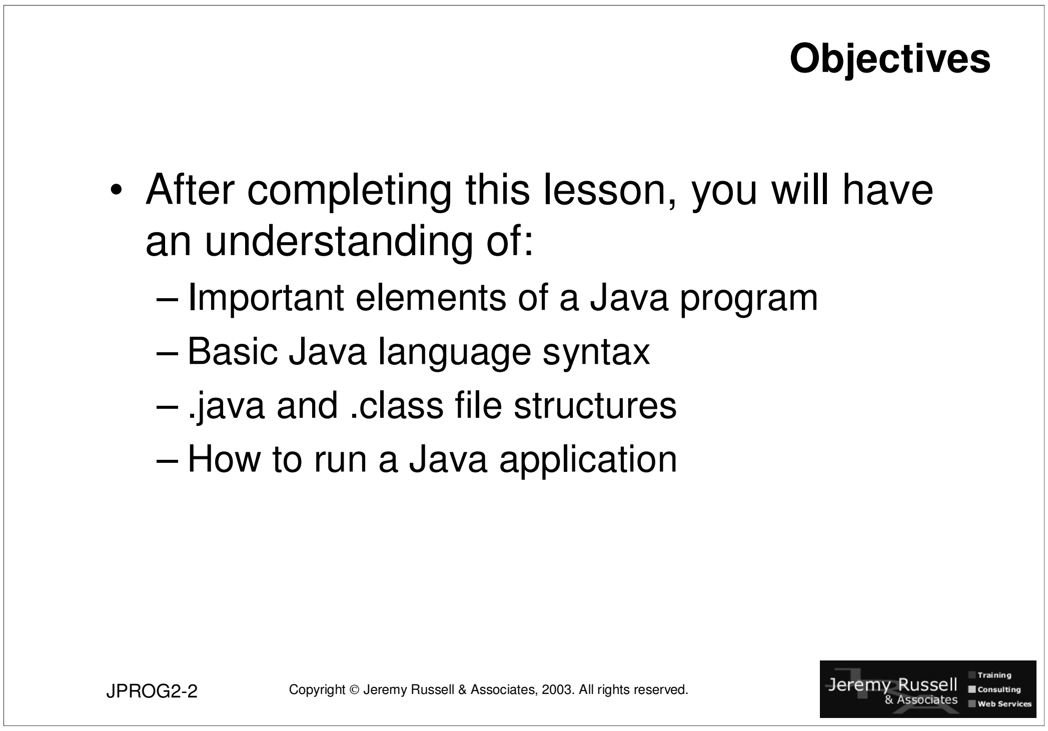
2

Basic Java Concepts

JPROG2-1

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Schedule:	Timing	Topic
	45 minutes	Lecture
	15 minutes	Lab
	60 minutes	Total



Objectives

- After completing this lesson, you will have an understanding of:
  - Important elements of a Java program
  - Basic Java language syntax
  - .java and .class file structures
  - How to run a Java application

JPROG2-2

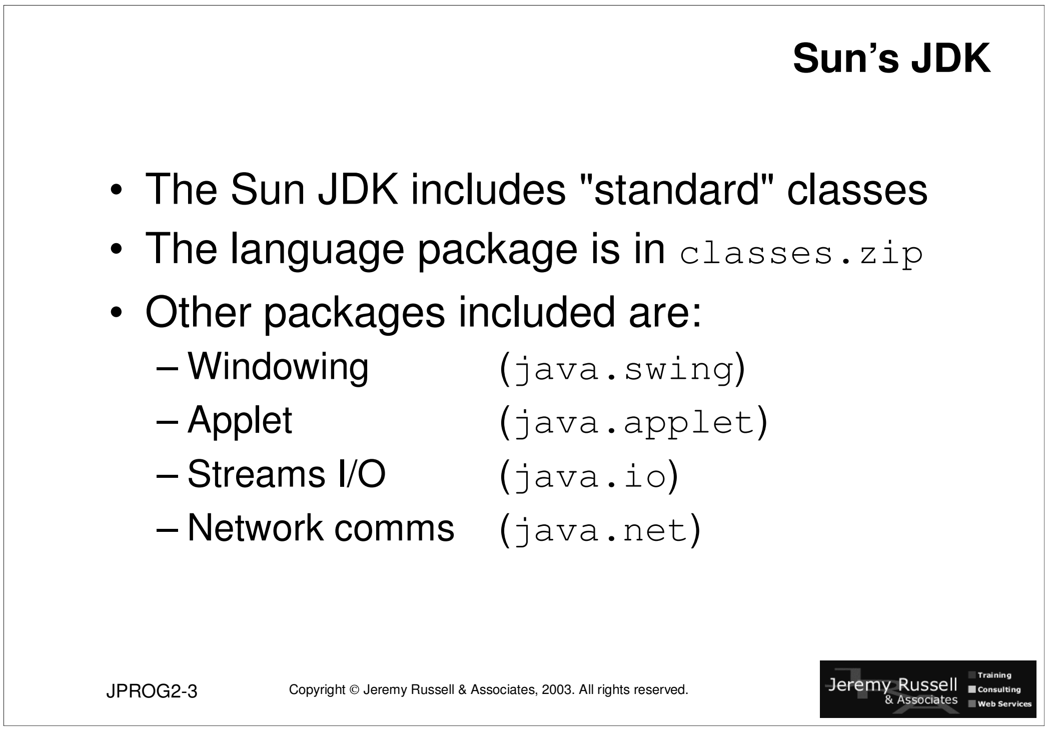
Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Objectives

Lesson 2 introduces key elements of the language for Java class creation.

Basic syntax is also demonstrated, together with coding and documentation standards.

The JDK is described in more detail to allow creation, compilation and running of simple Java programs.



Sun's JDK

- The Sun JDK includes "standard" classes
- The language package is in `classes.zip`
- Other packages included are:
  - Windowing (`java.swing`)
  - Applet (`java.applet`)
  - Streams I/O (`java.io`)
  - Network comms (`java.net`)

JPROG2-3

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Sun's JDK

The Java Developers Kit (JDK) (introduced in Lesson 1) includes a standard set of classes that provide the core functions of Java. The language package (`java.lang`) contains the lowest level of required classes. For example, `java.lang` includes the `Object` class, the foundation of all user defined (and many Java defined) classes.

The JDK includes several *packages* (groups of classes) that share common definitions. Packages also share a *name space*, the scope of internal definitions within the package. Different packages can contain objects of the same name – the fully qualified object name includes the package name for uniqueness. To simplify coding, a package can be imported into a class (discussed later), which removes the requirement to fully name packaged objects.

Included packages

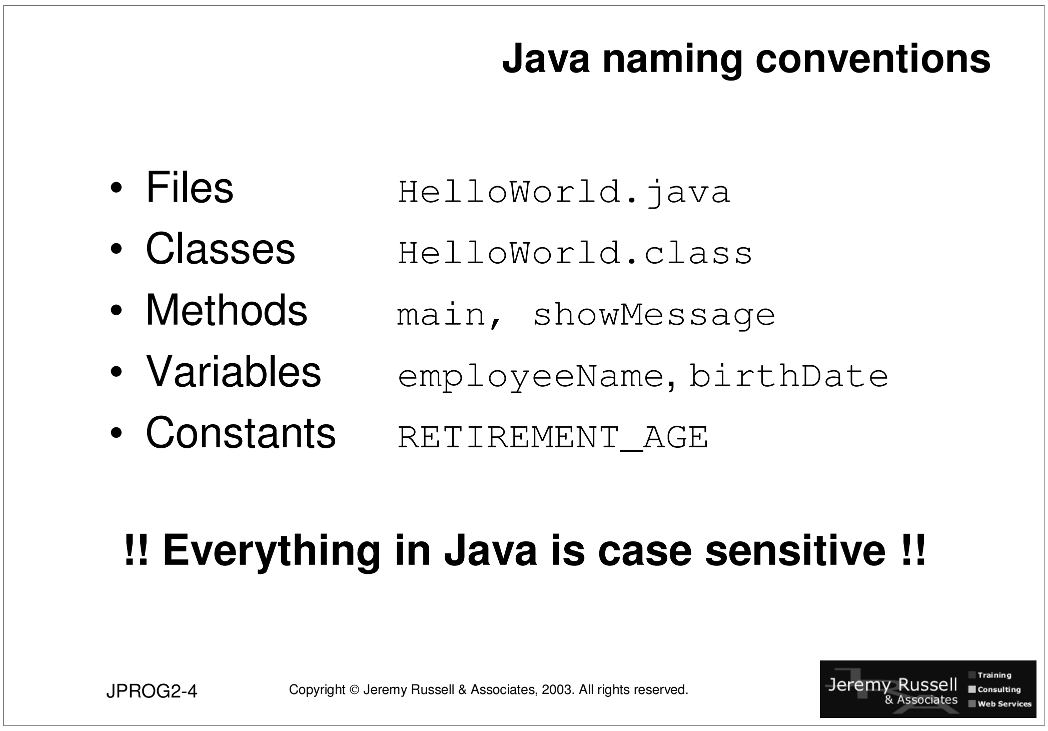
Some examples of included packages are:

▪ `java.lang`	Object, String, Thread, System, Math
▪ `javax.swing`	Window, Button, Menu, Font, Border
▪ `java.applet`	Applet, AudioClip,
▪ `java.io`	InputStream, OutputStream, File, StreamTokenizer
▪ `java.net`	Socket, URL, ContentHandler

Further information should be obtained from the JavaSoft website, from where the documentation can be browsed or downloaded. The download is an (approximately) 21.5mb ZIP file, which can then be unzipped into a set of HTML pages for use in any browser.

Alternatively, if you have access to a permanent Internet connection, the latest documentation can be browsed online.

The URL for documentation is `http://java.sun.com/j2se/1.3/docs.html.`



Java naming conventions

• Files	`HelloWorld.java`
• Classes	`HelloWorld.class`
• Methods	`main, showMessage`
• Variables	`employeeName, birthDate`
• Constants	`RETIREMENT_AGE`

!! Everything in Java is case sensitive !!

JPROG2-4

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Java naming conventions

All names and keywords in Java are case-sensitive.

Files

Source code are stored in files with the .java extension and **must** use the class name within the file as the prefix of the file name itself. The compiled classes are stored in a file with a .class suffix – the prefix must again be the same as the name of the initial class held within the file.

Classes

Class names should be nouns. The first letter of each word in the class name should be capitalised.

For example, `OrderLine`.

Methods

The name of each method is typically a verb. The first letter of the method name should be lowercase; the first letter of subsequent words should be capitalised.

For example, `getClientName()`.

Variables

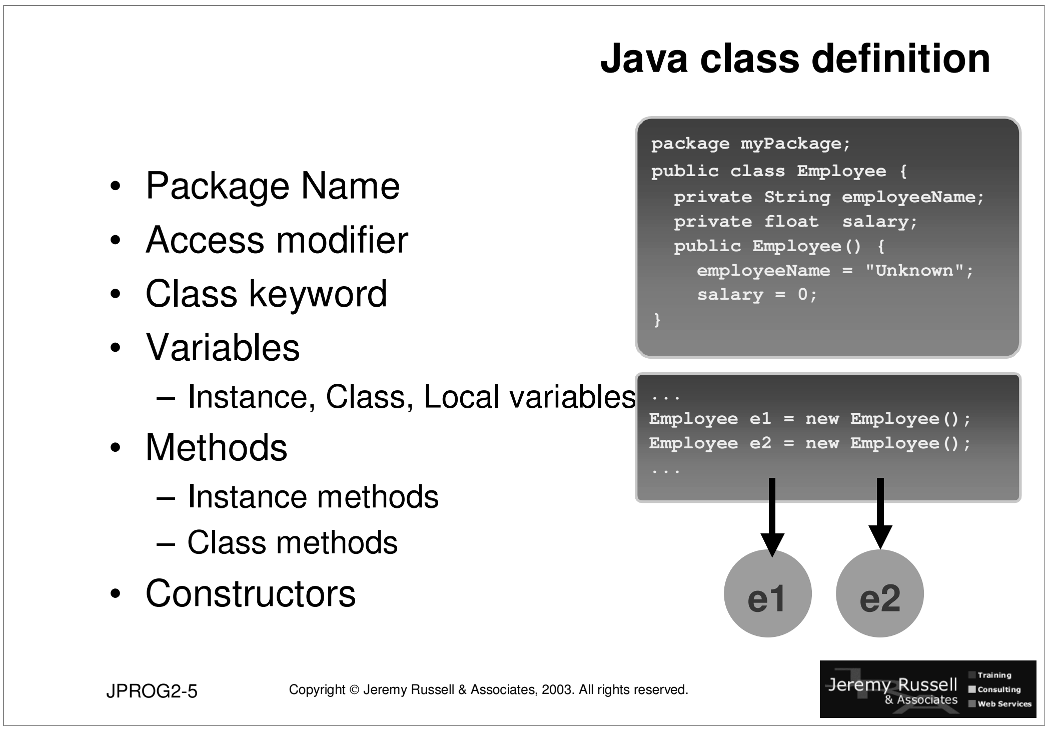
A variable name should be short but descriptive, avoiding where possible the common variable names like `i` and `j` etc.. Use the same mixed case as for method names.

For example, `employeeTaxCode`.

Constants

A constant should be declared with an all upper-case name, using underscores to separate internal words.

For example, `STANDARD_VAT_RATE`.



Java class definition

- Package Name
- Access modifier
- Class keyword
- Variables
  - Instance, Class, Local variables
- Methods
  - Instance methods
  - Class methods
- Constructors

```
package myPackage;
public class Employee {
  private String employeeName;
  private float  salary;
  public Employee() {
    employeeName = "Unknown";
    salary = 0;
}
```

```
...
Employee e1 = new Employee();
Employee e2 = new Employee();
...
```

JPROG2-5

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

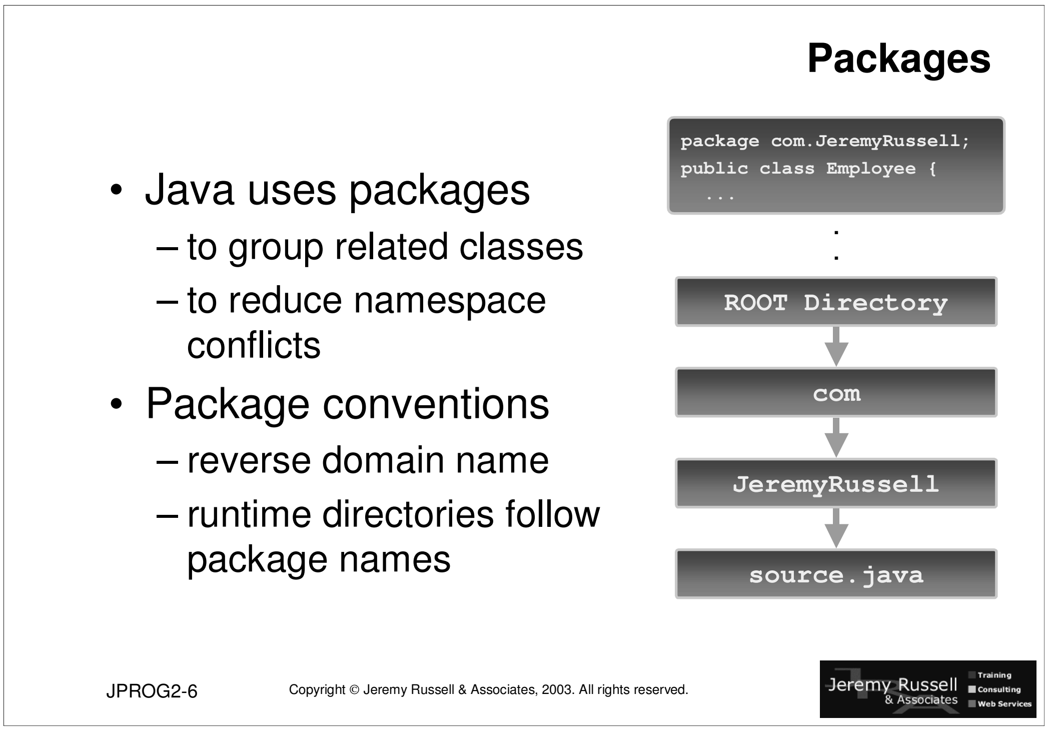
Java class definition

Classes are the *encapsulated* definition of properties (variables) and subroutines (methods) to operate on those properties. The class definition can be used as a model or blueprint for creating (*instantiating*) actual examples of the class.

The behaviour of the class is defined as *methods* that operate on the *attributes* of the class. Attribute values are stored as *variables*, either for a specific instance of the class (*instance variables*) or as variables shared by all instances of the class (*class variables*).

The class definition includes the following components:

▪ Package name	Name of the package where this class belongs – discussed later.
▪ Access modifier	Keyword to specify how external access to this class is managed. Options include `public` or `private`.
▪ Class keyword	A mandatory keyword.
▪ Instance variables	Variables (constants) defined outside of a method and available to all methods in the class.
▪ Class variables	Variables (constants) defined with the static keyword – a single instance of the variable is created which is shared by all instances of the class. Instance variables are created when the class is loaded initially and can be set and accessed even before new instances of the class are created.
▪ Local variables	Variables (constants) defined inside a method. The variable scope is inside the method where it is declared.
▪ Instance methods	Functions (subroutines) that operate on instances of the class.
▪ Class methods	Functions (subroutines) that operate on class variables of the class.
▪ Constructors	Methods that have the same name as the class and are automatically called to create new instances of the class (initialise instance variables).



Packages

- Java uses packages
  - to group related classes
  - to reduce namespace conflicts
- Package conventions
  - reverse domain name
  - runtime directories follow package names

```
package com.JeremyRussell;
public class Employee {
  ...
```

ROOT Directory
com
JeremyRussell
source.java

JPROG2-6

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Packages

Java uses "*packages*" to both group related classes and ensure that potential namespace conflicts are minimised.

Each class resides in a package – if the class does not include a package specifier, the class is a member of the default package. Each Java class is fully qualified by the fully qualified class name, which consists of the package name and the class name, concatenated with dot notation. For example, `java.lang.Object` class is the fully qualified name of the `Object` class. The `java.lang` package is included in every java class by default.

One generally accepted convention for package naming is to use the author's internet domain name as the initial components of the package name. Furthermore, the initial portion of the package name is often uppercased.

For example, packages created for use by Microsoft, with their domain name of 'microsoft.com', would typically have a package name of "COM.microsoft". If Microsoft were to create a class called "Customer" for their "licence" subsystem, the fully qualified class name would be "com.Microsoft.licence.Customer".

Package names of "java", "javax" and "sun" are reserved for use by Sun.

Class files at runtime

The root directory for the class hierarchy must also be identified in the environment variable `CLASSPATH` for both Windows and UNIX systems. This environment variable can contain a list of directories to be searched at runtime (by `java`) for the initial directory containing the class hierarchy.

At runtime, compiled Java classes must appear in a directory tree structure that corresponds to their package name. The compiler can be instructed to create the directory hierarchy by using the command line option –d, as below

```
SET CLASSPATH=C:\Course
javac –d . Source.java
```

If Source.java contains the following code:

```
package practice;
public class Question1 {
...
```

the compiled Source.class file must appear in the file `C:\Course\practice\Question1.class`.

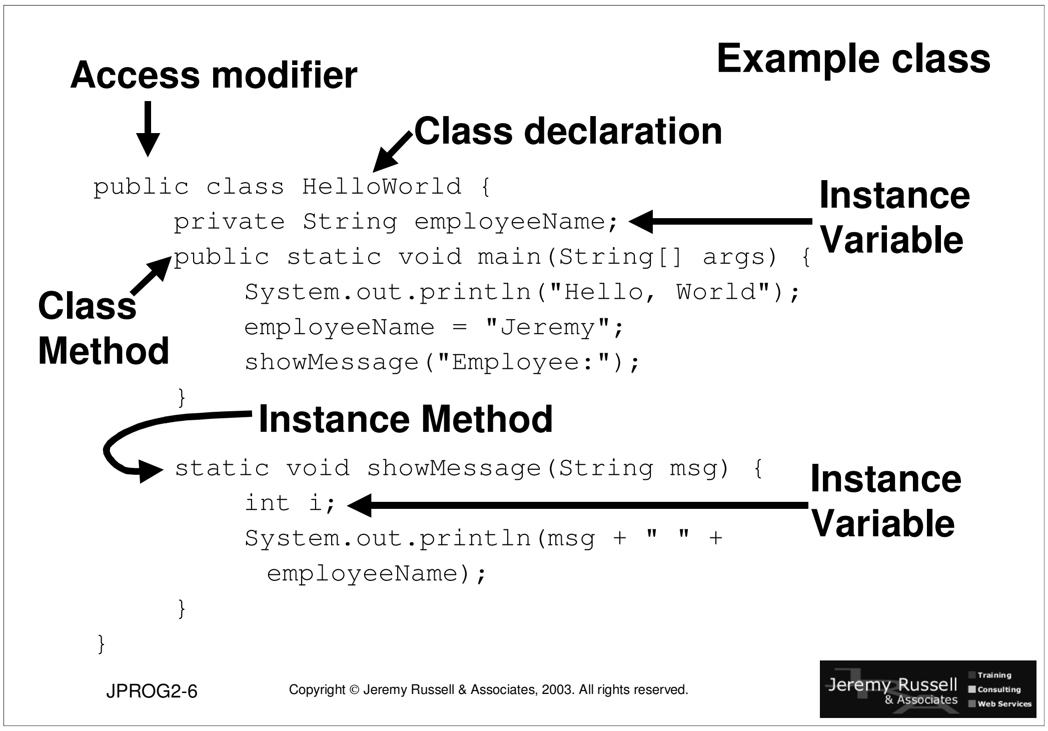
Compiled class files **must** be placed in a directory that matches their package name.

Microsoft's licence system, customer class must be stored for a Windows operating system, in "`..\com\microsoft\licence\Customer.class`" or, for a UNIX system, the directory "`../com/microsoft/licence/Customer.class`"

To execute this class, use this command:

```
java practice.Question1
```

Note that since CLASSPATH has been set already, this command can be invoked from any directory on your system.


Access modifier
Class declaration
Instance Variable
Class Method
Instance Method
Instance Variable

Example class

```
public class HelloWorld {
     private String employeeName;
     public static void main(String[] args) {
          System.out.println("Hello, World");
          employeeName = "Jeremy";
          showMessage("Employee:");
     }
     static void showMessage(String msg) {
          int i;
          System.out.println(msg + " " +
            employeeName);
     }
}
```

JPROG2-6 Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

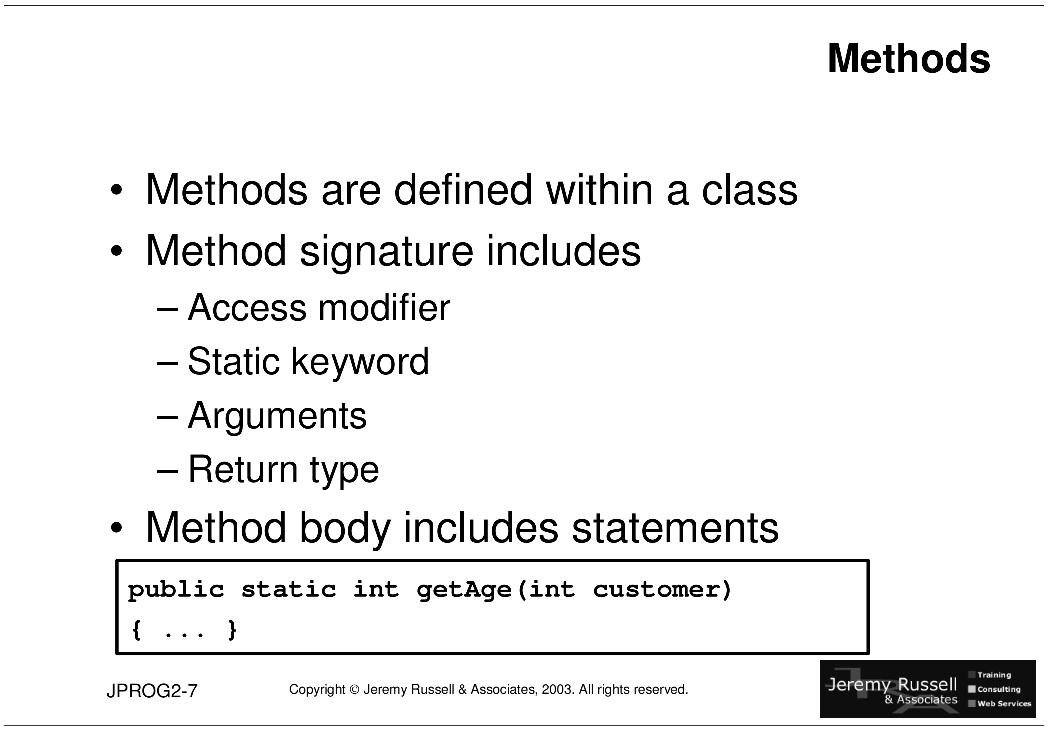
Java class definition

Classes are the *encapsulated* definition of properties (variables) and subroutines (methods) to operate on those properties. The class definition can be used as a model or blueprint for creating (*instantiating*) actual examples of the class.

The behaviour of the class is defined as *methods* that operate on the *attributes* of the class. Attribute values are stored as *variables*, either for a specific instance of the class (*instance variables*) or as variables shared by all instances of the class (*class variables*).

The class definition includes the following components:

▪ Access modifier	Keyword to specify how external access to this class is managed. Options include `public` or `private`.
▪ Class keyword	A mandatory keyword.
▪ Instance variables	Variables (constants) defined outside of a method and available to all methods in the class.
▪ Class variables	Variables (constants) defined with the static keyword – a single instance of the variable is created which is shared by all instances of the class. Instance variables are created when the class is loaded initially and can be set and accessed even before new instances of the class are created.
▪ Local variables	Variables (constants) defined inside a method. The variable scope is inside the method where it is declared.
▪ Instance methods	Functions (subroutines) that operate on instances of the class.
▪ Class methods	Functions (subroutines) that operate on class variables of the class.
▪ Constructors	Methods that have the same name as the class and are automatically called to create new instances of the class (initialise instance variables).



Methods

- Methods are defined within a class
- Method signature includes
  - – Access modifier
  - – Static keyword
  - – Arguments
  - – Return type
- Method body includes statements

```
public static int getAge(int customer)
{ ... }
```

JPROG2-7

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Methods

Each method is defined within a class, and can be equivalent to a subroutine, a procedure or a function in other programming languages.

For each method, the following forms part of the definition:

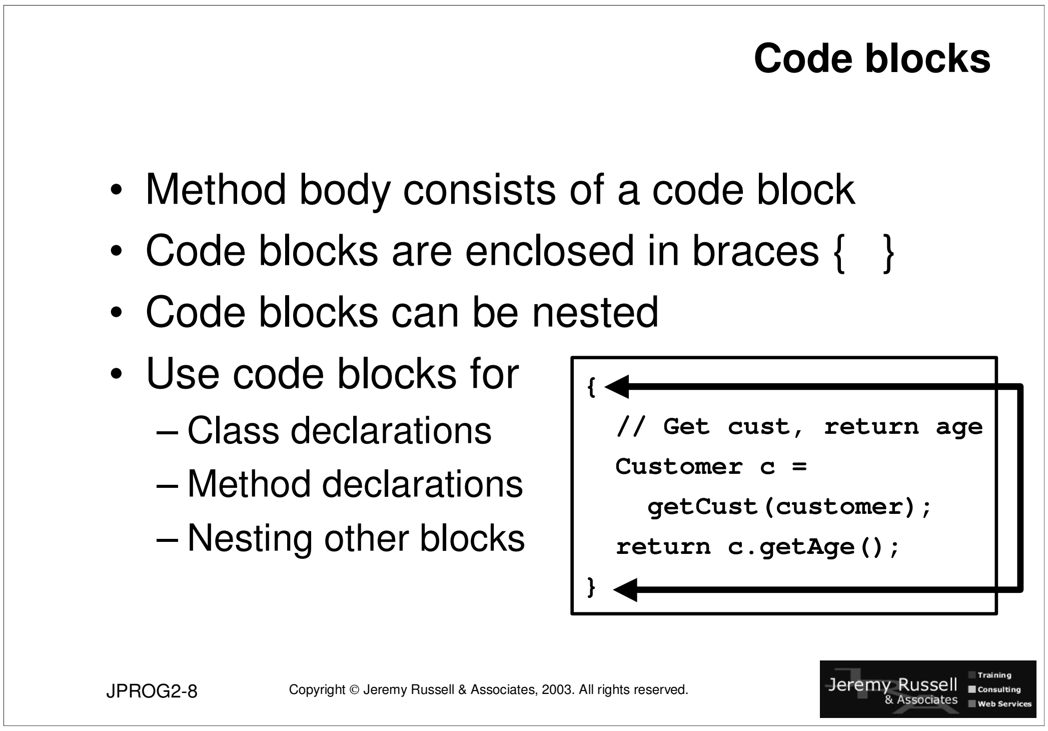
▪ Access modifier	Keyword to specify how external access to this method is managed. Options include: ▪ `public` - accessible from other classes ▪ `private` - not accessible outside this class ▪ `protected` - accessible from sub-classes of this class ▪ `default` - accessible from other classes in the same package (the default keyword does not appear but is assumed).
▪ Static keyword	A mandatory keyword for class methods.
▪ Return type	The data type returned by this method. If the method does not return a value, the data type `void` must be used.
▪ Method name	The name of the method.
▪ Arguments	A comma separated list of datatypes and names passed as parameters to this method.
▪ Method body	Java statements that provide the functionality of this method.

Unless a method does not return a value (and has a signature specifying a void return type), the method must end with a '`return`' statement to provide the value to the calling method. A method may have several exit points (`return` statements) – each statement must return the same type.

`main` method

Executable classes (*applications*) must have a "`main`" method defined. This method is the entry point to the application. Other classes do not need a `main` method but may have one defined for use as a testing entry point. The main method signature is always:

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



Code blocks

- Method body consists of a code block
- Code blocks are enclosed in braces { }
- Code blocks can be nested
- Use code blocks for
  - Class declarations
  - Method declarations
  - Nesting other blocks

```
{
  // Get cust, return age
  Customer c =
    getCust(customer);
  return c.getAge();
}
```

JPROG2-8

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

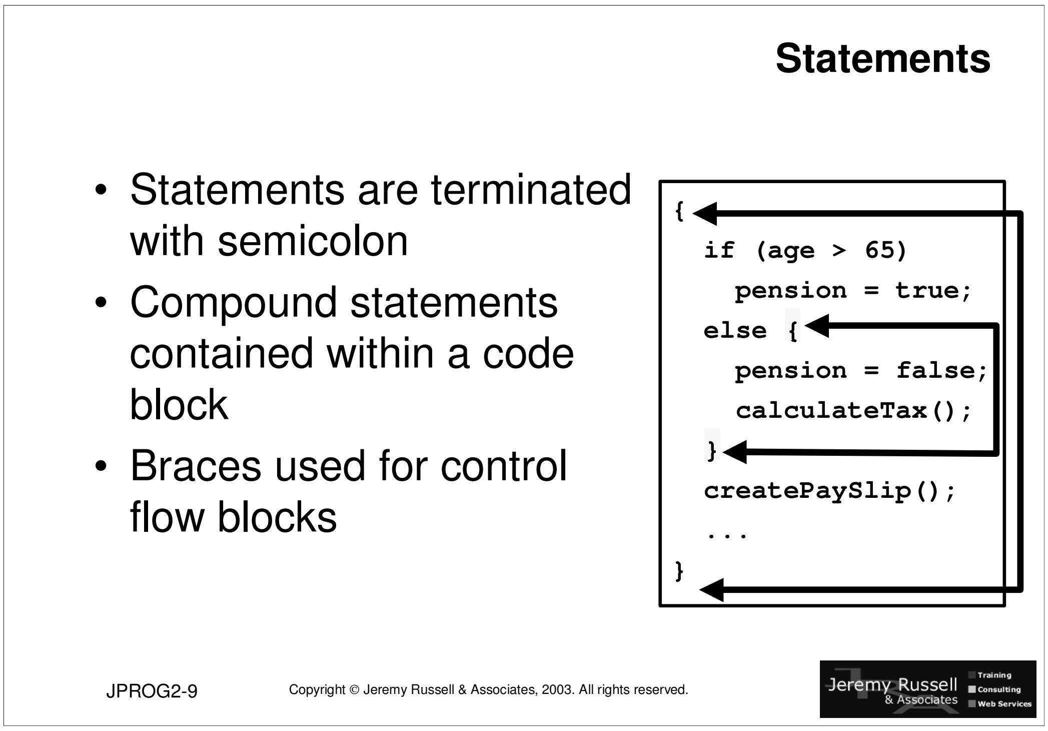
Code blocks

The body of a method appears as a code block, a set of Java statements enclosed within brace characters ({ and }).

Code blocks can be nested to form compound statements, often within conditional or loop constructs.

Variables defined within a code block have a scope of that code block only (temporary variables).

A temporary variable that is defined with the same name as a variable with a higher scope will *mask* the definition of the higher scoped variable – this is not good practice for code clarity.



Statements

- Statements are terminated with semicolon
- Compound statements contained within a code block
- Braces used for control flow blocks

```
{
  if (age > 65)
    pension = true;
  else {
    pension = false;
    calculateTax();
  }
  createPaySlip();
  ...
}
```

JPROG2-9

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

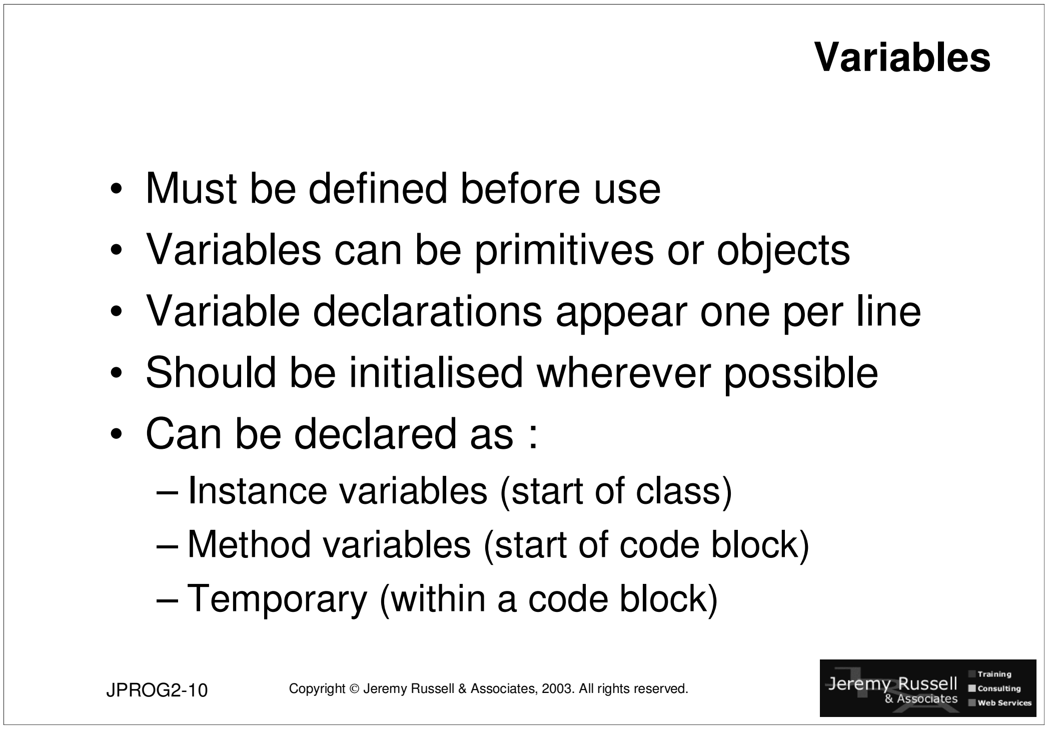
Statements

Java statements are always terminated with a semi-colon.

Statements may be Java statements (variable declaration, assignment, logic statements) or references to methods within the same or another class.

Compound statements are contained within a code block, delimited by braces.

Multiple statements can appear on a single line of source code, provided that each statement is delimited with a semi-colon. This is not good practice for coding clarity reasons and should be avoided whenever possible.



Variables

- Must be defined before use
- Variables can be primitives or objects
- Variable declarations appear one per line
- Should be initialised wherever possible
- Can be declared as :
  - Instance variables (start of class)
  - Method variables (start of code block)
  - Temporary (within a code block)

JPROG2-10

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

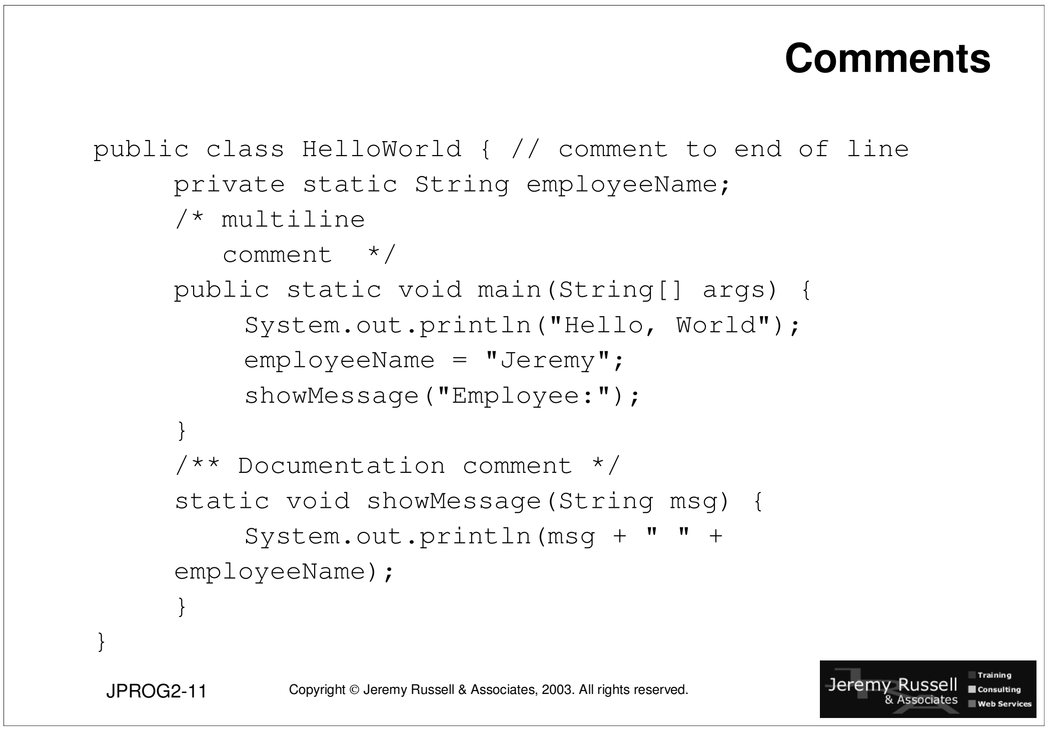
Variables

The Java language is *strongly typed*, meaning that variables must be defined **before** they are referenced.

For code clarity, a declaration should appear on a separate line in the source file. Multiple variables of the same type can be declared (and initialised) using a single statement.

Declarations can appear at the beginning of a code block or within a code block (for temporary variables).

Each variable should be initialised on declaration wherever possible. *Primitive* (built-in) variables and class variables are automatically initialised to an appropriate value (as described later). Object variables (user-defined) may be declared without initialisation values.



Comments

```
public class HelloWorld { // comment to end of line
     private static String employeeName;
     /* multiline
        comment  */
     public static void main(String[] args) {
          System.out.println("Hello, World");
          employeeName = "Jeremy";
          showMessage("Employee:");
     }
     /** Documentation comment */
     static void showMessage(String msg) {
          System.out.println(msg + " " +
     employeeName);
     }
}
```

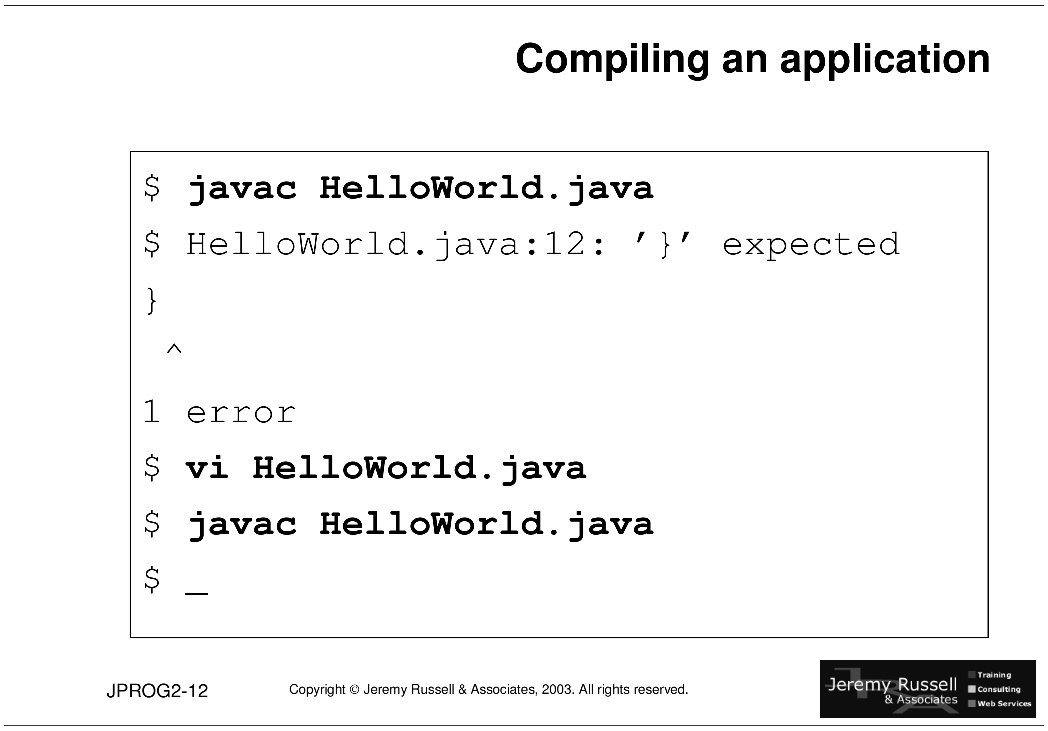
JPROG2-11 Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Comments

Code should be commented wherever possible, to ensure clarity for the original developer and for developers that may subsequently work on or use the class.

Comments can be specified in several ways:

▪ Using //	All characters after // are treated as comments by the compiler.
▪ Using /* and */	A multi-line comment is enclosed between a /* and */ pair
▪ Using /** and */	The /** prefix is used by the Javadoc utility for generating standardised documentation in HTML format (discussed later).



Compiling an application

```
$ javac HelloWorld.java
$ HelloWorld.java:12: '}' expected
}
 ^
1 error
$ vi HelloWorld.java
$ javac HelloWorld.java
$ _
```

JPROG2-12

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

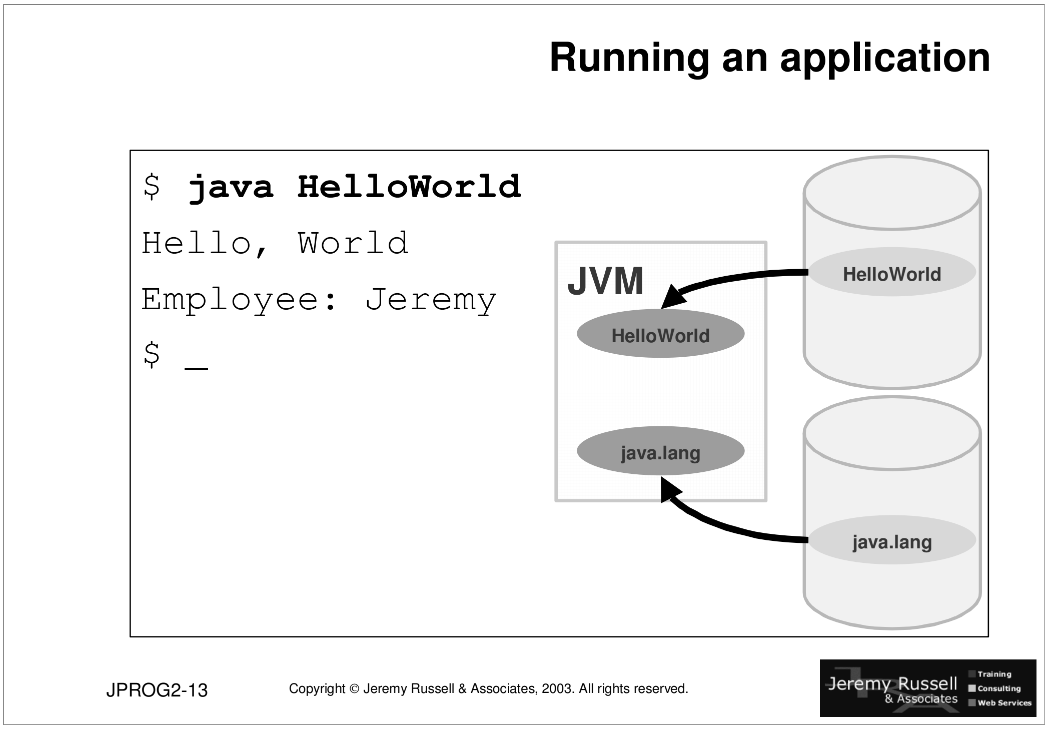
Compiling an application

To compile a Java source file, use the JDK Java compiler "`javac.exe`".

Change to the directory containing the "`.java`" file to be compiled, and run the compiler as shown above. Remember that the name of the source file is case-sensitive for the compiler, whichever operating sytem you are using for the compilation.

Errors in the code that prevent compilation will be reported by the compiler immediately.

If there are no errors, the compiler will create a class file in the same directory as the source file.



Running an application

```
$ java HelloWorld
Hello, World
Employee: Jeremy
$ _
```

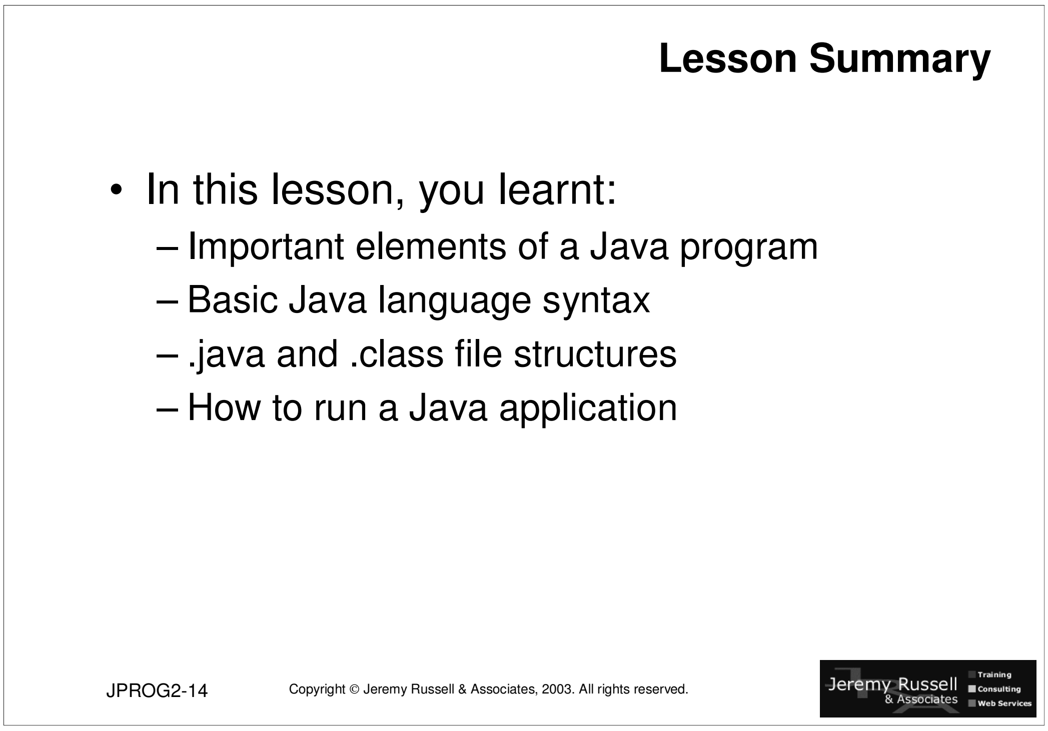
JVM
HelloWorld
java.lang
HelloWorld
java.lang
JPROG2-13
Copyright © Jeremy Russell & Associates, 2003. All rights reserved.
Jeremy Russell & Associates
Training
Consulting
Web Services

Running an application

To compile a Java source file, use the JDK Java runtime "`java.exe`".

Change to the directory containing the "`.class`" file to be execute, and invoke the Java runtime as shown above. Remember that the name of the class file is case-sensitive for the runtime, whichever operating sytem you are using for execution.

The java.exe program loads the class and verifies the integrity of the bytecodes before converting the bytecodes into machine code and executing the `main()` method. The JVM is started to manage the processing. Any other classes referenced in the application will be loaded into the JVM's memory space as required.



Lesson Summary

- In this lesson, you learnt:
  - Important elements of a Java program
  - Basic Java language syntax
  - .java and .class file structures
  - How to run a Java application

JPROG2-14

Copyright © Jeremy Russell & Associates, 2003. All rights reserved.

Practice 2

1) Using the notes in this lesson as a guide, define the following terms:
   a) Class:

   b) Instance variable:

   c) Instance method:

   d) Temporary variable:

2) Examine the following code example and answer the questions below:

```
public class HelloWorld {
   private String employeeName;
   private int age;
   private float salary;

   public static void main(String[] args) {
      System.out.println("Hello, World");
      employeeName = "Jeremy";
      showMessage("Hello, from a method");
   }

   static void showMessage(String msg) {
      int messageCode = 0;
      int messagePriority = 0;
      System.out.println(msg + " " +    employeeName);
   }
}
```

   a) What is the class name?
   b) What are the method names?
   c) What are the names of the instance variables?
   d) What are the names of the method variables?

3) List four components of the JDK.
   a)
   b)
   c)
   d)

This page intentionally left blank

Copyright © 2003 Jeremy Russell & Associates