



**Murdoch**  
UNIVERSITY

# Topic 3: Objects and the Java Language

ICT373: Software Architectures

# Recap

- WWW based Client-server programming
- Validation
- JavaScript

# Overview

- Java overview
- Objects
- Java revision
- O-O design and The Unified Modelling Language (UML)

## **Reading:**

Textbook (11<sup>th</sup> ed.) Chapters 1-8

# Learning Objectives (1/2)

- Give an overall description of the Java language and the Java SDK.
- Explain the Java architecture, and describe the role of byte-codes, Java API and the JVM in using Java code.
- Explain why Java is regarded as architecture neutral.
- Be able to write and document applications in Java that conforms to normal standards and style.
- Describe Java's object model: the role of primitives, objects, classes and references to objects.
- Explain how Java objects are created and destroyed?
- Explain the use of static data and functions.
- Explain the use of wrappers for primitives.

# Learning Objectives (2/2)

- Be able to use classes from the standard libraries (Java Platform Packages).
- Be able to use arrays in a Java program.
- Be able to use the Javadoc documentation facility.
- Be able to locate information from the Java on-line documentation.
- Describe the general principles of Object-Oriented Software Engineering (OOSE).
- Describe the main steps in Object-Oriented design.
- Describe the UML notation for individual classes and associations between them.

# What is Java?

- a free object-oriented general purpose programming language designed to be able to be used easily with the Internet.
- a popular choice for implementing O-O designs and for developing web-based applications.
- *architecturally neutral*, i.e. a program can run on a variety of different types of machine.

# The JDK (1/2)

- Java development tools, including the compiler, and debugger
- Java Runtime Environment (JRE) including the Java Virtual Machine (interpreter) and Java class libraries (APIs)
- supporting tools and components
- demo programs
- documentation

Commercially available integrated development environments (IDEs) allow these components to be used together in a user-friendly, productive way. Eg, NetBeans, Eclipse

In lectures and labs we will mostly use the NetBeans IDE and the basic command line version.

# The JDK (2/2)

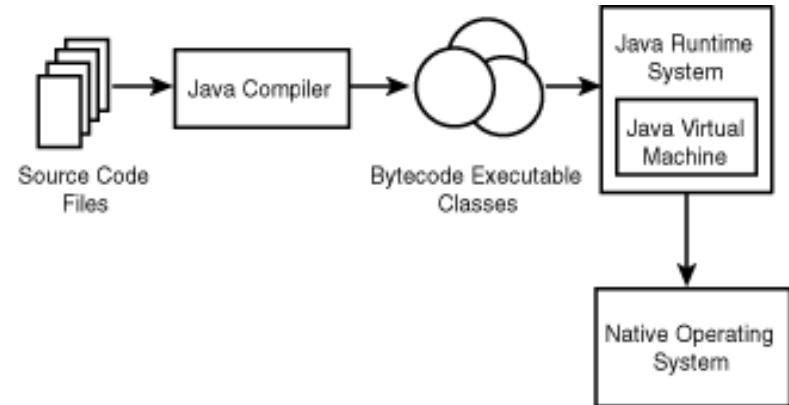
- The Java compiler is called “**javac**”.
  - Use it to compile a text file containing the java source code.
  - The result is a file of (sort of) executable code called Java ***bytecodes***.
- To execute the bytecode file, needs a Java Virtual Machine (JVM).
  - Which one to use depends on the environment and operating system.
- For command line running of ordinary programs (=application programs), we use the interpreter (JVM) called “**java**” (appropriate versions of this are supplied for Windows, Linux, Mac, etc ).



# Java Architecture (1/2)

- Java's architecture arises out of four interrelated technologies:
  - the Java programming language
  - the Java Application Programming Interface (API)
  - the Java class file format
  - the Java virtual machine (JVM)
- A program in source files (written in the Java programming language) is compiled into Java class files, and the class files are run on a JVM.
  - System resources (e.g. I/O) are accessed by calling methods in the classes that implement the Java API.

# Java Architecture (2/2)

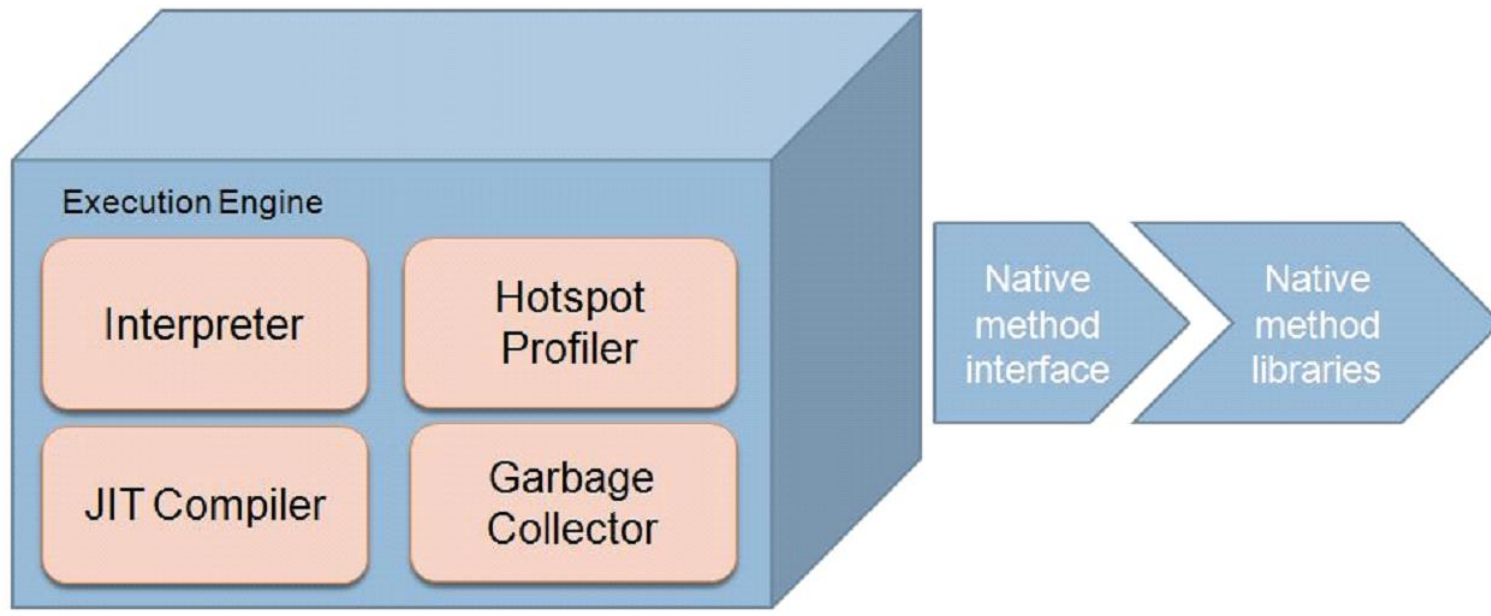


- The Java API is a set of runtime libraries that give you a standard way to access the system resources of a host computer.
- The JVM and Java API together form a platform for which all Java programs are compiled – called **Java 2 Platform** or the **Java runtime system**.
- The compiled Java program, in byte code, can run on any machine where the Java Platform (JRE) is present.
- Java programs can run on many different kinds of computers because the Java Platform can itself be implemented in software.

# Architecture Neutral

- The compiled Java program, in bytecodes, should run in exactly the same way under different operating systems, i.e., Java is Architecture Neutral.
- We say that the bytecodes are being run on a JVM and that the various interpreters are implementations of the JVM.
- Java source code can be compiled on any machine (with SDK), the compiled code can be sent to any other machine (e.g., over the Internet) and we know that the program will run as planned provided the second machine has an implementation of the JVM e.g., in a browser.

# Java Virtual Machine (JVM)



# Java Virtual Machine (JVM) (1/3)

- The Java virtual machine is an abstract computer.
  - Its specification enables it to be implemented on a wide variety of computers and devices.
- JVM's main job is to load class files and execute the bytecodes they contain.
  - JVM contains a *class loader*, which loads class files from both the program and the Java API.
  - The bytecodes are executed in an **execution engine** (part of the virtual machine) which can vary in different implementations.
  - The simplest form of execution engine just interprets the bytecodes one at a time. That is why sometimes the JVM is called the **Java interpreter**.

# Java Virtual Machine (JVM) (2/3)

- **Just-in-time (JIT) compilation**, also known as **dynamic translation**, is compilation done during execution of a program – at run time – rather than prior to execution.
  - Optimise
  - Most often this consists of translation to machine code, which is then executed directly, but can also refer to translation to another format.
- **Garbage collection** is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program.
  - Unlike C/C++ where you need to free unused memory explicitly.
- When running on a JVM that is implemented in software on top of a host operating system, a Java program interacts with the host by invoking **native methods**.

# Java Virtual Machine (JVM) (3/3)

- While you can write applications entirely in Java, there are situations where Java alone does not meet the needs of your application. Programmers use the JNI to write *Java native methods* to handle those situations when an application cannot be written entirely in Java.
  - A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular computer.
  - Native methods are stored in a dynamically linked library whose exact form is platform specific. While Java methods are platform independent, native methods are not.
- The implementation of the JVM can also include:
  - a safe execution environment (called the Sandbox) which checks for various file access and network access security violations.
  - Note that Java was designed to run its applets within a sandbox of safety which prevents the applet from writing to disk or accessing memory outside the sandbox, or erasing from a disk (as viruses do).

# An Example JAVA Program

```
// HelloWorld.java

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello Class");

        /* To run from command prompt, eg use:
        java HelloWorld Greetings ICT306 Students
        */

        for (int i = 0; i<args.length; i++)
            System.out.print(args[i] + " ");
        System.out.println();
    }
}
```

Compiling and executing this program results in:

Hello Class

being output on the screen.

run from command prompt with command line arguments, as follows: java HelloWorld Greetings ICT373 Students



# Description of the program

- This program defines a publically accessible class of objects called "HelloClass".
- The class has one method (a static or class method) which is the main method and so is called when the program is executed. The method can be given command line arguments which, in this example, are outputted to the screen.
- When the method is called the string argument "Hello Class" is given to the println method of the out object (the current value of a public member variable) belonging to the System class (defined in a library which manages the program's environment).
- The next statement outputs the contents of the String array args.

# Another program

```
/** New_IO.java
 * Uses the Scanner class of Java API to read input
 * Also, uses the printf method for formatted output
 */
import java.util.Scanner;
// Scanner class must be imported

public class NEW_IO{
    public static void main(String[] args){

        Scanner keyboard = new Scanner(System.in);
        int temp1, temp2;

        System.out.printf("%s%s\n%s\n", "Welcome to ICT373 !", "The Unit Coordinator is Ferdous Sohel.");
        System.out.printf("\nEnter 2 integers:");

        temp1= keyboard.nextInt();
        temp2= keyboard.nextInt();

        keyboard.nextLine(); // To skip past the end of line marker

        System.out.printf("%s%d%s%d\n\n",
            "The numbers are:", temp1, " and ", temp2);

        if (temp1 >= temp2)
            System.out.printf("Today's maximum forecast temperature is %d\n\n", temp1);
        else
            System.out.printf("Today's maximum forecast temperature is %d\n\n", temp2);

        System.out.println("Enter your name:");

        String name = keyboard.nextLine(); // read a string

        System.out.println("You entered:");
        System.out.printf("%s\n", name);
        System.out.println();

    } //end of main
}
```

- Other methods of Scanner class include:

nextInt(), nextFloat(), nextDouble(), nextLong(), hasNext(), hasNextInt(), hasNextFloat(), hasNextLine() etc.

- See the Scanner class documentation for full details.
- Online Java documentation can be found via <http://docs.oracle.com/javase/8/docs/>



**Murdoch**  
UNIVERSITY

# Objects



**Murdoch**  
UNIVERSITY

# Objects: Key characteristics of an O-O language

- Everything is an object.
- A program is a bunch of objects telling each other what to do by sending messages.
- Each object has its own memory made up of other objects.
- Every object has a type.
- All objects of a particular type can receive the same messages.

# Objects

- Each of the *values* that are manipulated in a Java will be of a certain *type*. The type determines what can be done with the value.
- Most of the values will be *objects* of *class* type, i.e. a type defined by a **class** definition in the program or in another file or library file.
- The only other values are values of a *primitive* built-in type such as **boolean**, **char**, **int**, **float** or **double**.
- Each variable in the program also has a type, either class type or primitive type. The type of a variable must be declared in the program.
- As the program runs, at each moment, each variable may (or may not) be associated with a value of the same type.
- A variable of primitive type stores a direct representation of its value.
- A variable of a class type may contain a reference to an object (of that type) that is stored elsewhere. The variable may instead contain a **null** reference.

# Creation of Objects and Declaration of Variables

- To use a new object, it needs to be created. This is done via the **new** keyword in the format

*type var = new type (args);* where:

- *var* is a variable of the right type to reference the object;
  - *type* is the (class) type of the new object;
  - *args* is a possibly empty list of values which determine which object of that type gets constructed.
- a **constructor** for that class is called with the supplied arguments to construct the object.
    - The code for construction (if any) resides in the constructor method in the code for the class definition wherever that is.

# Creation of Objects and Declaration of Variables

- **When new is used,**
  - space for the new object is allocated on the heap (area of memory),
  - the object is constructed there and the variable is given a reference to the object.
- Declaring variables can be done anywhere in the program (before the variable is used) and should include initialization:

**float g = 9.8;**

**String greeting = new String ("hello");**



# Destroying Objects

- When a variable goes out of scope then the program might lose all reference to a particular object.
  - The Java runtime system will notice this fact and, knowing that the object can no-longer play any part in the program, a background process called the **garbage collector** eventually releases the memory for that object.
- No need for the programmer to explicitly reclaim the memory.
- Some objects may require certain actions to be taken before they disappear. E.g., a network connection or file is to be closed properly.
  - the garbage collector looks for and runs a method called **finalize()** if such a method is supplied for the class of object which is being destroyed.

# Doing things to objects

- Objects in a class have
  - **fields** = named components of the object of fixed type (also known as instance variables or data members); and
  - **methods** = named functions with typed arguments, return type and coded effect on the fields.
- The actual values of the fields for a particular object may change during running of the program and together they define the **state** of the object.

# Doing things to objects

- The state can be changed (subject to appropriate permissions) and/or state info got by:
  - direct assignment to the named field in the object currently referenced by a variable

***var.field = value;***

- or direct access

***var2 = var.field;***

- calling a method for the referenced object

***var2 = var.method(args);***

- Variables declared **final** cannot be changed.
- Values can be assigned (using shallow copy):

***var2 = var1;***

# Methods and Controls

- In programs things are done in methods.
- Every method belongs to a class and is defined within the class declaration in code like:

```
modifiers return-type method-name ( typed parameters) {  
    body  
}
```

- When the method is called for a particular object of that class, the parameters are given the values in the corresponding arguments and the body is executed.
  - When the statement **return** *value* is encountered in the body then the specified value is returned.
- There need be no return statement if the return type is **void**. No value is returned by such a method.

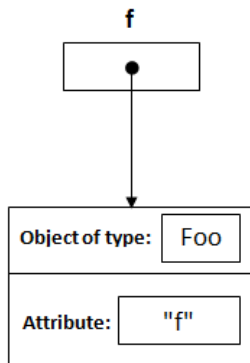
# Methods and Controls

- Control in the body is with **if/if-else, switch, for, while, do-while** etc.
- Parameters of primitive type are **passed by value** to the method.
- Class type parameters also have their argument passed by value. However, the value is a **reference to an object!**
  - the argument object may be changed by the method.
  - On the other hand, the argument variable is not passed and so cannot have its value changed – it keeps referring to the original object even if that object is changed or the parameter is made to refer to a different object.

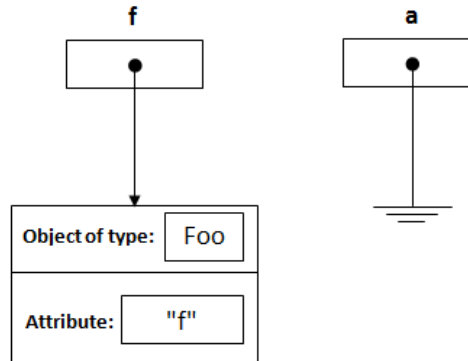
# Call by value / reference

```
public class Main{
    public static void main(String[] args){
        Foo f = new Foo("f");
        changeReference(f); // It won't change the reference!
        modifyReference(f); // It will modify the object that the reference variable "f" r
    }
    public static void changeReference(Foo a){
        Foo b = new Foo("b");
        a = b;
    }
    public static void modifyReference(Foo c){
        c.setAttribute("c");
    }
}
```

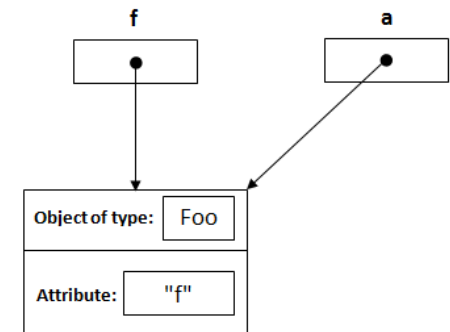
Foo f = new Foo("f");



public static void  
changeReference(Foo a)

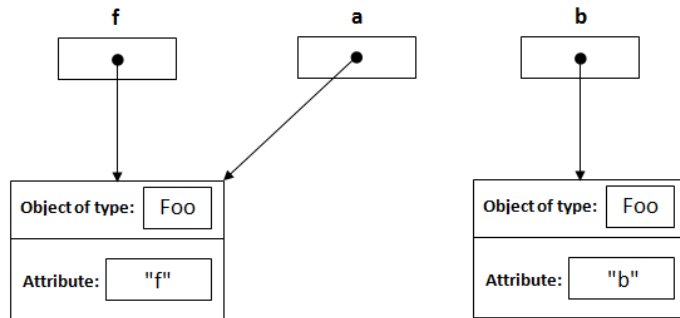


changeReference(f);

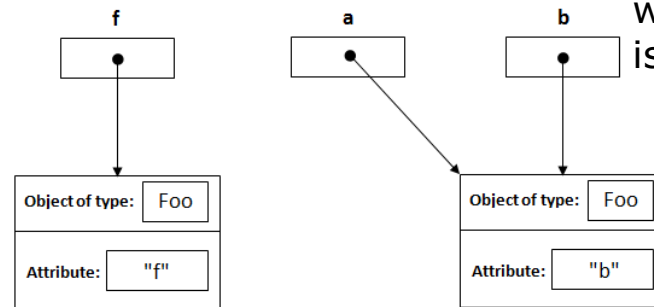


# Call by value / reference

Foo b = new Foo("b");

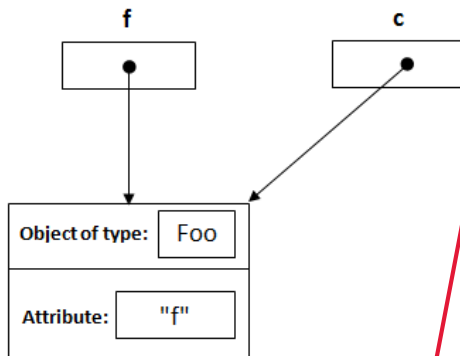


a = b

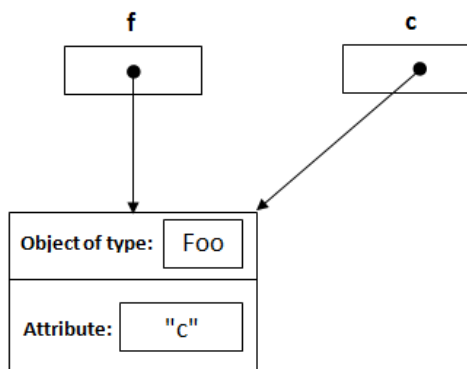


a = b is re-assigning the reference a NOT f to the object whose its attribute is "b".

modifyReference(Foo c)/modifyReference(f)



c.setAttribute("c");



will change the attribute of the object that reference c points to it, and it's same object that reference f points to it.

# Static members

- Members usually belong to objects of the class. However, sometimes it is useful to have members, which belong to the class as a whole: *class members* and *class methods*.
- These are declared in the class with the modifier **static**.
- If a data member is declared static then only one such value exists at any one time during the running of the program. It can be accessed by *classname.field*. The objects of the class can all access this one value. The value is initialized (as per code in the declaration) when the class is “loaded” when it is first needed.
- Static methods can also be defined. They can be called via *classname.method(args)* even if no objects of that class exist. They can only access static data members of the class.
- The **main** method of class X is run first when X.java is interpreted. It is a static method (there is no X object at the start).
- The **main** method has one array parameter which is an array of String objects given values in the command line arguments.



# Notes on Primitive Types: *Wrappers*

- Sometimes it is convenient to represent values of primitive types as class objects. To allow this, each primitive type is provided (in the java.lang library package) with a corresponding class called a **wrapper class**.
- Examples are: Integer, Double and Character classes. Eg  
**Integer fred = new Integer(2);**
- (This is useful as some container classes are only able to contain objects – see later).
- The above type conversion from a value of a primitive type (eg, int) to a corresponding object of its associated wrapper class (eg, Integer) is called **boxing**.
- Unwrapping, e.g., **int i = fred.intValue();**
- This reverse conversion from an object of wrapper class to its associated primitive type is called **unboxing**.

# Wrappers

- Starting with version 5.0, Java will automatically do this boxing and unboxing (called **autoboxing and auto-unboxing of primitives**). Eg, In Java 5.0 the above examples can be written as

```
Integer fred = 2;    // autoboxing
```

```
int i = fred;    // fred is an object of class Integer
```

- The automatic boxing and unboxing also apply to parameters,
  - a primitive argument can be provided for a corresponding formal parameter of the associated wrapper class, and a wrapper class argument can be provided for a corresponding formal parameter of the associated primitive type.

# Wrappers

- To convert a string containing an integer (eg "23") to an **int**, use Integer class static methods as follows:

```
int i=Integer.valueOf(str).intValue();
```

```
int number = Integer.parseInt (inputString);
```

```
String str = " 99.95 ";
```

```
double d = Double.parseDouble(str.trim());
```

```
// if the string has leading or trailing whitespaces.
```

- Methods for converting strings to the corresponding numbers are also available. E.g., Integer.toString(123), Long.toString(123), and Double.toString(99.95)

# Notes on Primitive Types (cont.)

- **Automatic type conversion**, as in C++, e.g.

```
float g = 6 + ( 2 / 3 );    // (? what ends up in g ?)
```

- **Explicit casting** can be done as in C++, eg

```
int n = (int)(6+ ((float)2) / ((float)3));
```

# Library Classes

- Java includes many ready-built classes. These supply well-designed data structures or help with input and output, networking, and GUIs.
- Some library classes are used directly by the compiler.
- Information on the classes can be found in the on-line documentation via the **package.html** page.
- The classes in the package **java.lang** are automatically available to any program. E.g., **java.lang.System**, **java.lang.Integer** and **java.lang.Math**.
- Other classes need to be *imported* either on their own

**import** *packagename.classname*;

or with the whole package

**import** *packagename.\**;

Eg, **java.util.Random**, **java.util.Date** or **java.math.BigInteger**.

# Library Classes

- Note that **java.lang.Math** (which is automatically available) contains all the basic mathematical functions like absolute values, random numbers and square roots.
- These are available as static methods of the class so you don't have any need to create a **java.lang.Math** object but you do need to call them properly, e.g. **Math.abs(x)**.

# An example

```
class Fraction {  
    int numerator, denominator;  
  
    Fraction(int a, int b) {  
        numerator=a;  
        denominator=b;  
    }  
  
    void print() {  
        System.out.println( numerator + " / " +  
                             denominator );  
    }  
}  
  
public class JavaApplication7 {  
    public static void main(String[] args) {  
  
        Fraction f = new Fraction(2,3);  
        f.print();  
    }  
}
```

# Arrays in Java

- An array is a (linear) collection of objects of the same type. Each array is itself an object (but there is no class of arrays).
- A variable can be declared to be capable of referring to arrays of specific types, eg,

```
int[] a;
```

```
Fraction[] fa;
```

- A new array object can be created, eg,

```
a = new int[7];
```

```
fa = new Fraction[2];
```

- Combined declaration + object creation;

```
int [] a = new int[7];
```



# Arrays in Java

- Now `a[0]`, ..., `a[6]` exist and are all 0. However, `fa[0]` and `fa[1]` exist but are **null** references. You need to initialize these references, e.g.,  

```
fa[0] = new Fraction(1,2);  
fa[1] = fa[0];
```
- Each array has a data member **length** which can be looked up but not changed, eg,  

```
int le = fa.length;
```
- Run-time errors result from trying to use an invalid array index.

# Comments

```
// single line comments
/* and multiline comments
in C++ style */

/**
Here is a program to show javadoc at work on documentation comments. Uses HTML tags.
@author Ferdous Sohel
@see java.lang.Math
@see <a href="http://handbook.murdoch.edu.au/units/?unit=ICT373"> See here for more information.</a>
*/

public class fred {
    /**
     * Print the absolute value of -2.
     * @param args not used
     */
    public static void main( String args[]){
        System.out.println(Math.abs(-2));
    }
}
```

# Comments: Javadoc

- If you run the tool “**javadoc**” on a program with documentation comments you get a nice HTML page summarizing the classes defined in it. For more info on javadoc, visit <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html>



**Murdoch**  
UNIVERSITY

# Unified Modelling Languages (UML)



**Murdoch**  
UNIVERSITY

# Object Oriented Design

- The general principles behind OOSE are:
  - use of abstraction, encapsulation and interfaces
  - re-use of exchangeable components
  - inheritance and polymorphism
  - designing for problem domains (vs specific problems).
- The specific approaches include OO methods for design. Eg, Rumbaugh's Object Modelling Technique -OMT (OO Modeling and Design, 1991), Booch's method (OO Analysis and Design, 1994), and Jacobson's method (OOSE, 1992) etc.
- **Unified Modelling Language (UML).**

# UML

## UML

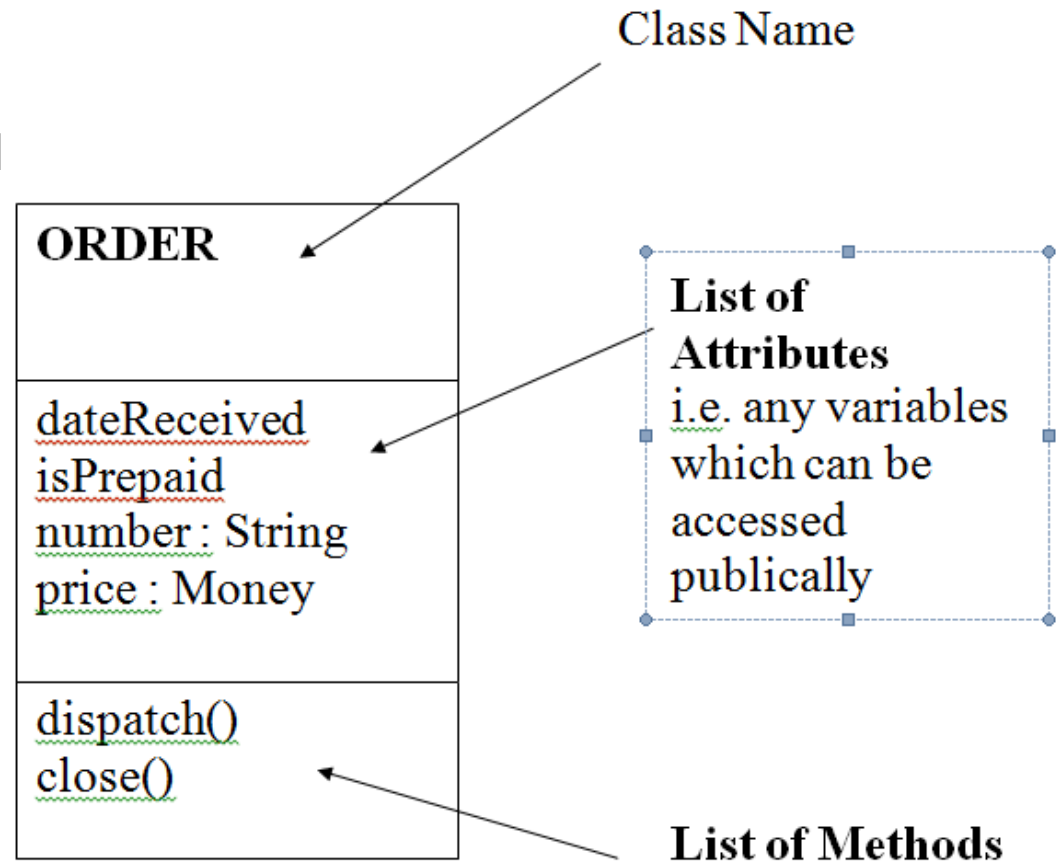
Graphical representation  
scheme

Enables developers to model  
object-oriented systems

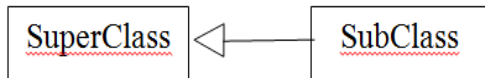
Flexible and extendible

## UML – a single class

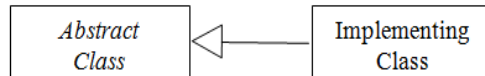
This is a class icon  
representing a single class



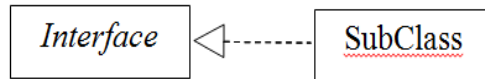
# Class diagram



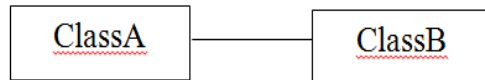
Inheritance



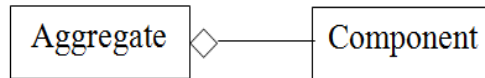
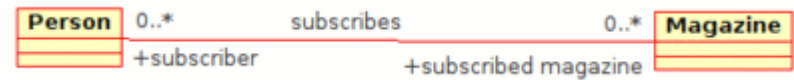
Inheritance



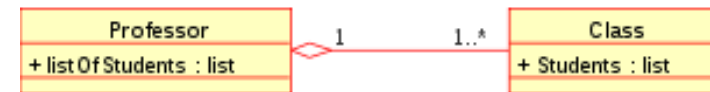
Implementation



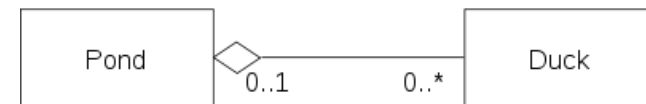
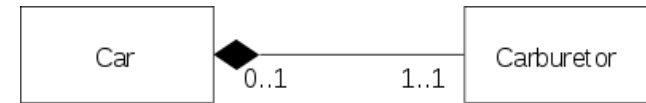
Association



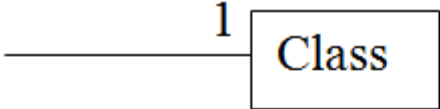

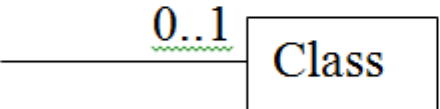
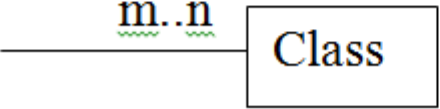
Aggregation,  
non-exclusive



Aggregation,  
composition



# Cardinality (multiplicity) notation

	<u>exactly one</u>
	<u>many</u> <u>(zero or more)</u>
	<u>optional</u> <u>(zero or one)</u>
	<u>numerically</u> <u>specified</u>



# Properties and visibility

## Class Name

-privateDataField : data\_type  
#protectedDataField : data\_type  
+publicDataField : data\_type  
staticDataField: data\_type  
+publicMethod(argument...): data\_type  
-privateMethod(argument...): data\_type  
+publicStaticMethod(argument): data\_type  
abstractMethod(argument): data\_type

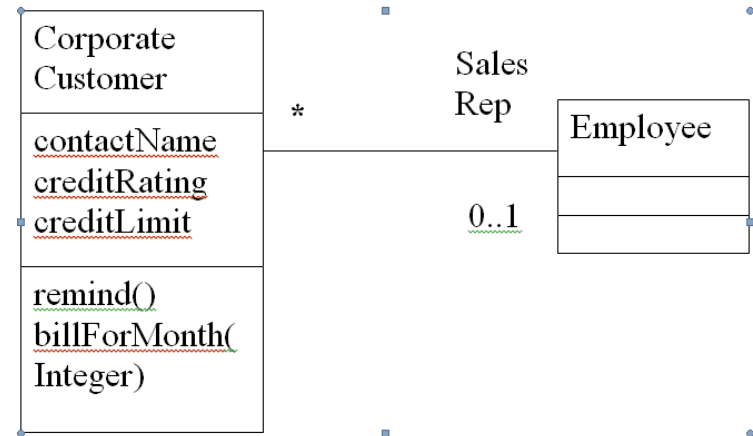
+	Public
-	Private
#	Protected
/	Derived (can be combined with one of the others)
~	Package

# An example: UML

```
public abstract class Person {  
    protected String personName;  
    private int age;  
    // constructor  
    public Person(String name) {  
        personName = name;  
    }  
    public static String makeJob() {  
        return "hired";  
    }  
    public int getAge() {  
        return age;  
    }  
    private void splitName() {}  
    abstract String getJob();  
}
```

Person
-age: int
#personName: String
+Person(java.lang.String)
+getAge(): int
getJob(): String
+ <u>makeJob(): String</u>
-splitName()

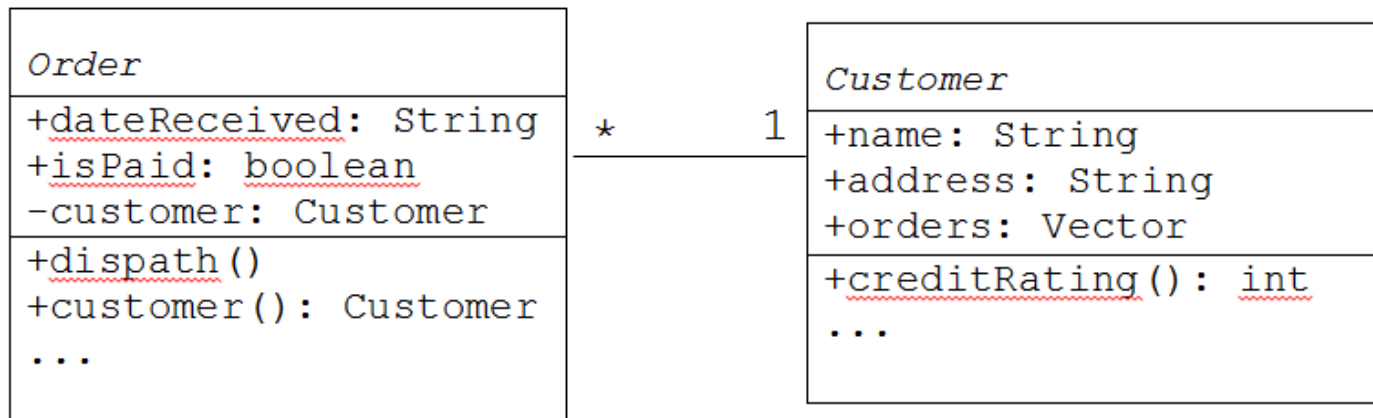
# UML – class associations



- **roles** i.e. what is the name of the role that an object of class B plays in its association with an object of class A.
- **multiplicity**, i.e. how many objects of class B can be in that relationship with each object of class A.
- A line between class icons represents association, the extra information is optional. Eg,
- A corporate customer object may have an employee object as its sales rep. It may have no rep or one rep. An employee object may be the sales rep of several different corporate customer objects.

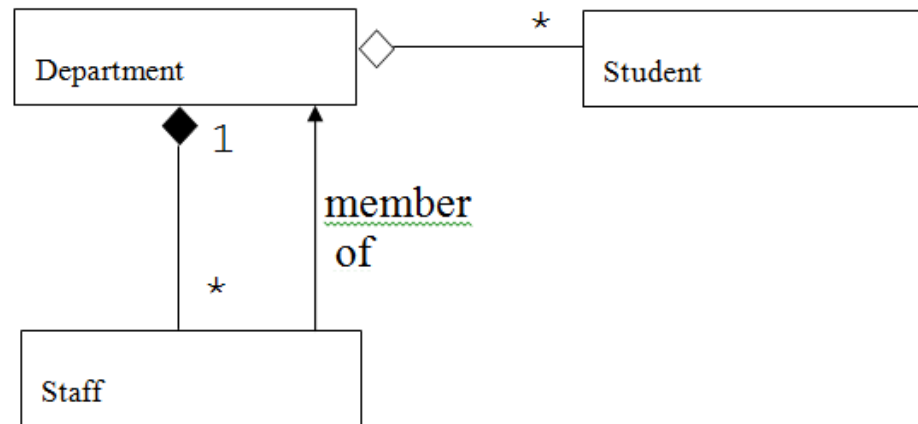
# UML – class associations

- Associations represent relationships between instances of classes, e.g. consider the following relationship: a Customer can make one or more orders but an Order is associated with only one Customer:

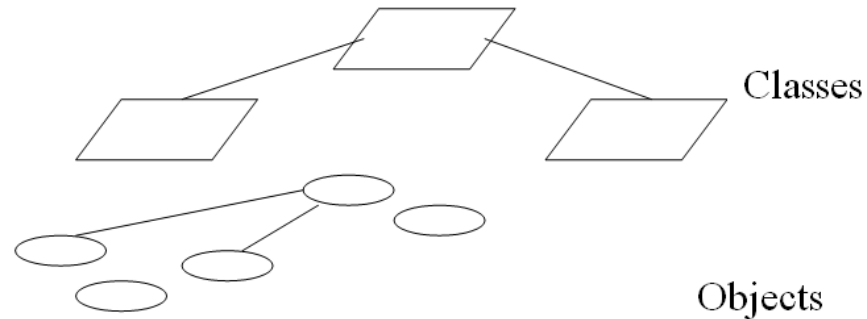


# Aggregation

- Aggregation is a special form of association. It represents a *has-a* or *part-of* relationship.
- A stronger form of aggregation is called *composition*. Example: a Faculty member belongs to one and only one department; a department has many students but students do not necessarily belong to just one department:



# The Sea of Objects



- It is useful to think of the class diagram as floating over a sea consisting of the objects which exist when a program is running.
- Each object belongs to a class. A class may be represented at any particular moment by one, none or many objects.
- Each object may hold a reference to another object of some class.
- There should be an association between the two classes.

# OO Design Methods

- The basic steps in OOSE are:
  - **analysis** = identify the classes, their relationships and expected dynamic behaviour.
  - **design** = refine the results sufficiently to allow ...
  - **implementation** = implement the classes
  - (Iteration and incremental development as usual.)
- Note that the distinction between analysis and design is blurred. Both are really about finding a bunch of classes and being precise in specifying their:
  - methods and dynamic behaviour, and
  - the relationships between them.

# Finding Classes

- List all the **abstractions** in the problem domain and any that you think may be needed in the solution, i.e. list all things, entities, roles, data structures.
- For each class, identify the knowledge and abilities each object will need in order to carry out its responsibilities and provide its services. Determine the **data members and public methods**.
- Find the super and sub classes of a class (if applicable)
- Go through all the **scenarios**, i.e. the sequences of steps which the program goes through in order to get a particular task done. Check that each object of each class has the right knowledge and abilities. (**Use role playing.**)



# Finding Classes

- Determine **collaborators**, i.e. which other classes must a particular class communicate with. We then say that there is an **association** between the two classes.
- Record the data members for each class.
- Record the overall arrangements between classes on a **class diagram**. E.g. use UML.
- Check, simplify, increment, iterate.

# Design to Implementation

- Once we have a sufficiently detailed UML class diagrams for a problem, then the step from design to implementation becomes relatively straightforward.
- For each class, we already have a list of the required member fields and public methods. Direct translation of the details in the class icons (into, e.g., Java) can give a skeleton of the class implementations.
- The remaining task will be to provide the bodies of the methods and a few extra methods such as constructors and any useful private methods.
- The result needs to be carefully tested through the various scenarios.



**Murdoch**  
UNIVERSITY

# Summary



# Summary: Objects and Java

- This topic introduced the program structures and object model used in Java.
- A number of different diagram notations have been used in past years. We introduced the Unified Method Language (UML) to illustrate concepts from object-oriented design and the structures in Java programs.



**Murdoch**  
UNIVERSITY

