CMT205 Object-Oriented Development with Java

Week 1

Module Information
Java Basics
Object-Oriented Concepts
Introduction to Eclipse
Simple Graphics using JavaFX

Contact Information

- Yukun Lai
  - LaiY4@cardiff.ac.uk
  - Office: S/3.06

- Matthew Morgan
  - MorganMJW@cardiff.ac.uk
  - Office: C/2.01

Module Syllabus

Week	Content
Week 1	Module Information, Java Basics, Object-Oriented Concepts, Introduction to Eclipse, JavaFX Simple Graphics
Week 2	Arithmetic, Decision, Loop Control, Keyboard Input, Command Line Input, Character
Week 3	Methods, Arrays, Exceptions
Week 4	Text and Binary Files, Packages, Inheritance
Week 5	JavaFX GUI
Week 6	JavaFX GUI
Week 7	Eclipse Features, Mathematical Methods, String Manipulation, Generics and Collections, Sorting, Class Design
Week 8	Introduction to Networking, UDP Applications, TCP Applications
Week 9	Multithreading
Week 10	Remote Method Invocation, Writing Elegant Code
Week 11	Code reuse

Module Delivery

- A very practical module
- 4 hour sessions every Thursday.
- For Weeks 1-7 and 11
  - 3 hours of lectures T/0.31 (9am-12pm)
  - 1 hour of lab C/2.08 (12-1pm)
- For Weeks 8-10
  - 2 hours of lectures T/0.31 (9-11am)
  - 2 hours of lab C/2.08 (11am-1pm)
- Material appears in either the lectures or labs is examinable

Assessment

- Coursework 30%
  - Practical Java programming based coursework with GUI and problem solving elements
  - Hand out: Week 5
  - Hand in: Week 10
  - Online submission on learning central (detail to follow)

- Exam 70%
  - 2 Hours
  - Exam Period

Reference Books

- Computing Concepts with Java Essentials, 3rd Ed., C. Horstmann, John Wiley & Sons Inc., ISBN 0-471-46900-9.
- Big Java, C. Horstmann, John Wiley & Sons Inc, ISBN 9780470553091.
- Core Java vols 1 and 2, C. Horstmann, G. Cornell, Prentice Hall, ISBN 0-13-235476-4, 0-13-235479-9.
- Java in Two Semesters. Q. Charatan, A. Kans, McGraw-Hill, ISBN 0077122674
- The Object Primer, Ambler, S., CUP, 2004
- M. Fowler, UML Distilled 3rd ed., Addison-Wesley Professional, 2004
- D Skrien, Object-Oriented Design Using Java, McGraw-Hill, 2009.
- M. Heckler, et al. JavaFX 8: Introduction by Example, 2nd Edition, 2014, ISBN: 978-1430264606

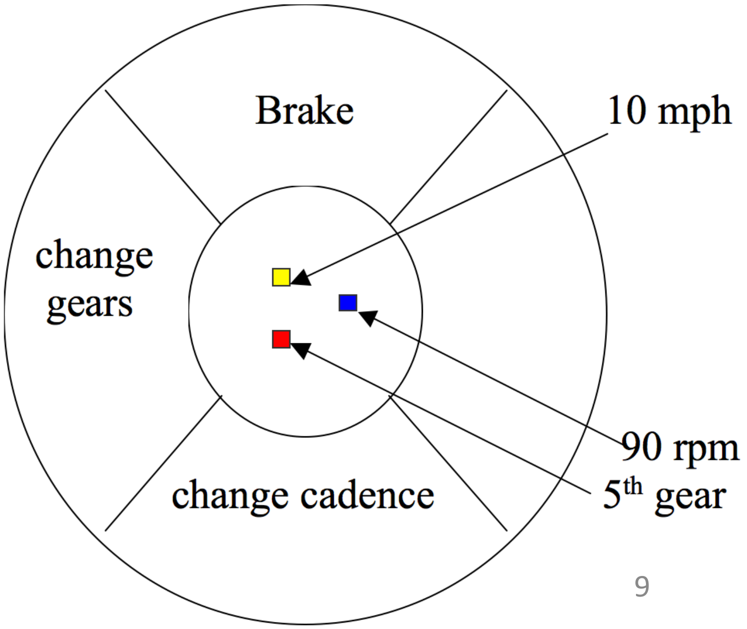
- **Java Basics, Object Oriented Concepts**
- Introduction to Eclipse
- JavaFX Simple Graphics

Introduction to Objects

- All real world **objects**, such as a **bicycle**, have a **state** and a **behaviour**.
- A **bicycle's state** includes **current gear**, **current pedal cadence (i.e. speed of rotation)** and **number of gears**.
- Its **behaviour** includes **changing cadence**, **changing gears** and **braking**.
- Software **objects** also have **state** (using **variables)** and **behaviour** (using **methods**).
- *An object is a software bundle containing variables and related methods.*

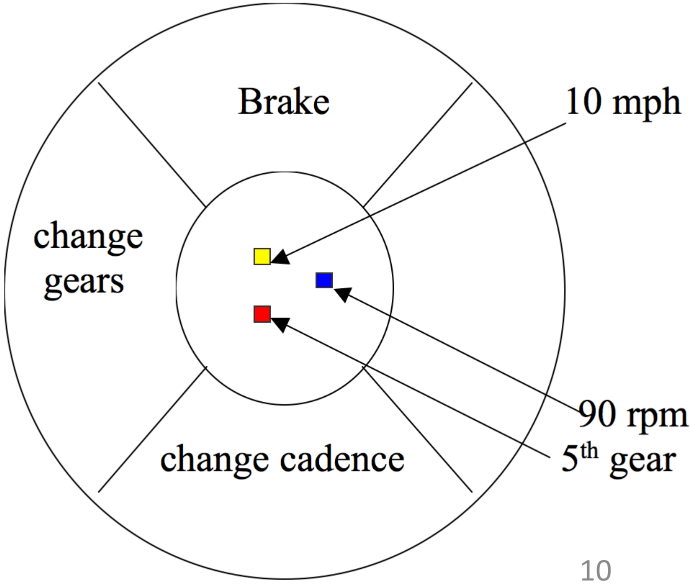
Introduction to Objects (cont.)

- Everything that the software **object** knows (**state**) and can do (**behaviour**) is expressed by the **variables** and **methods** within that **object**.
- Example: a software **object** that modelled the real world **bicycle object.**



Introduction to Objects (cont.)

- The object’s variables make up the centre, or nucleus, of the object.
- Methods surround and hide the object’s nucleus from other objects in the program.
- **Encapsulation:** packaging an object’s variables within the protective custody of its methods.



Introduction to Objects (cont.)

- Encapsulation benefits:
  - **Modularity:** the source code for an object can be written and maintained independently of the source code for other objects.
  - **Information hiding**: the object provides a **public** interface that other objects can use to communicate with it. The object can maintain **private** information and methods that can be changed at any time without affecting the other objects that depend on it.

Introduction to Classes

- You often have many **objects** of the same kind (e.g. bicycles).
- Using **object-oriented** terminology, your **bicycle object** is an **instance** of the **class** of objects known as **bicycles**.
- **Bicycles** have some **state** and **behaviour** in common.
- **Each bicycle's state** is independent of and can be different from that of other bicycles.

Introduction to Classes (cont.)

- Manufacturers take advantage of the fact that bicycles share characteristics by building many bicycles from the same **blueprint**.
- It would be very inefficient to produce a new **blueprint** for every individual bicycle manufactured.
- In **object-oriented** software, it is also possible to have many **objects** of the same kind that share characteristics such as **student records**.
- *A **class** is a **blueprint**, or **prototype**, that defines the **variables** and the **methods** common to all **objects** of a certain kind.*

Introduction to Classes (cont.)

- Instance Variables
  - The **class** for a **bicycle** would declare the **instance variables** necessary to contain the **current gear** and the **current cadence** for each **bicycle object**.
  - The **class** would also declare and provide implementations for the **instance methods** that allow the rider to **change gears**, **brake** and **change pedalling cadence**.
  - After you have created the **bicycle class**, you can create any number of **bicycle objects** from the **class**. When you create an **instance** of the **class**, the system allocates enough memory for the object and all its instance variables.
  - Each **instance** gets its own copy of all the **instance variables** defined in the **class**.

Introduction to Classes (cont.)

- Class Variables
  - A **class variable** contains information that is shared by **all instances** of the **class**.
  - If all **bicycles** had the **same number of gears**, it would be inefficient to define an **instance variable** to hold the **number of gears** as each **instance** would have its own copy of the variable but the value would be the same for every instance.
  - If the **number of gears** is defined as a **class variable**, all **instances** share this variable. If one object changes the variable, it changes for all other objects of that type.

Procedural and Object-Oriented Programming

- Programs consist of **modules** which are parts that can be **designed**, **coded** and **tested** separately and then assembled to form an entire program.
- In a **procedural language**, such as **C** or **Pascal**, the **modules** are procedures.
- A **procedure** is a sequence of **imperative** statements such as **assignment statements**, **loops** and **subprocedure** invocations.
- P**rocedural languages** are sometimes called **imperative languages**.

Procedural and Object-Oriented Programming (cont.)

- In a language such as **C**, all **procedures** are **functions** which map **arguments** to a **return value**. For example:

```
int find_max( int num1, int num2 )
{
   if ( num1 > num2 )
      return num1;
   else
      return num2;
}
```

Procedural and Object-Oriented Programming (cont.)

- **Procedural programming** is associated with a design technique known as **top-down** design
  - A problem is associated with a **procedure**.
  - If the problem is a **simple** problem, it can be solved by writing a single **C** procedure named **main**.
  - If more complicated, the problem is **decomposed** into **subproblems** where each **subproblem** is assigned to a **procedure**, which is a **function** that **main** invokes.
- The main drawback of **procedural programming** is **software maintenance**
  - A **change** in a procedure cascades or ripples down to its subprocedures and to their subprocedures and so on until the change impacts on much of the decomposition hierarchy.

Procedural and Object-Oriented Programming (cont.)

- **Object-oriented programming** (OOP)
  - an alternative to **procedural programming**.
  - the central **modules** are **classes** rather than **procedures.**
- **Object-oriented design**
  - the design technique associated with OOP
  - a **class** is a collection of **objects**
  - e.g. the class ***Employee*** is a collection of objects which are the employees of a company

Procedural and Object-Oriented Programming (cont.)

- In OOP, a **class** is a **data type** and **objects** are **instances** of such a **type**.
- Java declaration examples:

```
int num;
String greeting;
String greeting = "hello world!”;
```

- The programmer uses **classes** that the programming language provides in its standard libraries (e.g. the **String** class) and builds other **classes** suited to the application.
- An organisation might have a user-defined class **Employee** to represent an employee.

Variables and Methods

- **Instances** of a **class** share certain **properties** or **features**.
- At the programming level, **variables** contained or **encapsulated** in a **class** are used to represent these **properties** or **features**.
- To use an object, it needs to be first **constructed.**
- **Object-oriented languages** have special functions known as **constructors** for this purpose.

Variables and Methods (cont.)

- In **Java**,

  ```
  new Employee()
  ```

  would construct an **Employee object**.

  - The **constructor** in this case is **Employee()**
  - The **new** is an operator that allocates the storage for the appropriate **object**
  - A **class** has **operations** that are distinctive to it.
  - The **operations** associated with **classes** are represented by procedures **encapsulated** within the **class**.
  - **Methods** are the **functions** that represent the **operations** appropriate to a particular **class**.

Encapsulation

- In **procedural programming**, the central **modules** are **procedures** and **data** are manipulated by **procedures**
  - Data manipulation is by passing arguments to and returning a value from a **function**
- In **object-oriented programming**, the central **modules** are **classes**.
  - **Data** and the **procedures** to manipulate the data can be **encapsulated** or contained within a **class**
  - The **encapsulated** data and procedures are the **class's members**.

Class and Instance Members

- Class members vs. instance members
- In **Java**, a member marked as **static** is associated with the **class** itself
- A **static** member is a **class** member rather than an **instance** member
- Any member not marked as **static** is an **instance** member i.e. a member associated with a particular **object**
- A class's **constructors** are always **instance** members i.e. **nonstatic** members
- In general, classes tend to have more **instance** (**nonstatic**) members than **class** (**static**) members, but some classes, such as the **Math** class, only have **static** members

Difference between English and Java

- In ***English***, all ***sentences*** are terminated by a ***period*** (**.**) and a ***paragraph*** comprises ***several sentences***.
- In ***Java***, all ***statements*** are terminated by a ***semicolon*** (***;***) and a ***block*** comprises ***several statements***.

Variable Names

- The names of variables used in **Java** may contain letters, digits, the dollar sign (**$**) and the underscore character (**_**) but cannot start with a digit.
- You cannot use ***reserved words*** such as **double** or **return** as variable names.
- Traditional **Java** practice is to use lower case letters for variable names and upper case letters for symbolic constants.

Variable Sizes

- **Java** has fixed sizes for primitive types (unlike **C/C++**)

Variable type name	**Variable type**	**Size in bytes**	**Size in bits**
byte	Integer	1	8
short	Integer	2	16
int	Integer	4	32
long	Integer	8	64
float	Real number	4	32
double	Real number	8	64

Compilation and Execution

- Unlike Python, but like C/C++, Java programs need to be **compiled** before they can be run.
- The compiler translates the **Java *source code*** into ***bytecode***, consisting of
  - virtual machine instructions
  - information on how to load the program into memory prior to execution
- The ***bytecode*** of the program **Hello.java** is stored in a file **Hello.class**.

Compilation and Execution (cont.)

- A **Java *interpreter*** loads the bytecode of your program, starts the program and loads the necessary ***library bytecode files*** as they are required.

- To ***compile*** and ***run*** the **Hello** program, you would type:

```
javac Hello.java
java Hello
```

An Example Program

Contents of Hello.java

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

An Example Program (cont.)

Compilation of Hello.java

```
javac Hello.java
```

Execution of Hello.class

```
java Hello
```

Result

```
Hello World
```

An Example Program (cont.)

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

- The class **Hello** must be written to a file **Hello.java**
- The keyword **public** denotes that the **class** is useable by the ‘public’
- Every **Java** application must have a **main** method but can contain other methods
- The parameter, **String[ ] args**, is required by the **main** method to process command line arguments
- The keyword **static** is mandatory for the **main** method and indicates that the **main** method does not change objects of the class **Hello** and Java does not need to create an instance of this class to call this method

Further Discussion of Hello.Java

- An ***object*** is an entity that you can manipulate in your program by calling ***methods***.
- **System.out** was an ***object*** and it was manipulated by calling the **println** ***method***.
- When the **println** ***method*** is called, activities occur inside the **System.out** ***object*** which cause text to appear on the terminal screen.
- An ***object*** can be considered as a '**black box**' with a public ***interface***, the ***methods*** you can call, and a hidden ***implementation***.

Using Comments

- Comments are ignored by the compiler but improve the readability of the program
- It is good practice to write comments for classes, methods etc. and for key blocks of code
- Java comments
  - // text: the compiler ignores everything from // to the end of the line
  - /* text */: the compiler ignores everything from /* to */
  - A special case: /** text */, just like /* text */ for the compiler but `javadoc` use this to generate documentation (more detail later)

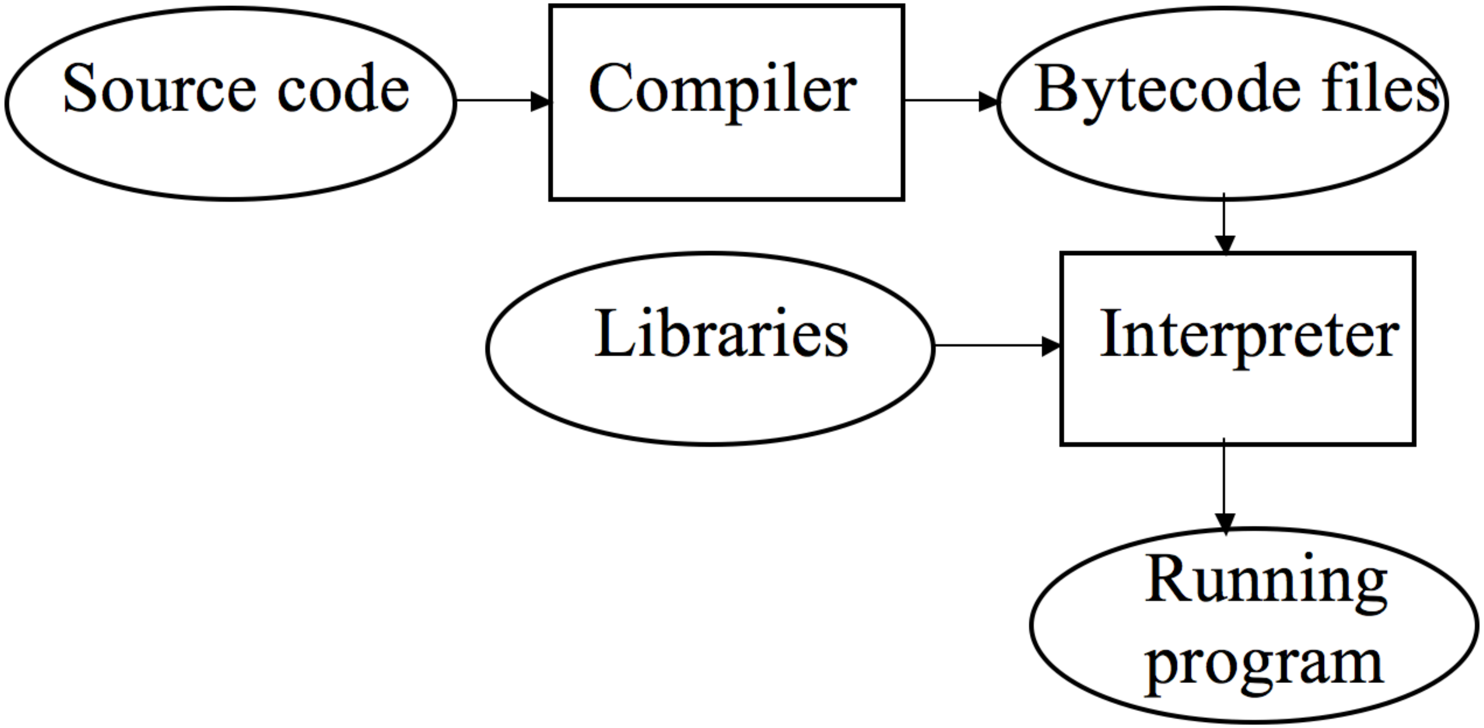
A better HelloWorld.java

```
/**
 * The HelloWorld class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

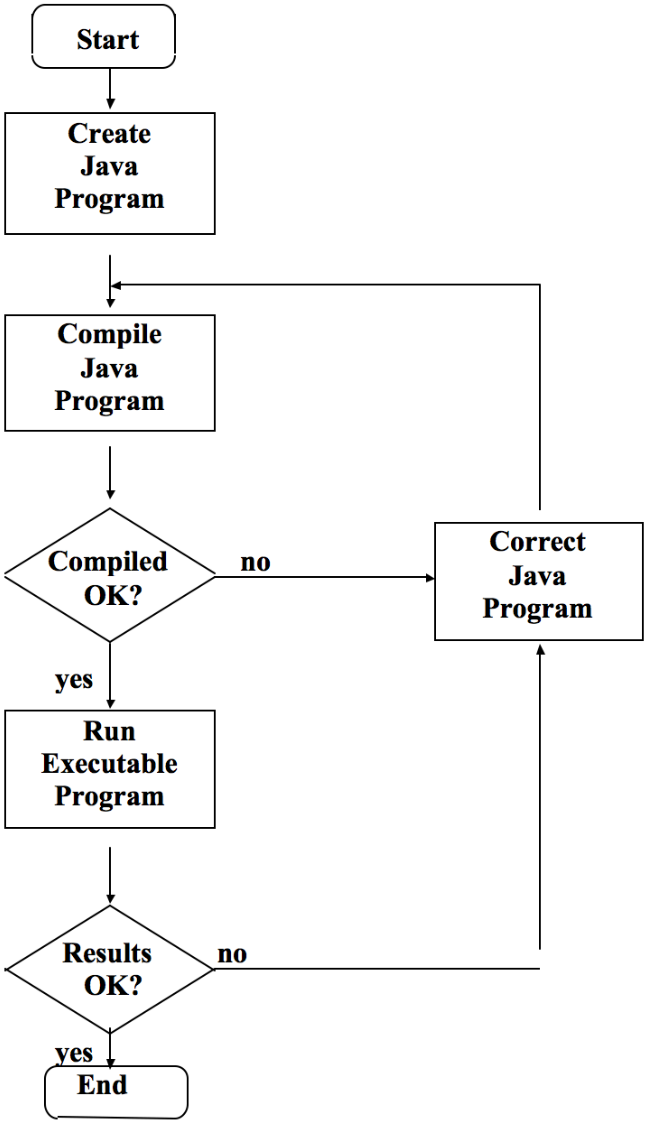
Result:

```
Hello World!
```

Running a Java Program



Program Develop Cycle



Class Implementation

- The following steps are used to implement a class
  - Discover the behaviour
  - Define the methods
  - Determine the instance variables for representing the state of an object of the class
  - Implement the methods

Class Definition

- A ***class*** contains
  - one or more ***methods***
  - zero or more ***constructors***
  - zero or more ***instance*** variables
- The ***class*** of an application ***must*** contain the ***main*** method, which is a ***special*** method.
- The ***instance*** variables of a ***class*** reflect the ***state*** of an ***object*** of that ***class***. The ***methods*** of a ***class*** are used to change the ***state*** of an ***object*** of that ***class***.

Class Definition (cont.)

```
public class ClassName
{
        // constructors
        accessSpecifier  ClassName ( parameterType parameterName, . . . )
        {
          constructor implementation
        }
        . . .
        // methods
        accessSpecifier returnType methodName ( parameterType parameterName, . . . )
        {
          method implementation
        }
        . . .
        // instance variables
        accessSpecifier instanceVarType instanceVarName;
        . . .
}
```

Constructing an Object

- The **behaviour** of a **class** is the **complete list of methods** that you can apply to **objects** of the given **class**.
- To **construct** an **object** of the given **class** you first have to declare an **object variable**.
- An **object variable**, ***variableName***, may be created for **class**, ***ClassName*** as:

```
ClassName variableName;
```

- The **object variable** ***variableName*** above does not refer to an **object** (a **null** reference) yet.

Constructing an Object (cont.)

- The object variable is initialised (assuming there are no **construction parameters**):

```
variableName = new ClassName( );
```

- To declare and ***initialise*** the ***object variable*** in a single statement:

```
ClassName objectName = new ClassName( );
```

Methods

- A ***method*** header consists of the following parts:
  - An ***access specifier*** (such as **public**)
  - The ***return type*** of the method (such as **double**)
  - The ***name*** of the method
  - A list of the ***parameters*** of the method

```
public class ClassName
{
        . . .
        accessSpecifier returnType methodName
               ( parameterType parameterName, . . . )
        {
                method implementation
        }
        . . .
}
```

Methods (cont.)

- When a ***method*** is declared as **public**, it can be accessed by the ***methods*** of all classes.
- If a ***method*** does not return a value, a ***return type*** of **void** is used.
- The ***parameters*** are the inputs to the ***method***.

- We discuss briefly here, more details of methods come later.

Instance Variables

- An ***instance variable*** is a variable that is present in every ***object*** of a ***class***.
- The current ***state*** of an ***object*** is stored in one or more ***instance variables***.
- An ***instance variable*** declaration consists of the following parts:
  - An ***access specifier*** (such as **private**)
  - The ***type*** of the variable (such as **double**)
  - The ***name*** of the variable

```
public class ClassName
{
         . . .
         accessSpecifier  type  variableName;
         . . .
}
```

Instance Variables (cont.)

```
public class ClassName
{
         . . .
         accessSpecifier  type  variableName;
         . . .
}
```

- ***Instance variables*** are declared with the ***access specifier*** **private** so that they can only be accessed by the ***methods*** of the ***same class***
- If the ***instance variables*** are declared **private**, then all data access must occur through the ***public methods***.
- The ***instance variables*** of an object are effectively hidden from the programmer who uses the ***class***.
- The process of hiding data is called ***encapsulation***.

Constructors

- The ***constructor*** initializes the ***instance variables*** of an ***object***. The format of a ***constructor*** is:

```
public class ClassName
{
      . . .
      accessSpecifier  ClassName
              ( parameterType parameterName, . . . )
      {
            constructor implementation
      }
      . . .
}
```

Constructors (cont.)

- ***Constructors*** always have the same name as their ***class***.
- ***Constructors*** are generally declared as ***public*** to enable any code in a program to ***construct*** new ***objects*** of the ***class***.
- ***Constructors*** do not have ***return types*** and are always invoked using the ***new*** operator.
- The ***new*** operator allocates memory for the ***object***, and the ***constructor*** initialises it.
- The value returned by the ***new*** operator is a ***reference*** to the newly allocated and constructed ***object***.

Constructors (cont.)

- Normally, the ***object*** reference is stored in an ***object*** variable.
- A ***default constructor*** is a ***constructor*** which takes ***no parameters***.
- A ***class*** may have more than one ***constructor***.
- These ***constructors*** will have the same ***name*** as that of the ***class*** but ***different parameters***.
- Multiple ***methods*** (or ***constructors***) with the same ***name*** are said to be ***overloaded***.
- When and only when no user-defined ***constructors*** are given, a default constructor is automatically provided which initialises the member variables to standard default values.

A Simple BankAccount Example

- Contents of **BankAccount.java**

```
public class BankAccount
{
   // default constructor
   public BankAccount()
   {
      balance = 0;
   }
   // second constructor
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
   // method for depositing money
   public void deposit(double amount)
   {
      balance = balance + amount;
   }
```

A Simple BankAccount Example (cont.)

```
   // method for withdrawing money
   public void withdraw(double amount)
   {
      balance = balance - amount;
   }
   // method for getting a balance
   public double getBalance()
   {
      return balance;
   }
   // instance variable
   private double balance;
}
```

A Simple BankAccount Example (cont.)

- Contents of **BankAccountTest.java**

```
public class BankAccountTest
{
   public static void main( String[] args )
   {
      // Open a new bank account with 10000 pounds
      BankAccount account = new BankAccount( 10000 );
      // Display balance of account
      System.out.println( "Balance of account is "
              + (int) account.getBalance() );
      // Withdraw 3500 pounds
      account.withdraw( 3500 );
      // Display balance of account
      System.out.println( "Balance of account is "
              + (int) account.getBalance() );
      // Deposit 1500 pounds
      account.deposit( 1500 );
      // Display balance of account
      System.out.println( "Balance of account is "
               + (int) account.getBalance() );
   }
}
```

A Simple BankAccount Example (cont.)

- Result
- `java BankAccountTest`

```
Balance of account is 10000
Balance of account is 6500
Balance of account is 8000
```

Method Parameters

- Explicit parameters vs. the implicit parameter
  - a ***parameter*** in the ***parameter list*** of a ***method*** is known as an ***explicit parameter***
  - the reference to an ***object*** is not ***explicit*** in the ***method*** definition, and is called the ***implicit parameter*** of the ***method***
  - the **implicit** parameter has the name **this** when referred in the method implementation
- The statement to call a method
  - *implicitParameterValue.methodName(explicitParameterValues)*
- When you refer to an ***instance variable*** in a ***method***, you automatically refer to the ***instance variable*** of the ***object*** for which the ***method*** was called

Java Language Coding Guidelines

- Variable and method names are lowercase with occasional uppercase characters in the middle.
- Class names start with an uppercase letter.
- Constant names are uppercase with an occasional underscore.
- Leave a blank space after but not before each comma, semicolon and keyword.
- Leave a blank space before and after each operator.

Java Language Coding Guidelines (cont.)

- Braces must line up horizontally or vertically.
- Use a constant definition instead of embedding a numeric constant in code.
- Every method, except for the **main** method and overridden library methods, must have a comment.
- At most **30** lines of code may be used per method.
- All variables which are not **final** variables must be **private**.

Important Definitions

- **instance variable**
  - A variable that is a permanent part of a particular **object**
  - Memory space for the variable is allocated when the object is created.
- **class variable**
  - A variable associated with **all objects** of the **same class**.
  - Created when the **class** is loaded.
- **state**
  - the values stored in the **instance variables** and **class variables** of an **object** at any given time.

Important Definitions (cont.)

- **query**
  - **obtains** the **current state** of an **object**.
- **command**
  - **changes** the **state** of an **object**.
- **class**
  - a set of **objects** having the same features and properties
  - every **object** is an **instance** of some **class**, which determines the object's features.

- Java Basics, Object Oriented Concepts
- **Introduction to Eclipse**
- JavaFX Simple Graphics

Introduction to Eclipse

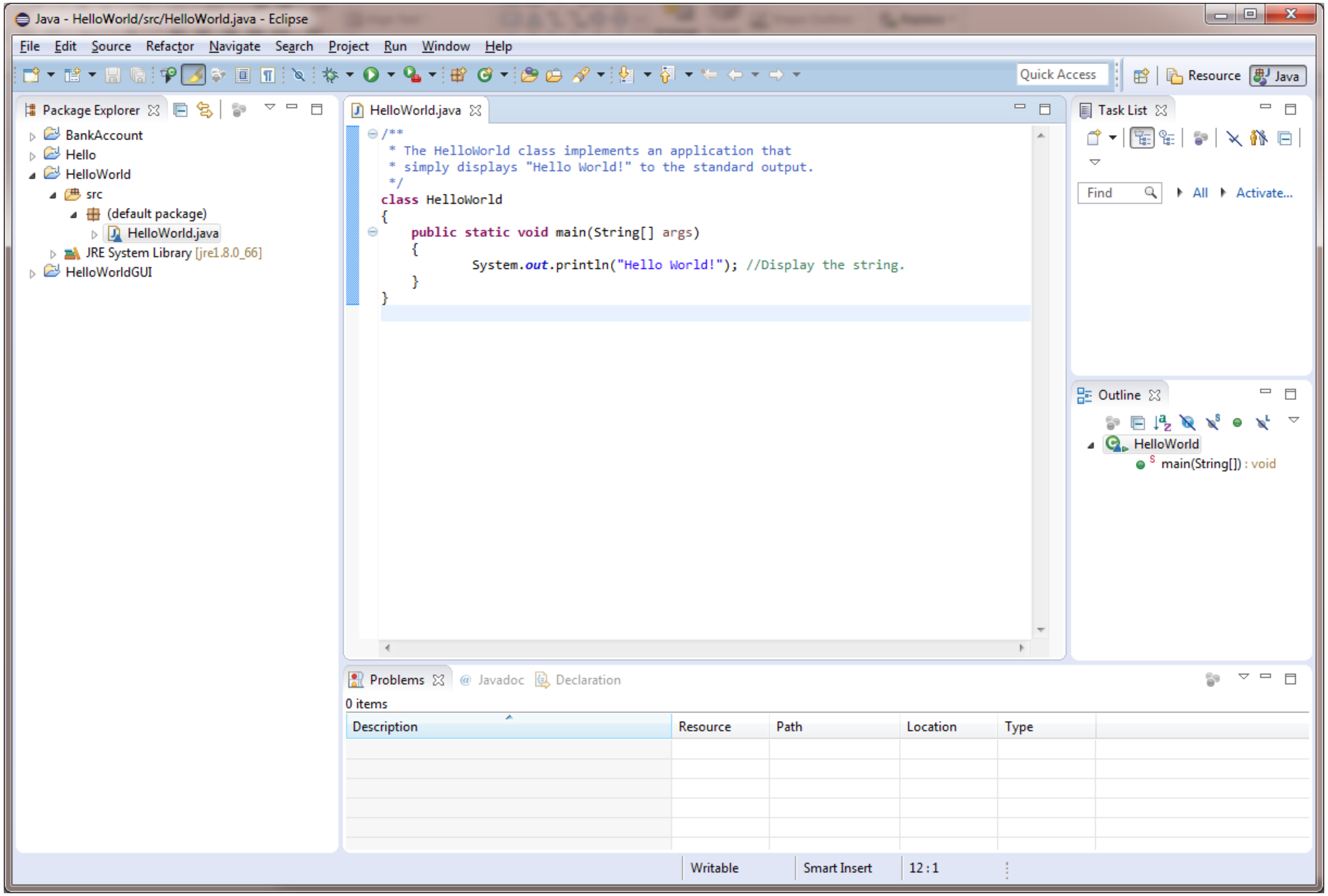
- Instead of using command line tools, Java programs can also be developed using IDE – Integrated Development Environment
- Eclipse is a free, open-source IDE for Java (and some other languages)
- It is available on Windows, Mac and Linux.
- For introductory notes, see

https://docs.cs.cf.ac.uk/notes/eclipse-sdk-for-java/

Benefits of Using IDE

- Eclipse provides a variety of features to speed up the development process and fix bugs:
  - Effectively organising Java source files in a project, compiling all the modules in the project and running and debugging the program, all in an integrated environment.
  - More powerful editing support for the language, including syntax highlight, line indentation, bracket matching, live hints and code completion.
  - Powerful built-in debugging.
- We give the basic introduction now and will cover more advanced features later.

Eclipse UI



```
/**
 * The HelloWorld class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorld
{
    public static void main(String[] args)
    {
            System.out.println("Hello World!"); //Display the string.
    }
}
```

0 items

Description	Resource	Path	Location	Type

Writable | Smart Insert | 12 : 1

Eclipse UI

- A typical Eclipse UI includes
  - **The main menu and toolbar**
  - **Editor** (for code editing)
  - **Package Explorer** (listing all the packages in the current workspace)
  - **Outline** (showing the code structure and allowing easy navigation)
  - **Problems/Javadoc/Declaration/Console** (useful information)
  - **Panels for Debugging** (including those only shown when the program is being debugged)

Workspace

- Workspace
  - A workspace is used to manage a collection of projects
  - You are asked to choose a location for the workspace when you start Eclipse for the first time
  - You can use **File → Switch Workspace** if you would like to switch to a different workspace

Perspectives and Windows

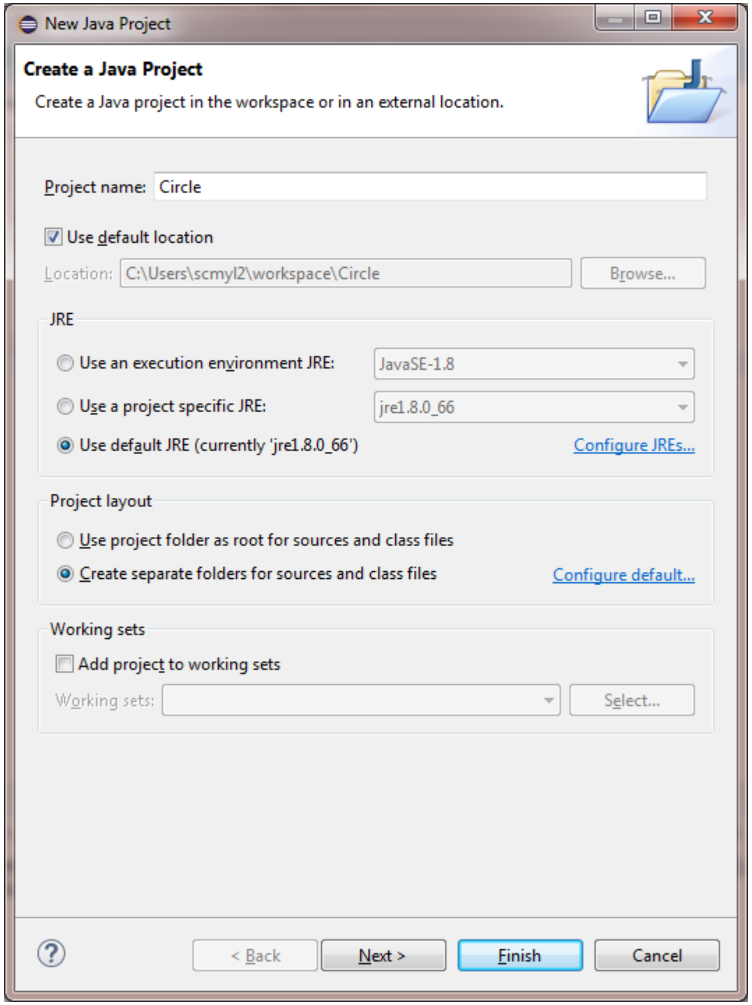
- Managing windows
  - Eclipse is an IDE for a variety of languages, Java, C++ etc.
  - You may want to show different panels when doing different tasks (e.g. typing code, debugging)
  - Perspectives: allow quick switching between different window configurations. For Java programming, choose **Window → Perspective → Open Perspective → Java** (or if **Java** is not listed, choose **Other**, then **Java**)
  - You may also use **Window** menu to open a specific window, e.g. **Window → Show View → Package Explorer** will show/switch to **Package Explorer**.

Creating, Building and Running Projects

- Eclipse uses a project to manage a collection of source files that put together achieve some integral purposes.
- The newly created project will be added to the current workspace.
- You may also import existing projects into the workspace.

Creating a Project

- Choose **File → New → Java Project…**
- In the **New Java Project** dialogue box
  - Enter the project name
  - Choose where the project is to be stored (you can use default location which is a subdirectory in the current workspace)
- You may click **Next** and specify further information
- Click **Finish** to create the project



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: Circle

☑ Use default location

Location: C:\Users\scmyl2\workspace\Circle

JRE

○ Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jre1.8.0_66

● Use default JRE (currently 'jre1.8.0_66')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

□ Add project to working sets

Working sets:

Creating a New Class

- To create a new class in the current project
  - Right click on the project name in the **Package Explorer**
  - Choose **New → Class**
  - Enter the name of the class, and specify the superclass (the default is **java.lang.Object** if the class to be created does not have a dedicated superclass)
  - Click **Finish**

Editing the code

- After clicking on the **Finish** button and Eclipse will create the skeleton code for the class.
- Add the following code within the class

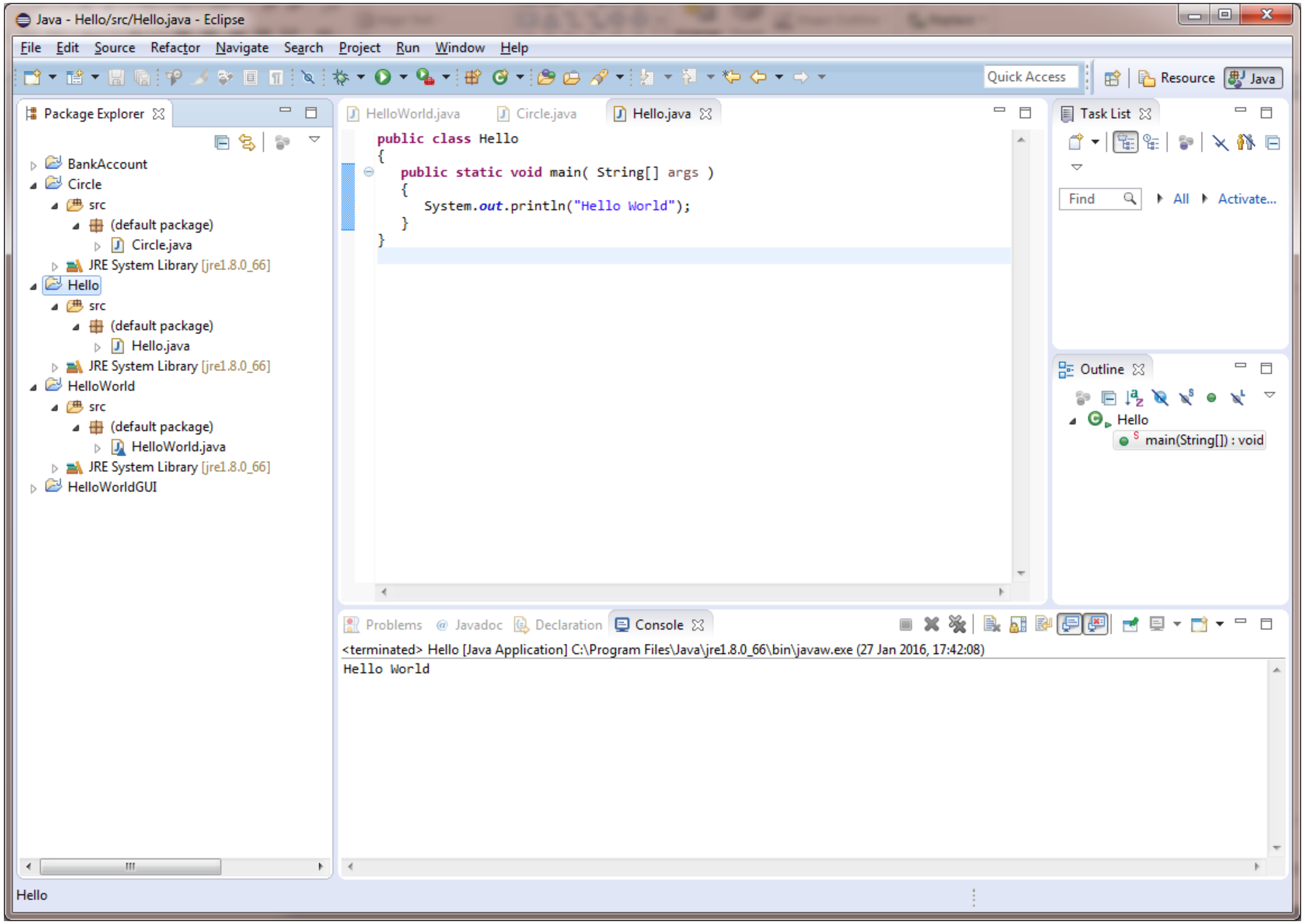
```
public static void main( String[] args )
{
   System.out.println("Hello World");
}
```

- When entering code, you can benefit from
  - Syntax Highlighting, indentation, code prompt/completion, bracket matching etc.
  - Java documentation in the Javadoc panel
  - Error highlighting

Building and Running a Project

- Eclipse provide automatic compiling which means you don’t need to manually compile the project before running it.
- The compilation process is still needed but will be done automatically.
- Select a project in the Package Explorer, and click the **Run As** button ▶ on the toolbar (choose **Java Application** in the **Run As** dialogue box when the project is first run).
  - If the project compiles properly, it will run and the output will be displayed in the **Console** panel.
  - If there are errors, they will be shown in the **Problems** panel.

Building and Running a Project (cont.)

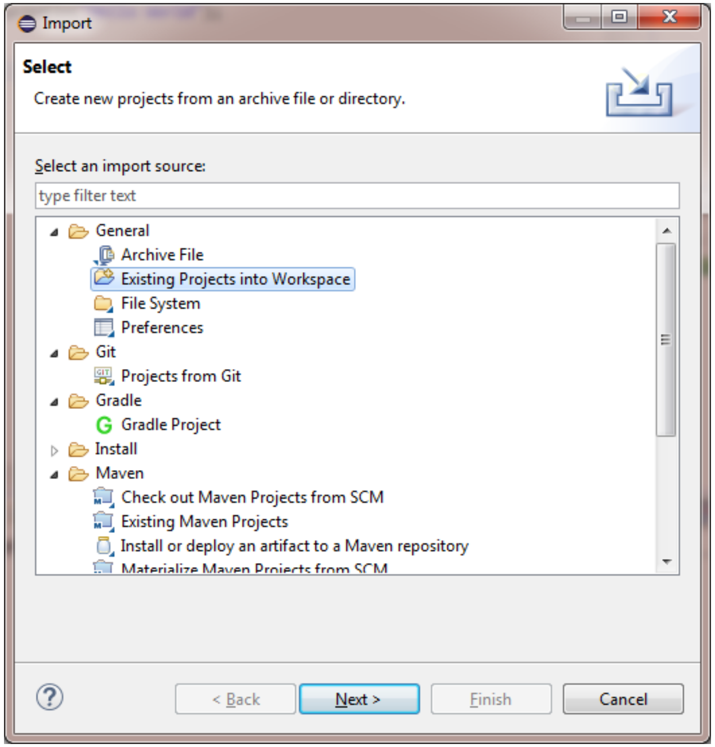


```
public class Hello
{
   public static void main( String[] args )
   {
      System.out.println("Hello World");
   }
}
```

```
<terminated> Hello [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (27 Jan 2016, 17:42:08)
Hello World
```

Importing Projects

- To import existing Eclipse projects to your current workspace
  - Choose **File → Import**
  - In the **Import** dialogue box, choose **General → Existing Projects into Workspace**
  - Click **Next**
  - In the **Import Projects** dialogue box, choose **Select root directory** and browse to the directory that contains Eclipse projects (can be at higher level)
  - Select **Copy projects into workspace**
  - Select listed projects you wish to import
  - Click **Finish**
  - The selected projects will be imported to the current workspace



Import

Select

Create new projects from an archive file or directory.

Select an import source:

- Java Basics, Object Oriented Concepts
- Introduction to Eclipse
- **JavaFX Simple Graphics**

Graphical User Interface (GUI)

- Graphical user interface (GUI) utilises windows and controls to interact with users.
- A common paradigm for the GUI is known as WIMP (Window, Icon, Menu and Pointing device).
  - often more friendly to users
  - the major choice of modern programs
  - often built with windows and various operable elements such as menus, buttons, and textboxes

Java GUI Libraries

- Java provides three major libraries for GUI
- **Abstract Windowing Toolkit (AWT)**
  - deals with user interface elements by delegating them to the native GUI toolkit on the specific platform (e.g. Windows, Mac OS).
  - the same Java code can be run on different platforms
  - subtle differences in appearance and behaviours exist on different platforms
- **Swing**
  - built on top of AWT library
  - that **paints** user interface elements
  - ensures consistent look and feel across platforms
  - Swing applications may still use AWT features
- **JavaFX**
  - Most recent library for Java GUI, simple to use and support advanced features
  - Provided as an extension in previous Java versions, but integrated in JDK 8
  - JavaFX and Swing can be used together
- We use JavaFX for simple graphics here; detailed GUI using JavaFX will be covered later.

Hello, World GUI Program

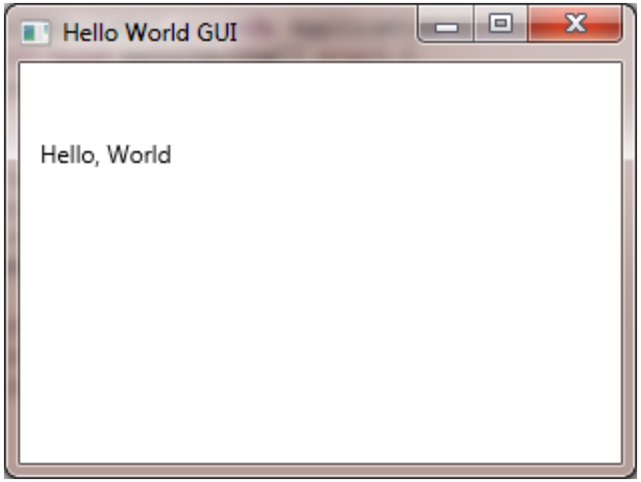
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.text.Text;

public class HelloWorldGUI extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200);
        Text t = new Text(10, 50, "Hello, World");
        root.getChildren().add(t);

        primaryStage.setScene(scene);
        primaryStage.setTitle("Hello World GUI");
        primaryStage.show();
    }
}
```

Hello, World GUI Program

- Result



Hello, World

Hello, World GUI Program Explained: class and main

- JavaFX provides a blueprint for GUI applications as class **javafx.application.Application.**
- A specific JavaFX GUI application is a tailored application, thus by the concept of OOP, a descendent of **Application** class, in this case, **HelloWorldGUI** class is declared that **extends Application**.
- The entry point of a GUI program is the same as a consoled based program, i.e. the **main** method. It only needs to contain the following code to launch the application (args is the input arguments – we will cover this later):

```
    launch(args);
```

Implementing **start** Method

- The main task to begin building a GUI application is to implement the **start** method
  - `public void start(Stage primaryStage)`
- The input to this is a **javafx.stage.Stage** object created by JavaFX
  - **Stage** is the highest level container in JavaFX
  - With a stage, you can specify the **Scene** associated with the **Stage** by **setScene()**, where **Scene** is the container of actual GUI elements
  - You can specify the window title by using **setTitle()**
  - You can use **show()** to make the stage visible

Scene and Nodes

- **Scene** contains all the GUI elements, which are formed in a Scene node hierarchy
  - When a **Scene** object is created, you need to specify the root node, width and height of the Scene
  - The root node can be an instance of **javafx.scene.Group**, which allows a variety of other nodes to be added (such as text, shapes etc.), thus forming a hierarchy
- To add **Text** node, we first create an instance of **javafx.scene.text.Text** object (x, y coordinates and text to display), and add it to the root group:

```
Text t = new Text(10, 50, "Hello, World");
root.getChildren().add(t);
```

Importing Classes

- Java organises classes in some hierarchies
  - To refer to a specific class, you can use the fully qualified name, e.g. **javafx.application.Application**
  - Or the class can be imported using

```
import javafx.application.Application;
```

  and the class can then be referred to as **Application**.

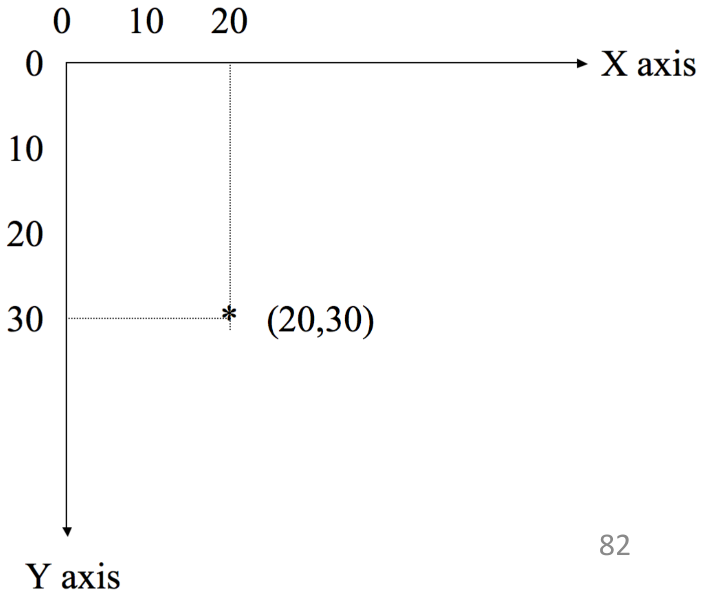
  - To import a collection of classes in the same level of hierarchy, use '*', e.g.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.text.*;
```

- Eclipse IDE can suggest classes to import

Java Coordinate System

- By default, the top left corner of the screen has coordinates (0,0).
- Coordinate units are measured in ***pixels*** (a display monitor’s smallest unit of resolution)
- Note that certain areas of a frame are likely to be covered by system.



Text Styles

- **Font** class (**javafx.scene.text.Font**) can be used to specify the font for text
  - **Font.font()** method returns a nearest font with specified font name, font weight (e.g. **FontWeight.BOLD**), font posture (e.g. **FontPosture.ITALIC)** and font size
  - Use **Text.setFont** to specify the font for a text object

Example: **Display String**

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.text.*;

public class DisplayString extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200);
        Text t = new Text(100, 50, "Graphics");
        t.setFont(Font.font("Helvetica", FontWeight.BOLD, FontPosture.ITALIC, 24));
        root.getChildren().add(t);

        primaryStage.setScene(scene);
        primaryStage.setTitle("Display String");
        primaryStage.show();
    }
}
```

Colour

- By default, lines (or polylines) are drawn with a **black** stroke (outline), and other shapes are drawn with a **black** fill.
- To change the colour, you need to
  - create an object of type **Color** (**javafx.scene.paint.Color)**
  - use **setStroke** and **setFill** to change the colour of the outline and fill
  - This works for text, shapes etc.
- All colours in **Java** are defined in terms of the **RGB** colour model
  - specifying amounts of the primary colours **red**, **green** and **blue**
  - The amounts are given as **double** values and vary from **0.0** (primary colour not present) to **1.0** (maximum amount of primary colour present)
- To create a **Color** object using RGB information, use

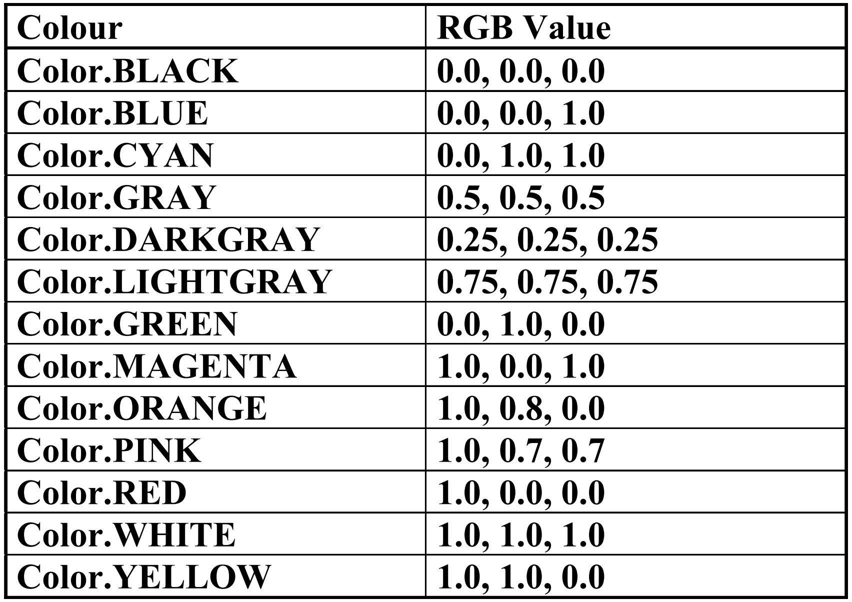
```
    Color.color(red, green, blue)
```

Colour (cont.)

- For example, to create a bright purple (magenta) colour:

```
Color magenta = Color.color( 1.0, 0.0, 1.0 );
```

- Many predefined colour constants for convenience, e.g.



Colour	RGB Value
Color.BLACK	0.0, 0.0, 0.0
Color.BLUE	0.0, 0.0, 1.0
Color.CYAN	0.0, 1.0, 1.0
Color.GRAY	0.5, 0.5, 0.5
Color.DARKGRAY	0.25, 0.25, 0.25
Color.LIGHTGRAY	0.75, 0.75, 0.75
Color.GREEN	0.0, 1.0, 0.0
Color.MAGENTA	1.0, 0.0, 1.0
Color.ORANGE	1.0, 0.8, 0.0
Color.PINK	1.0, 0.7, 0.7
Color.RED	1.0, 0.0, 0.0
Color.WHITE	1.0, 1.0, 1.0
Color.YELLOW	1.0, 1.0, 0.0

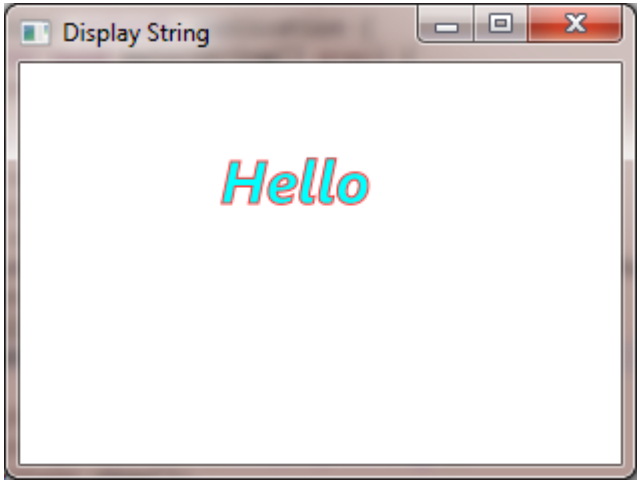
Example: Display Coloured String

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.paint.Color;
import javafx.scene.text.*;
public class Colour extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200);
        Text t = new Text(100, 70, "Hello");
        t.setFont(Font.font("Times", FontWeight.BOLD, FontPosture.ITALIC, 30));
        t.setStroke(Color.color(1.0, 0.25, 0.25));
        t.setFill(Color.AQUA);
        root.getChildren().add(t);

        primaryStage.setScene(scene);
        primaryStage.setTitle("Display String");
        primaryStage.show();
    }
}
```

Example: Display Coloured String

- Result:
  - Note the different colours for the outline and interior



Hello

Shapes

- JavaFX provides classes to represent shapes:
  - Import shape related classes using:

```
import javafx.scene.shape.*;
```

- **Line** class: represents a line segment
  - setStartX(), setStartY(): set the x, y coordinates of the starting point
  - setEndX(), setEndY(): set the x, y coordinates of the end point
- Once created, add shape objects to the scene hierarchy (similar to text)

Draw Line Example

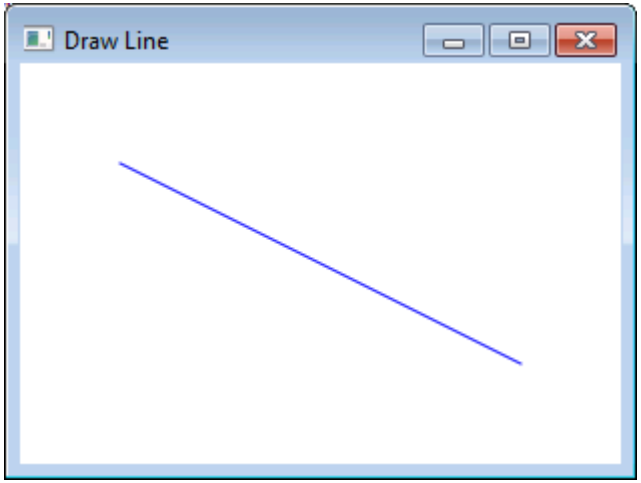
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.scene.shape.*;

public class DrawLine extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200);
        Line line = new Line();
        line.setStartX(50);
        line.setStartY(50);
        line.setEndX(250);
        line.setEndY(150);
        line.setStroke(Color.BLUE);
        root.getChildren().add(line);

        primaryStage.setScene(scene);
        primaryStage.setTitle("Draw Line");
        primaryStage.show();
    }
}
```

Draw Line Example

- Result:
  - Note: using **setStroke()** to set the line colour
  - Lines can be further customised using **setStrokeWidth, setStrokeType** etc.



Shapes (cont.)

- **Rectangle** class: represents a (normal or rounded) rectangle
  - setX(), setY(): set the x, y coordinates of the top left corner
  - setWidth(), setHeight(): set the width and height of the rectangle
  - setArcWidth(), setArcHeight(): for rounded rectangle, specify the width and height of the rounded corners

Example: DrawRectangles

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.scene.shape.*;

public class DrawRectangles extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 400, 300);

        // Rectangle (20, 50) - (120, 250)
        Rectangle rect1 = new Rectangle();
        rect1.setX(20);
        rect1.setY(50);
        rect1.setWidth(100);
        rect1.setHeight(200);
        rect1.setStroke(Color.BLUE);
        rect1.setFill(Color.AQUA);
        root.getChildren().add(rect1);
```

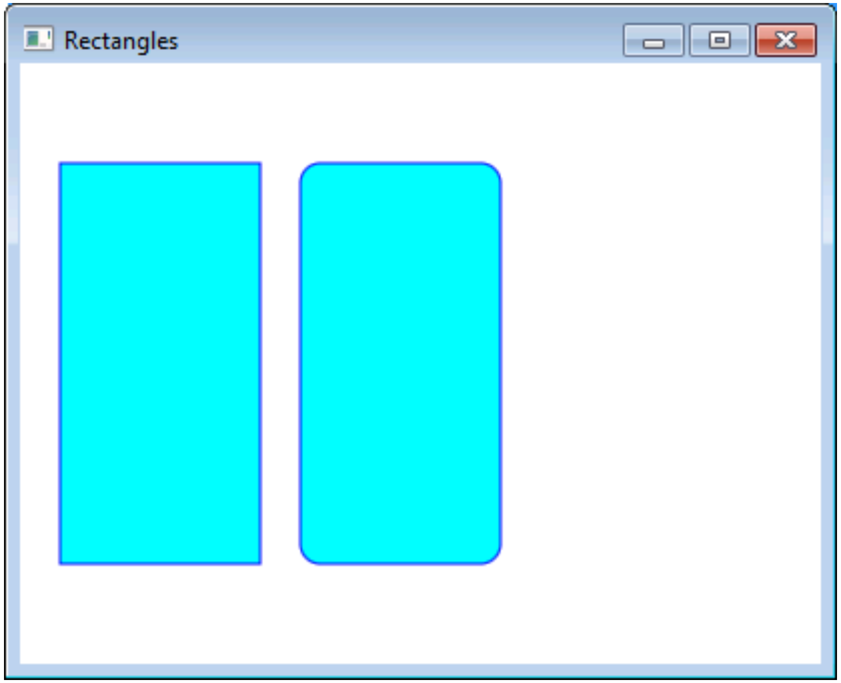
Example: DrawRectangles (cont.)

```
        // Rounded rectangle (140, 50) - (240, 250)
        Rectangle rect2 = new Rectangle();
        rect2.setX(140);
        rect2.setY(50);
        rect2.setWidth(100);
        rect2.setHeight(200);
        rect2.setArcWidth(20);
        rect2.setArcHeight(20);
        rect2.setStroke(Color.BLUE);
        rect2.setFill(Color.AQUA);
        root.getChildren().add(rect2);


        primaryStage.setScene(scene);
        primaryStage.setTitle("Rectangles");
        primaryStage.show();
    }
}
```

Example: DrawRectangles (cont.)

- Result:



Shapes (cont.)

- **Circle** class represents a circle
  - setCenterX(), setCenterY(), setRadius(): set the x, y coordinates of the centre and the radius
- **Ellipse** class: represents an ellipse
  - setCenterX(), setCenterY(): set the x, y coordinates of the centre
  - setRadiusX(), setRadiusY(): set the radius along the x and y direction

Example: DrawOvals

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.scene.shape.*;

public class DrawOvals extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage primaryStage) {
        Group root = new Group();
        Scene scene = new Scene(root, 400, 300);

        // A red oval
        Ellipse oval1 = new Ellipse();
        oval1.setCenterX(200);
        oval1.setCenterY(150);
        oval1.setRadiusX(100);
        oval1.setRadiusY(50);
        oval1.setFill(Color.BLUE);
        root.getChildren().add(oval1);
```

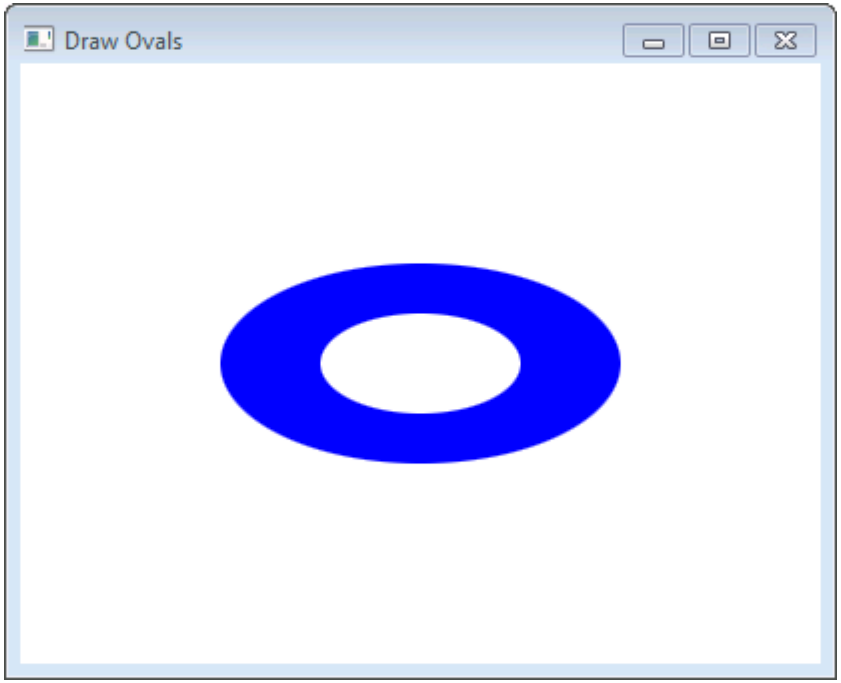
Example: DrawOvals (cont.)

```
        // With a white oval overlaid at the centre
        Ellipse oval2 = new Ellipse();
        oval2.setCenterX(200);
        oval2.setCenterY(150);
        oval2.setRadiusX(50);
        oval2.setRadiusY(25);
        oval2.setFill(Color.WHITE);
        root.getChildren().add(oval2);

        primaryStage.setScene(scene);
        primaryStage.setTitle("Draw Ovals");
        primaryStage.show();
    }
}
```

Example: DrawOvals (cont.)

- Result:
  - Note: By default the shapes added later cover the shapes added previously
  - Use **setOpacity()** to set the opacity of shapes



JavaFX GUI

- JavaFX provides more graphics capabilities:
  - Shapes such as polygons, polylines, arcs, etc.
  - Colour gradients
  - Effects for blending, blurring etc.
- We will cover more detailed JavaFX GUI later
  - GUI elements: buttons etc.
  - Event handling (response to user interactions)
  - etc.