GC01 Introductory Programming

Week 2 – Lecture

Dr Ghita Kouadri Mostefaoui
Department of Computer Science

Basic Object-Oriented concepts

Programming paradigms

- Different methodologies are more suitable for solving certain kinds of problems or applications domains.
- Same for programming languages and paradigms.
- Programming paradigms differ in:
  - the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints, etc.)
  - the steps that compose a computation (assignation, evaluation, data flow, control flow, etc.).

Programming paradigms

- In **object-oriented programming**, programmers can think of a program as a collection of interacting objects, while in **functional programming** a program can be thought of as a sequence of stateless function evaluations.

- In **process-oriented programming,** programmers think about applications as sets of concurrent processes acting upon shared data structures.

Programming paradigms

- Some languages are designed to support one particular paradigm
  - Smalltalk supports object-oriented programming
  - Haskell supports functional programming

- Other programming languages support multiple paradigms
  - Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Perl, Python, Ruby, Oz and F#.

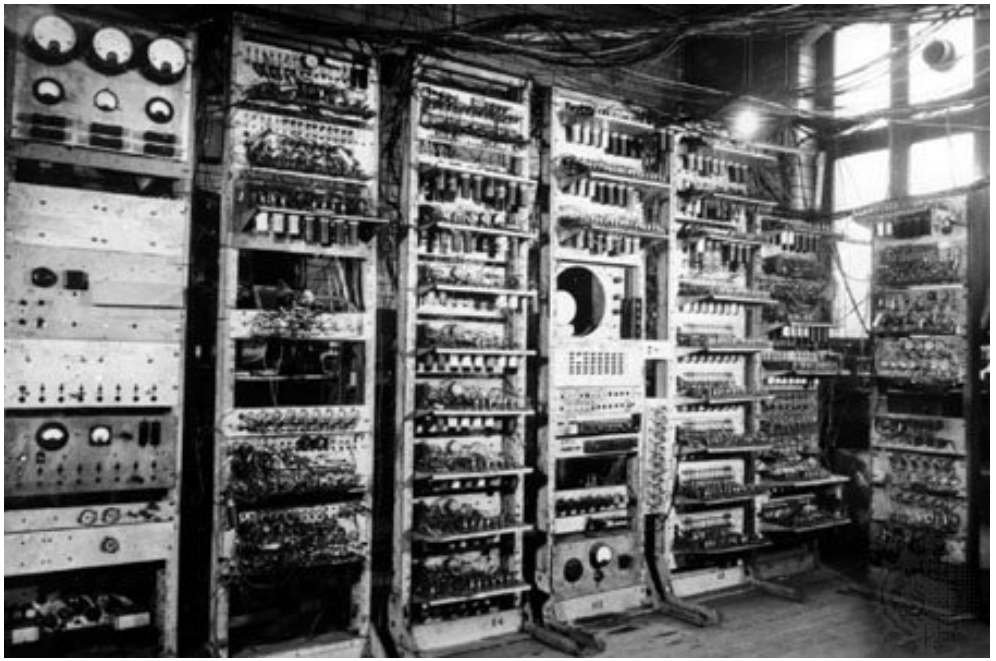
- The design goal of multi-paradigm languages is to allow programmers to use the best tool for a job, admitting that no one paradigm solves all problems in the easiest or most efficient way.

Evolution of programming languages

Low level - Genesis

- Initially, computers were **hard-wired** or **soft-wired** and then later programmed using **binary code** that represented control sequences fed to the computer CPU.

- This was difficult and error-prone. Programs written in binary are said to be written in machine code, which is a very low-level programming paradigm. Hard-wired, soft-wired, and binary programming are considered **first generation** languages.



```
01010101 01101110 01100100 01100101 01110010 01110011 01110100 01100001
01101110 01100100 01101001 01101110 01100111 00100000 01110100 01101000
01100101 00100000 01100010 01101001 01101110 01100001 01110010 01111001
00100000 01101110 01110101 01101101 01100010 01100101 01110010 00100000
01110011 01111001 01110011 01110100 01100101 01101101 00100000 01101001
01110011 00100000 01100001 00100000 01110101 01110011 01100101 01100110
01110101 01101100 00100000 01110011 01101011 01101001 01101100 01101100
```

Low level – Assembly language

- To make programming easier, assembly languages were developed.
- These replaced machine code functions with **mnemonics** and memory addresses with **symbolic labels**.
- Assembly language programming is considered a low-level paradigm although it is a '**second generation**' paradigm.
- Assembly languages of the 1960s eventually supported **libraries** and quite sophisticated conditional **macro** generation.

```
fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

    @@:
    push ebx
    mov ebx, 1
    mov ecx, 1

    @@:
        lea eax, [ebx+ecx]
        cmp edx, 3
        jbe @f
        mov ebx, ecx
        mov ecx, eax
        dec edx
    jmp @b

    @@:
    pop ebx
    ret
```

Assembly language still relevant

- Assembly was, and still is, used for **time-critical** systems and frequently in **embedded systems**.
- Assembly programming can directly take advantage of a specific computer architecture and, when written properly, leads to **highly optimized code**.
- However, it is bound to this architecture or processor and thus suffers from **lack of portability**.
- Assembly languages have **limited abstraction capabilities**, which makes them unsuitable to develop large/complex software.

High level

- A programming language that enables a programmer to write programs that are more or less independent of a particular type of computer.
- Such languages are considered high-level because they are **closer to human languages** and further from machine languages. Easier to read, write, and maintain.
- Must be translated into machine language by a compiler or interpreter.
- Ada, Algol, BASIC, COBOL, C, C++, FORTRAN, LISP, Pascal, and Java.

```
unsigned int fib(unsigned int n)
{
    if (n <= 0)
        return 0;
    else if (n <= 2)
        return 1;
    else {
        int a,b,c;
        a = 1;
        b = 1;
        while (1) {
            c = a + b;
            if (n <= 3) return c;
            a = b;
            b = c;
            n--;
        }
    }
}
```

Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm that uses "**objects**" – data structures encapsulating data fields and procedures together with their interactions – to design applications and computer programs.

- Four fundamental OOP concepts: **abstraction**, **encapsulation**, **polymorphism**, and **inheritance**.

- Though it was invented with the creation of the **Simula** language in 1965, and further developed in **Smalltalk** in the 1970s, it was not commonly used in mainstream software application development until the early 1990s.

- Many modern programming languages now support **OOP**.

Why object-oriented design?

- Software is complex (too many people is doing too many things – the mess is inevitable ☺)
- One of the main goals in programming is to avoid the redundancy and objects can help to do this (**inheritance**)
- Objects can help increase modularity through data hiding (**encapsulation** or **data hiding**)

Class and Object

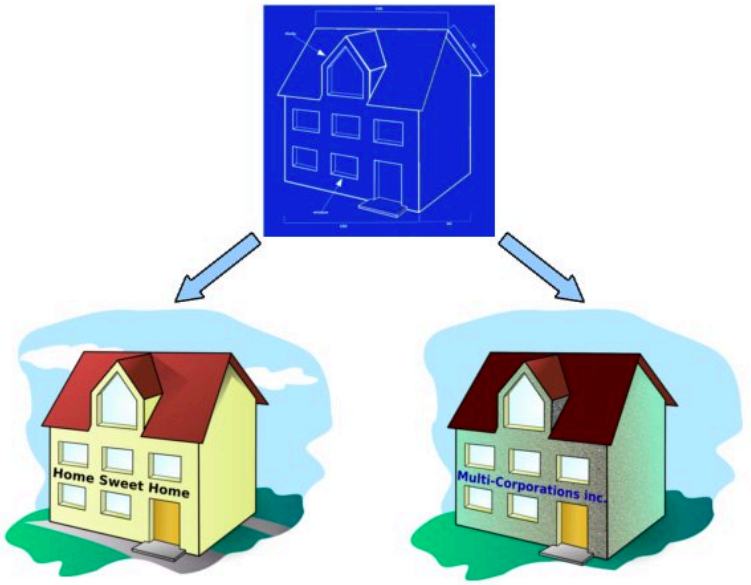
- "Class" refers to a blueprint. It defines the *variables* and *methods* the objects support.

- "Object" is an *instance* of a class. Each object has a class which defines its *data* and *behavior*.

- *Explanation…*

Class and Object

- *A class is to an Object what a blueprint is to a house*
- A class establishes the characteristics and the behaviors of the object
- *Classes* are the plan; *objects* are the embodiment of that plan
- Many houses can be built from the same blueprint



Anatomy of a class

In the Java language, the general form of a class definition is:

```
accessModifier class Classname{
    // field, constructor, and
    // method declarations
}
```

Class body

- The keyword **class** begins the class definition for a class named ***Classname***.
- *Access modifiers* such as ***public***, ***private***, and a number of others that you will encounter later.
- The *className*, with the initial letter capitalized by convention.
- The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block.
- Anything included between the two curly brackets is called the *class body*

Class example: Minimal form

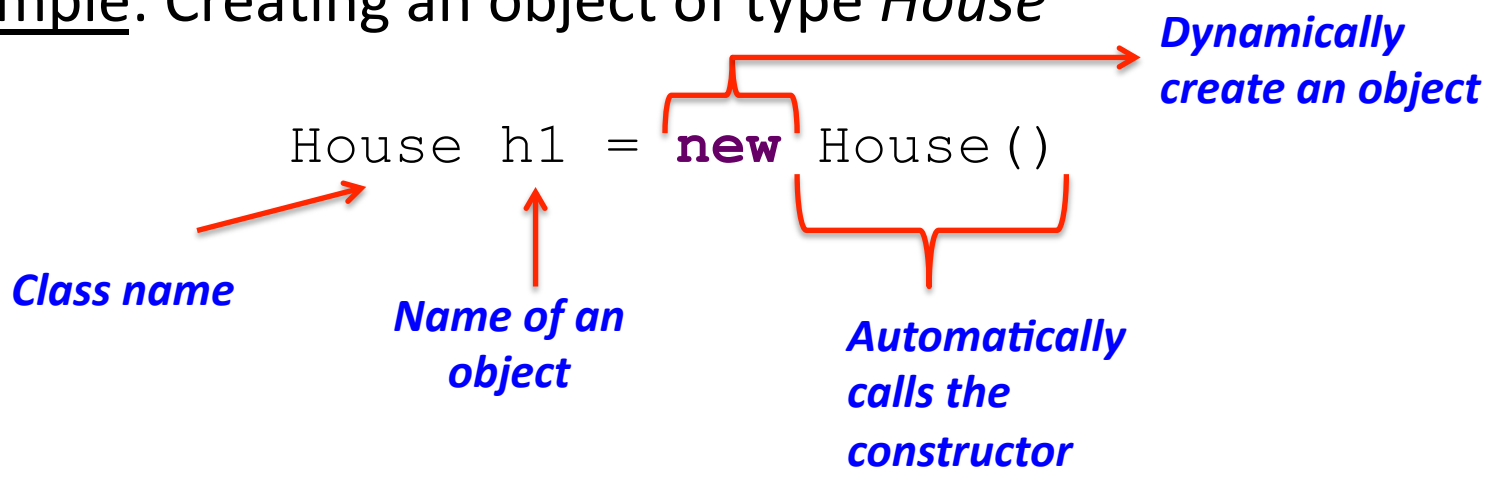
```
public class House{

}
```

instantiating a class = Creating an object

- The **`new`** operator instantiates a class by allocating memory for a new object and returning a reference to that memory.

- ***Note***: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

- Example: Creating an object of type *House*



instantiating a class = Creating an object

```
House h1 = new House()
```

- Memory is allocated for an object after executing this statement.
- This Statement will create instance of the class *House* and name of instance is nothing but actual object *h1*.

Driver class or Main class

```
/* This is a driver class also known as
main class or entry point */
public class HouseApp {

public static void main(String[] args) {

     House h1 = new House();
   }
}
```

Class and Object

- “Class” refers to a blueprint. It defines the *variables* and *methods* the objects support.

- “Object” is an *instance* of a class. Each object has a class which defines its *data* and behavior.

Anatomy of a class: Instance variables

```
public class Classname{

    type instance-variable1;
    type instance-variable2;

    // ..

    type instance-variableN;
}
```

instance-variable1, *instance-variable2*, etc… are called *instance variables*, *fields* or *member variables* (more general definition).

Anatomy of a class: Instance variables

```
public class House{

// These are all instance variables
   int propertyNumber;
   int nbBedrooms;
   boolean bathroomIsClean;
}
```

- `int, boolean`, etc… are the variables types.
- A variable can be of a primitive type or reference/object data Type.

Driver class

```
/* This is a driver class also known as main class
or entry point */
public class HouseApp {
   public static void main(String[] args) {

      House h1 = new House();

      System.out.println("House number: " +
h1.propertyNumber + " has " + h1.nbBedrooms + "
bedrooms. The bathroom is  clean? " +
h1.bathroomIsClean);
   }
}
```

Initialisation of instance variables (1)

- You can often provide an initial value for an instance variable in its declaration.

```
public class House{
// These are all instance variables
   int propertyNumber;
   int nbBedrooms = 5;
   boolean bathroomIsClean;
}
```

Driver class

```
/* This is a driver class also known as main class
or entry point */
public class HouseApp {
   public static void main(String[] args) {

      House h1 = new House();

      System.out.println("House number: " +
h1.propertyNumber + " has " + h1.nbBedrooms + "
bedrooms. The bathroom is  clean? " +
h1.bathroomIsClean);
   }
}
```

Initialisation of instance variables (2)

- You can also provide an initial value for an instance variable in the newly created object (instance).
- We use the *dot notation* and the *assignment operator.*

```
public class HouseApp{
…
House h1 = new House();
      /* accessing h1's copy of instance variable
         using the dot notation */
      h1.nbBedrooms = 7; // assignment
…
}
```

Driver class

```
/* This is a driver class also known as main class
or entry point */
public class HouseApp {
   public static void main(String[] args) {
      House h1 = new House();
      /* accessing h1's copy of instance variable
         using the dot notation */
      h1.nbBedrooms = 7; // assignment
      h1.propertyNumber = 25;
      h1.bathroomIsClean = true;

      System.out.println("House number: " +
h1.propertyNumber + " has " + h1.nbBedrooms + "
bedrooms. The bathroom is  clean? " +
h1.bathroomIsClean);
   }
}
```

Class and Object

- “Class” refers to a blueprint. It defines the variables and *methods* the objects support.

- “Object” is an *instance* of a class. Each object has a class which defines its data and *behavior.*

Methods

- A class with only instance variables has no life. Objects created by such a class cannot *respond to any messages.*

- Methods are declared inside the body of the class but immediately *after the declaration of instance variables.*

Methods: General form

The general form of a method declaration is:

```
modifier returnType nameOfMethod (Parameter List) {
 // Method-body
}
```

- *modifier*: It defines the access type of the method and it is optional to use.
- *returnType*: Method may return a value.
- *nameOfMethod:* Choose meaningful names.
- *Parameter List*: The list of parameters, it is the type, order, and number of parameters of a method. A method may contain zero parameters.
- *method body*: The method body defines what the method does.

Example

```
public class House{
    int propertyNumber;
    int nbBedrooms = 5;
    boolean bathroomIsClean;

    // return the house type
    public String getOfficeType(){
        if (propertyNumber == 11){
            return "Police Station";
        }else if ( propertyNumber == 20){
            return "Community center";
        }else {
            return "Family house";
       }
    }
}
```

Note: The use of the **return** keyword.

Method calling

```
public class HouseApp {
    public static void main(String[] args) {
        House h1 = new House();
        h1.propertyNumber = 23344;

        // method calling
        String type = h1.getHouseType();

        System.out.println("h1 is of type: " + type);

    }
}
```

The **`void`** keyword

- The void keyword allows us to create methods which *do not return a value*.

- Example:

```
public void getInfo(){
    System.out.println("Any question please call: 890
                        980 980");
}
```

- Call to a void method must be a statement i.e.

```
h1.getInfo();
```

Class constructors

- A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

- Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to default values. However, once you define your own constructor, the default constructor is no longer used.

Class constructors

```
// constructor
public House(){
        propertyNumber = 12;//see slide on 'this' keyword
        bathroomIsClean = true;

        System.out.println("Constructor called.")

    }
```

Parameterised constructors

- Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

```
// parameterised constructor
public House(int c){
        propertyNumber = c;
        System.out.println("Constructor called." )
    }
```

The `this` keyword

```
// parameterised constructor
public House(int propertyNumber, boolean
                    bathroomClean){

this.propertyNumber = propertyNumber;
this.bathroomClean = bathroomClean;
      }
```

- `this` is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor.
- The keyword `this` is used to differentiate the instance variables from local variables if they have same names, within a constructor or a method.

Multiple constructors

- A class can have multiple constructors.
- Each one must accept a unique set of parameters.
- (See previous slides)

Common constructor bugs

- Accidentally writing a return type such as `void`:

```
public void House(int number) {
    this.propertyNumber = number;
}
```

  - This is not a constructor at all, but a method!

  - Storing into local variables instead of fields ("shadowing"):

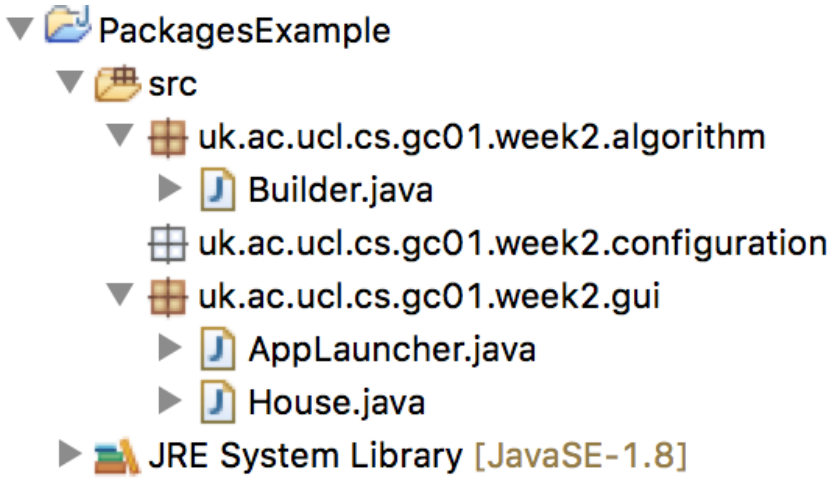
```
public House (int number) {
    int propertyName = number;
}
```

  - This declares a local variable with the same name as the fields, rather than storing values into the fields. The fields remain with their default values.

Java packages

- Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to.

- A Java package is a Java programming language mechanism for organizing classes into namespaces.



Package Naming Conventions

- Package names should be all lowercase characters whenever possible.
- Frequently a package name begins with the top level domain name of the organization and then the organization's domain and then any subdomains listed in *reverse order*.
- The developer can then choose a specific name for their package..

```
uk.ac.ucl.cs.gc01
```

Using packages

- If class is contained in a given package, at the top of its source file (before any imports or anything else other than comments), you should have a package declaration.

```
package uk.ac.ucl.cs.gc01.week2.gui;
```

- To use a package inside a Java source file, it is convenient to import the classes from the package with an import statement.

```
import uk.ac.ucl.cs.gc01.week2.algorithm.*;
```

Controlling access to members of a class

An access modifier determines whether other classes can use a particular *member variable* or call a particular *method.*

- **public**
  - Accessible to everyone, both designer and users of the class
- **private**
  - Accessible only within class implementation, only class designer
- **protected**
  - Accessible to the package and all subclasses (classes that are designed based on this one). We will cover it when we study inheritance.
- default (when you list no keyword)
  - Accessible to the package.

Java access modifiers: Summary

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N