UCL DEPARTMENT OF ELECTRONIC & ELECTRICAL ENGINEERING

ELEC0021 - PROGRAMMING II
OBJECT-ORIENTED PROGRAMMING

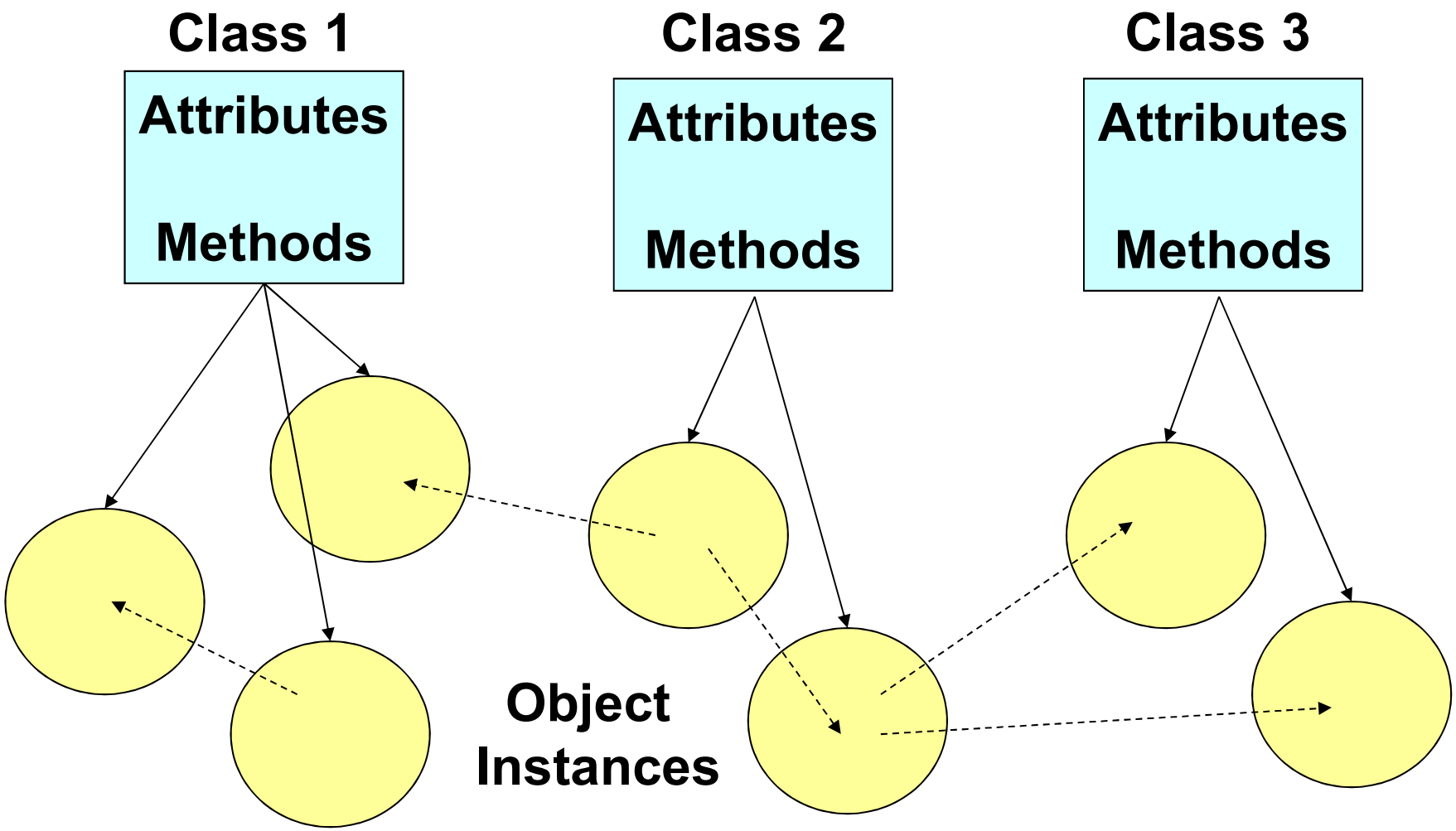
Classes and Objects

Prof. George Pavlou
Communication and Information Systems Group
Dept. of Electronic and Electrical Engineering
http://www.ee.ucl.ac.uk/~gpavlou/
g.pavlou@ucl.ac.uk

Classes

- Classes in Java and in any O-O language are collections of (public) methods and (private) data
- Methods define the operations that code of other classes can perform to an object instance
  - The simpler term object is short for object instance
- A class also defines data or object instance variables (or attributes) that all its methods have access to
  - Different copies of this data exist for every object instance
- A constructor is a special method with the same name as the class and typically initialises instance variables
  - It is not called directly but through the **new** operator every time some code creates a new object of a class

Object-Oriented Program

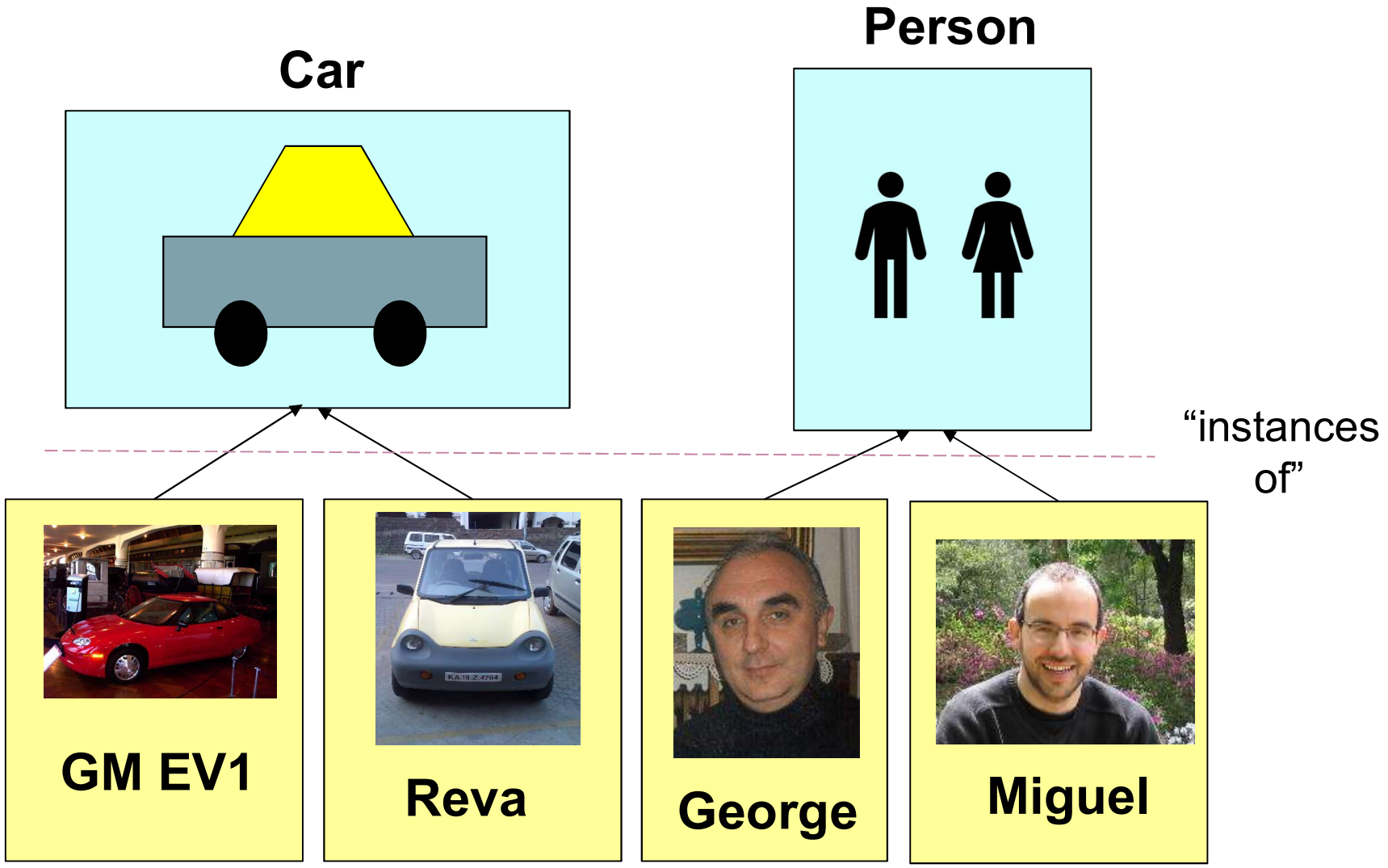


Static Methods

- Static methods are attached to a class and do not operate on any of the instance variables; also static instance variables exist once only for the class and not per object instance
- They can be called without an object instance created first
  - We simple prepend the method with the class name
    e.g. double d = Math.sqrt(900); int i = Integer.parseInt("10");
  - The same applies to static public instance variables
    e.g. System.out.printf("Hello world!\n"); // out is such a variable
- Every Java program has by convention to be a class that includes the static "main" method
  - The filename should be the same as the "program class name" otherwise the compiler will complain

    e.g. MeanAndVariance.java for a mean & variance program, DiceThrower.java for a dice thrower program
- The static main method is where the program starts & ends
  - public static void main (String[] args)

Defining Simple Classes



Defining Simple Classes (cont'd)

// Note: incomplete classes, constructors not fully coded

```
class Car
{
   String make;
   String model;
   int      engSize;

   Car (String mk, String mdl, int engSz) {
      // initialisation code goes here
   }
   String toString () {
      return "Car " + make + " " + model +
                 " has engine " + engSize;
   }
}
   …
   Car c1 = new Car("VW", "Golf GTI", 2000);
   System.out.printf("%s\n", c1.toString());
   …
```

```
class Person
{
   String name;
   String surname;
   int      age;

   Person (…) {
      // also here
   }
   String toString () {
      return "Person " + name + " " +
                 surname + " is " + age +
                       "  years old";
   }
}
 // we create a Car object
 // and print out its content
```

Constructors

```
// note “overloaded” constructors:
// two constructors with different parameters
public class Point
{
    private int x, y;       // private instance variables - public methods

    public Point () {
        x = 0;  y = 0;
    }

    pubic Point (int xarg, int yarg) {
        x = xarg;  y = yarg;
    }

    public String toString () {
        return “[“ + x + “, ” + y + “]”;
    }
    // class incomplete: getX, getY, setX, setY methods required
}
```

Method Overloading

- A class can include methods with the same name but different parameters, this is called method overloading
  - The compiler distinguishes overloaded methods through their signature, which is a combination of the method's name and the number, type and order of parameters
  - Two methods with the same name and parameters but with a different return type are not allowed!
- Constructors can also be overloaded
- Function overloading is NOT possible in C because function signatures do not include the parameter types
  - Method overloading is possible in C++ – as in Java

Method Overloading (cont'd)

```
public class MathCalc   // class with only static methods -
{                       // similar to a (thematic) collection of C functions

   // square method with int argument
   public static long square (int intValue)
   {
      System.out.printf( "Called square with int argument: %d\n", intValue );
      return intValue * intValue;
   } // end method square with int argument

   // square method with double argument
   public static double square (double doubleValue)
   {
      System.out.printf( "Called square with double argument: %f\n", doubleValue );
      return doubleValue * doubleValue;
   } // end method square with double argument

} // end of Class
```

An Example Simple Program

```
import java.util.*;    // necessary for the Scanner class

public class MathCalc {
   // here go the two square methods of the previous slide …

   public static void main (String[] args) {

      // create a Scanner object to scan the standard input
      Scanner input = new Scanner(System.in);

      // get first the integer and then the double argument
      int       intArg  =  input.nextInt();
      double dblArg =  input.nextDouble();

      // print out each of the arguments and its square
      System.out.printf("square of %d is %d\n", intArg, MathCalc.square(intArg));
      System.out.printf("square of %f  is %f\n", dblArg, MathCalc.square(dblArg));
   } // end main method

} // end class MathCalc
```

Block Statements and Scope of Variables

- Most programming languages use block statements, also known as “code blocks”, delimited by { and }
  - A *class* is a block, a *method* is a block, a *for loop* is a block
  - Primitive types and object references declared in a code block are added onto the program “stack” (see later) and are removed (emptied) when the block ends, i.e. disappear from scope
- The scope of a variable is the program part that can access that variable. Scope rules are as follows.
  - The scope of a local variable in a method is from the declaration point until the end of that block, i.e. the closing “}”
  - The scope of a for statement initialisation variable is the *for* body
  - The scope of a method parameter is the entire method
  - The scope of an instance variable is the entire class
- For variables of the above types *with the same name* (which is bad practice!) the “innermost” prevails

Scope of Variables Example

```
class Point {
    int x, y;

    pubic Point (int x, int y)  {    // not good practice, same names x, y!!
        // here the x, y parameters "overwrite" instance variables x, y
        this.x = x;  this.y = y;      // we need to use "this", see later
    }

    public void scopeTest (int x) { // deliberately bad parameter name
        int x = 10;                          // deliberately bad local variable name
        // the following x will be 10, overwriting parm x and inst variable x
        System.out.println ("x is " + x);

        for (int x = 0; x < 2; x++)     // deliberately bad for loop parm name
            // the following x will be 0 and 1 in two iterations,
            // overwriting local variable x=10, parm x and instance variable x
            System.out.println ("x is " + x);
    }
}
```

Primitive and Reference Types

- All method parameters and also instance variables must have a type
  - There exist primitive types and reference types
- Java supports the following 8 primitive types:
  boolean, char, byte, short, int, long, float, double
  - Primitive types have well defined standard size (between 1-8 bytes) and can be stored in a fixed amount of memory
  - All primitive types declared are stored in the program "stack"
- Objects of any type (and Arrays, which also are objects) can have any size in memory and are accessed by references – hence reference types
  - The space for any object created by **new** is in the program "heap"

Program Stack and Heap

- When a program is executed, it is given an initial amount of memory by the operating system
  - This can be increased later if the program needs more memory
- A part of this memory is the program "stack": method parameters and also primitive types & references declared in methods go onto the stack; its top-most part is emptied when they go out of scope i.e. at the end of every code block { }
  - The first thing to go onto the stack is the *args* parameter of *main*
  - A stack is a *last-in-first-out* structure, just like a "stack of dishes"
- Another part of this memory is the program "heap", where object instances created using the **new** operator are placed
  - This is *dynamic* memory as opposed to the *static* memory on the stack, similar to memory allocated using *malloc* in C

Reference Types

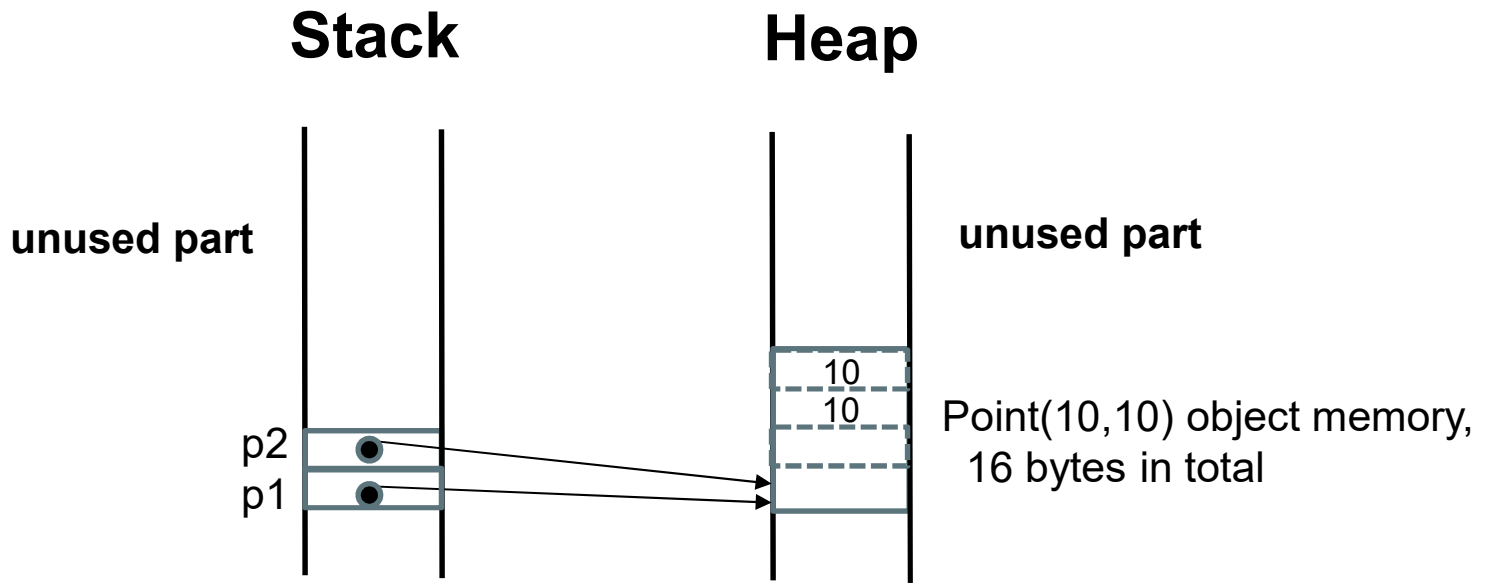
- When declaring an object reference, it goes onto the program “stack”, exactly as the primitive types

```
Point p1 = null;  // p1 is a reference to a Point object (not created yet)
```

- New object instances go in the program “heap”

```
p1 = new Point (10, 10);  // we create new Point object in the heap
                          // pointed to by reference p1 in the stack
Point p2 = p1;   // new reference p2 pointing to the same object above
// the actual references p1 & p2 are in the program stack
// and will vanish at the end of the code block
```

Reference Types (cont'd)



- References p1 and p2 are memory addresses pointing to the same location in the heap, i.e. to object Point(10,10)
- This is a 32-bit machine architecture, with 4-byte "words"; every object has 8 bytes of meta-data, so a Point is 16 bytes long in memory given it has 2 *int* instance variables of 4 bytes each

Wrappers for Primitive Types

- Java provides “wrapper classes” that provide object encapsulations for all the primitive types so that they can be stored in collection classes such as list etc.
  - These wrapper classes start with an upper case letter in relation to the primitive types
  - **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double** (note that it is Integer/Character, NOT Int/Char)
  - A key method of Integer is “static int parseInt(String s)” which converts a numerical string to an int e.g. “10” to 10
- For example, a *generic* linked list can store objects, so it can store instances of Integer, String or whatever
  - But cannot store “int” because this is primitive type, not an object, unless we make a *specific* list that stores only int

Wrapper Type Examples & Stack/Heap

```
Integer j0 = null; // a reference on the stack to an Integer object, not yet created
{
    int i1 = 10;  // int variable i1 on the stack, initialised to 10

    Integer j1 = new Integer(i1);  // new Integer object on the heap, contains value 10

    j0 = j1; // assign Integer object reference j1 to j0 declared above, outside the block

    int i2 = 20; // int variable i2 on the stack, initialised to 20

    int i3 = j1.intValue()+i2; // int variable i3, gets 10 from j1 + 20 from i2 = 30

    Integer j2 = new Integer(40);  // new Integer object on the heap, contains value 40
}

// i1, j1, i2, i3 and j2 disappear from scope at end of block, i.e. stack is emptied
// we still have access to object j1 via reference j0 but not to object j2,
// this is still somewhere in the heap but we did not copy its reference

System.out.printf("j0 is %d\n", j0.intValue());  // this will print “j0 is 10”
```

C Pointers

- In C we can declare pointers to any struct type, and also to primitive data types such as int and char
- We can also perform pointer arithmetic and “walk” through the underlying computer’s memory
- The following is an example of C pointer usage:

```
int   i = 2;
int   j = 10;
int* pi = &i; // this points to the stack memory position of int i
pi++;         // it now moves to the next int location on the stack,
              // so it now points to the stack memory position of j
```

- This type of low-level memory manipulation through pointers is NOT possible in Java

Java Object References

- The notion of an equivalent pointer in Java is an object reference
- It points to the piece of memory where the object is stored in the program heap. But:
  - We can not do pointer arithmetic in Java and walk through a piece of memory starting from a reference, as we can do with any pointer in C
  - References are strongly-typed, for example we cannot cast a reference to Integer to a reference to String while in C we can cast an int* to a char* or to anything else we like
  - A reference is effectively a *strongly-typed immutable pointer*
- C pointers are more powerful than Java references but also more dangerous as it is pretty easy to use them incorrectly!

Strings

- Strings in Java are supported by the String class and, as such, are of reference type
  - This is in contrast to C where a string is simply a pointer to a series of char terminated by '\0', i.e. char*
  - Also in contrast to C++ which supports both C-style strings and a standard library String class, but the latter is not a "first class type" as in Java
- Manipulating strings is a very important feature of many programs
  - The existence of the String class makes it quite easy
- It is worth looking at the methods the String class as they can be useful for many things

The Java Class Library

- The Java programming language contains a basic set of programming functionality
- The **Java Class Library** is distributed with the compiler and contains more than 3200 classes
- You can see the list at: http://java.sun.com/j2se/1.5.0/docs/api/index.html
- A good programmer is the one who manages to program less and **use** more!
  - Use existing library classes when you can

Some Packages in the Java Class Library

- **java.util** – various utility classes
- **java.io –** input/output (i/o): streams and files
- **java.net** – classes for networking applications
- **java.awt** – for creating graphics and images
- **javax.swing –** for user interfaces
- **java.bean –** for developing components (i.e. beans)

Commenting your Code

- It is very important to comment your code! For your colleagues, for the person who will mark your exam paper or assignment and for yourself in the future
- It is good practice to put the comments first and then the code itself
- Java supports both single line (C++ style - // …) and multiple line (C style - /* … */) comments
  - You should mostly use the single line style // …
  - The multiple line style /* … */ should be only used to comment out whole blocks of code when debugging

Code Layout: Indentation

- The code within a code block delimited by { and } should be indented to the right by a fixed amount of white space (typically 4 spaces)
- For example in a file with the code of a class:
  - Instance variable and methods should all be 4 spaces right
  - The code within every method should be another 4 spaces to the right, starting at “column” 9
  - The code within a for loop in that method, i.e. another block, should be another 4 spaces right, starting at “column” 13 etc.
- See also http://en.wikipedia.org/wiki/Indent_style
- Badly indented code will be penalised!

Example of Correct Indentation

```
// class starts at column 1
public class List
{
   // instance variables and methods start at column 5
   // instance variables …

   public void insertAtFront (Object newData)
   {
      // if else statement at first level within method starts at column 9
         // the if and else internal blocks start at column 13
      if (firstNode == null)  // empty list
         firstNode = lastNode = new ListNode(newData, null);
      else
      {
         ListNode newFirstNode = new ListNode(newData, firstNode);
         firstNode = newFirstNode;
      } // end else
   } // end insertAtFront method

   // more methods starting at column 5 …

} // end class
```