Object-Oriented Programming

Encapsulation

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September 18, 2018

Table of Contents I

Table of Contents II

Learning outcomes

After this lecture and the practical students should...

- understand the meaning of access level modifiers
- be familiar with the 4 access specifiers and their effect
- understand the concept of encapsulation
- understand the use of getter and setter methods
- to be able to implement getter and setter methods
- be able to identify the public interface of a class
- be able to tell the difference between a static variable and an instance variable, and understand how the use of either effects a class
- understand how to define and call static methods

Visibility

- When working in teams we want to make others use our classes the way we want
- We use access level modifiers for this

Access levels

- `public`
- `protected`
- `default`
- `private`

- We usually call this as visibility

Access level modifiers

- Can be applied to classes, methods, instance variables, constructors and more
- They define where code can be access from

Syntax

```
public int day;
protected int year;
int month;
private int hour;
```

- Note that `default` is defined by no keyword

Access levels

This table shows where each access level allows access

Modifier	The same class	The same package	A subclass	everywhere else
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Public

When something in a class is declared `public`, this means that it can be accessed by any code

Example

- A Ship object with public instance variables, can be changed changed
- I can set the x coordinate of the player to -12312 or his weapon cooldown to 9000
- A ship should not have a value for health below 0
- So we should probably not declare these as public

Protected

- When something is declared as protected, this means that it can be accessed from within the same class, package or by a subclass of this class

Default

- When no access specifier is defined, this means that it can be accessed from within the same class or package

- Useful with a good package structure

Private

- Private is the most restrictive access level
- Can only be accessed, changed or used from within the same file
- If we use private instance variables in the Ship class, then other programmer can only access or change the values by calling methods
- We control the methods

Choosing Access Levels

- There are 4 levels of access that we can allow for every instance variable, class or method.
- But how do we know which level should be used?
- It comes down to a single question, who needs to see or change this?

Access Level for Classes

- For classes, we will mostly declare them as public
- When we define a class it is supposed to be used by other classes in different parts of the program

Access Level for Methods

- Methods are often declared with different access levels
- If we define a method that we expect others to use then it will be declared public
- If we define a method that we do not want others to use, but we would like to use it in other parts of our package, then we declare it default
- If we write a method that is only use from within our class then it should be declared as private

Access Level for Instance Variables

- When declaring instance variables of a class, we think backwards
- Most instance variables should be declared as private
- Only where an instance variable **needs** to be used by another class would we use any other access level
- This idea is called information hiding
- This means that we keep control over **how** they can change
- Information hiding is part of an idea know as encapsulation

Encapsulation

- Encapsulation in programming languages is the combination of two ideas,
  - The grouping together of instance variable and methods into classes
  - The ability to restrict access to some of the objects components

Information hiding

- The main purpose behind information hiding is the idea of a black box

Black box

A black box is a device, system or object which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings. Almost anything might be referred to as a black box: a transistor, algorithm, or the human brain

Information hiding

Black Box Example

A very simple real world example of this is a car. I know the idea of how a car works, I can provide the input to make the car move, stop, turn, etc..
But when I press the accelerator, I do not know how the engine performs the task of moving

Black Box in Programming

- A game is being developed by a large group
- Group A is responsible for the code defining a Ship
- Group B is responsible for the overall game logic
- Group B does not need to know how the location of the ship is stored, just what methods they can call to access or change it
- We can change instance variables in the class as long as the methods work
- When `getX` is called, the correct value will still be supplied

Bullet Class

```
1  public class Bullet {
2    private int x, y;
3    private boolean up; // true for up, false for down
4    private int damage;
5    public Bullet(int x1, int y1, boolean u, Ship o) {
6      x = x1;
7      y = y1;
8      up = u;
9      origin = o;
10   }
11   public void printBullet() {
12     ...
13   }
14 }
```

Effect of Changes

- Other classes can no longer access the values of our instance variables
- This means that the `isHit` method in the ship class will not work
- So how can we let other parts of the program know the values without allowing them to change?
- By adding new methods to the class

Getter Methods

- A getter method is a method that tells us the value of an instance variable
- `x` and `y` are not available from outside the bullet class, so we add a getter method to the class to return the value

```
1 public int getX() {
2   return x;
3 }
4 public int getY() {
5   return y;
6 }
```

- Values cannot be changed

Setter Methods

- A setter method can be used to change the value of an instance variable
- A setter method will usually take a parameter, and use this value to change the instance variable

Setter Method - move

```
1 public void move() {
2    if(up) {
3       y = y - 2;
4    } else {
5       y = y + 2;
6    }
7 }
```

- This code ensures changes the y coordinate in the correct direction

Using getters and setters

- The private instance variables in the Bullet class causes errors in the program
- Replace access with getter method
- A ship object followed by `.x` is replaced with `.getX()`
- If we changed the value `.x = expression;` then we would replace it with `.setX(expression);`.

Separating Ideas

- Through applying the ideas of encapsulation, we have essentially spilt our classes into two separate parts
- The first part is what we allow other programmers to see and use, this is called the public interface
- The second part is the actual way in which all the code is written, this is called the private implementation.

Public Interface

- The public interface of a class is the methods, constructors and instance variables that are public
- The public interface of the Bullet class is simple;
  - `public Bullet(int x, int y, boolean u, int d)`
  - `public int getX()`
  - `public int getY()`
  - `public void move()`
  - `public void printBullet()`
- These are the only methods that can be used to interact with a bullet object

Class Variables (Static Variables)

- Instance variables have a **separate** value for each object, class variables share a single value over all objects
- 5 objects based on the same class will always have the same value for a class variable
- Only created in memory once
- Class variables are usually called static variables

Declaring a Static Variable

- Syntax for a static variable is to add the keyword `static` before the type
- Any visibility
- Not very common

Example

```
1 public class  Counter {
2   private int count;
3   public void  increment(){
4     count++;
5   }
6   public int getCount (){
7     return count;
8   }
9 }
```

Here `count` is an instance variable, so each `Counter` object has its own value

Static Variables

Example with instance variable

```
1 Counter a = new Counter();
2 Counter b = new Counter();
3 a.increment();
4 a.increment();
5 System.out.println(b.getCount());
```

- Create two `Counter` objects
- Increment `a` twice
- Print out the value of `count` in `b`

What is the result?

Static Variables

Example

```
1 public class  Counter {
2   private static int count;
3   public void  increment(){
4     count++;
5   }
6   public int getCount (){
7     return count;
8   }
9 }
```

Here count is an static variable, so every Counter object shares the same value

Static Variables

Example with static variable

```
1 Counter a = new Counter();
2 Counter b = new Counter();
3 a.increment();
4 a.increment();
5 System.out.println(b.getCount());
```

If we preform the same steps as before:

- Create two `Counter` objects
- Increment `a` twice
- Print out the value of `count` in `b`

What is the result?

Using Class Variables

- When a class variable is public or default, it can be accessed directly from other classes without an object
- Because class variables are connected to the class, these variables are referenced using the class name
- We use the syntax `ClassName.variableName`
- For example, to access the count class variable (if it was public) we would use the syntax `Counter.count`.

Examples of Class Variables

- There are many examples of useful class variables that we can use when performing calculations
- For example, when performing calculations related to circles or spheres, we could use the value of the class variable `PI` from the `Math` class
  - e.g. `double area = Math.PI * radius * radius;`
- Additionally, `MAX_VALUE` and `MIN_VALUE` define the largest possible value and smallest possible value for many of the different primitive values
- These are stored in the classes Integer, Float, Double etc.

Final Variables

- Final variables can be assigned only once
- Once the value has been set, it will never change
- The syntax for declaring a final variable is to add the keyword `final` before the type of the variable
- Compiler error if we change it

Constant Variables

- Constant variables are variables that are final and static
- The have the attributes of both types of variables
- Usually used to store values that will not and should not change
- Often called **constants**

Constants

- A good example of useful constants, would be to represent values such as the screen bounds
- This could then be used in multiple locations
- Makes code consistent
- Change in only one place

Example

```
1 public void move(int xm, int ym) {
2    int x1 = this.x + xm;
3    int y1 = this.y + ym;
4    if(x1 > 0 && x1 + this.width <
     GameScreen.SCREEN_WITDH && y1 > 0
     && y1 < GameScreen.SCREEN_HEIGHT){
5       this.x = x1;
6    this.y = y1;
7    }
8 }
```

Naming for Constants

- When a constant is declared it is shown in how the variable is named
- Constant variable names are declared using only upper-case characters
- Words are separated by an underscore (_)
- For example `SCREEN_HEIGHT`, `SCREEN_WIDTH` etc...
- Easy to recognise constants

Class Methods (Static Methods)

- Methods can be static too
- Not connected to objects, cannot use instance variables
- A class method is declared by adding the keyword `static` before the return type of the method

Uses of Static Methods

- In many games, the result of an action performed by the character is partly random and partly based on the instance variables of the character
- In older games this random element would be calculated by rolling a number of dice
- We will create this functionality and place it in a class named `Dice`

Declaring Static Methods

- The methods will be declared as static methods
- All of the variables must be static
- The only variable we need is an object based on the `Random` class from the package `java.util`, which we will use to roll our dice.
- We will add two methods `roll1D6` and `roll2D6`
- These will implement the functionality of rolling a single 6 sided dice and 2 6 sided dice respectively

```java
public class Dice {
   private static Random rng = new Random();

   public static int roll1D6(){
      int d = rng.nextInt(6);
      return 1+d;
   }

   public static int roll2D6(){
      int d = roll1D6() + roll1D6();
      return d;
   }
}
```

Calling a Static Method

- When we are calling a method, we needed a reference to an object of that class
- The syntax for calling a static method is `ClassName.methodName(parameters);`
- This means that we would call our methods like this `Dice.roll1D6();` or `Dice.roll2D6();`

```
1  private int randomEffect(int effect){
2     int r = Dice.roll2D6();
3     if(r == 2) {
4        effect = 0;
5     } else if(r < 6){
6        effect = (int) (effect * .6);
7     } else if(r > 8 && r < 12){
8        effect = (int) (effect * 1.5);
9     } else if(r == 12){
10       effect = effect * 5;
11    }
12    return effect;
13 }
```

The Main Method

- `public static void main(String[] args)`
  - `public` - The access level of the method (can be called from anywhere)
  - `static` - The method belongs to the class (do not need an object to call it)
  - `void` - The return type
  - `main` - The name of the method
  - `String[] args` - This is the parameters passed to the program on the command line

Enumerated Types

- There are many types in Java
- We can create new types by defining classes
- We can also define new types using an enumerated type
- An enumerated type, usually called an enum, is a type with a **fixed list** of possible values
- These values are specified when the enum is defined

Enum Example

- To represent majors we could use a String ("Software Engineering", "Internet of Things Engineering" , "Electronic Engineering" or "Finance")
- Any string is allowed
- Enumerated type lets specify exactly what the allowed values are

Syntax of Enumerated Type

```
enum EnumName { list of possible values }
```

Enum Naming

- The possible values of an enumerated type are constants, so we use the same naming rules when we declare them

- Majors would be named `SOFTWARE_ENGINEERING`, `INTERNET_OF_THINGS_ENGINEERING`, `ELECTRONIC_ENGINEERING` and `FINANCE`

Enum Naming

- For the actual enumerated type, we use the same naming rules as a class, so we would use something like `Major`

Major Enum

```
1 public enum Major{
2    SOFTWARE_ENGINEERING,
     INTERNET_OF_THINGS_ENGINEERING,
     ELECTRONIC_ENGINEERING, FINANCE
3 }
```

Using enums

- Constructed automatically
- Declare like variable, E.g. `Major degree;`
- Assign like a constant, E.g. `degree = Degree.SOFTWARE_ENGINEERING`