Object Oriented Development, Abstraction & Inheritance

CSCU9A3
Data Structures, Objects and Algorithms

David Cairns

Abstraction

- The key to building a stable system is abstraction.
- Abstraction allows us to form a view of a concept which considers only necessary info and hides unnecessary detail.
- Programming language design has seen a progression of abstraction mechanisms . . .
  - Procedural Abstraction
  - Data Abstraction
  - Object Oriented Data Abstraction

Procedural Abstraction

- Consider a function called power which raises one number to the power of another:

```
// Return x raised to the power p
double power( double x , int p ) ;
```

- We could change the implementation (perhaps make it more efficient) and the code that invokes it won't have to be changed.

Data Abstraction

- Each language provides a basic set of types
  - e.g. float, int, char in Java and gives the user operations for manipulating variables of these types (e.g. +,-,* etc).
- Data Abstraction takes the idea of abstraction further - instead of just defining operations, we can define our own data types.
- Data Abstraction allows the definition of new types, complete with a set of operations and methods, which can then be used as if they were part of the language. We call these Abstract Data Types (ADTs).

Object-Orientation

- Object orientation takes data abstraction a step further via the use of:
  - Encapsulation
  - Inheritance
  - Polymorphism
- We will look at these in more detail . . .

Encapsulation

- Encapsulation means that the internal details of an ADT (e.g. any data) should only be accessed via methods.
- Restricting access to methods stops the state of the ADT from being modified in undesirable ways.
- It also insulates the user of a class from dependency on internal details

Encapsulation via Access Protection

- Most OO languages including Java allow you to protect fields from external modification. This allows you to guarantee that objects are in a consistent state.
  - For example, inside a Car class we'd like to be confident that the speed can never be greater than the maximum speed, i.e. we want a way to make the following illegal:

```
Car c = new Car();
c.maxSpeed = 100.0;
c.speed = 150.0; // Want to prevent this!
```

  - This code violates the conceptual constraints of the class. We ideally want the compiler to enforce these constraints.
  - To achieve this, we can specify who will be allowed to access which parts of the class.

Encapsulation

```
public class Car
{
    private String numberPlate;
    private double speed;
    private double maxSpeed;

    public Car(String numberPlate, double maxSpeed)
    {
        if (maxSpeed < 0.0)
            throw new IllegalArgumentException();
        this.numberPlate = numberPlate;
        this.maxSpeed = maxSpeed;
    }

    public double getMaxSpeed() { return speed; }

    public double getSpeed()  { return maxSpeed; }
}
```

The Benefits of Encapsulation

- Encapsulation has three main benefits:
  - It allows you to enforce constraints on an object's state.
  - It provides a simpler interface to the class. Users of the class don't need to know everything that's in the class in order to use it, only the public parts.
  - It separates interface from implementation, allowing them to vary independently. For instance, we could make the numberPlate field of Car a NumberPlate class instead of a String .

Polymorphism

- Derives from the Greek word for "many forms".
- It is central to Object-Orientation.
- Encourages the creation of class hierarchies, in which families of behaviour are related by inheritance.

Inheritance - Scenario

- Suppose that we now wish to create a traffic simulation to monitor the environmental cost of different forms of transport. In addition to cars, we might also wish to include trains, planes and boats.
- Using polymorphism, we can create a *framework* for the simulation by abstracting the key features that are common to all these forms of transport.
  - Our framework could manage and model a set of Vehicles
- Car, Train, Plane and Boat are all examples of the abstract concept Vehicle.
  - You could add your own derivation of a vehicle later
  - You do not need to predict in advance what other types of vehicle may be used

Vehicle Class

```
public class Vehicle
{
     protected String name;
     private double speed;
     private double maxSpeed;
     private double heading;

     public Vehicle()
     {
          name = "?";
          maxSpeed = 0;
          speed = 0;
     }

     public Vehicle(String nm)
     {
          name = nm;
     }
```

Vehicle Class

```
    public void turn(double degrees)
    {
        heading = (heading + degrees) % 360;
    }

    public void accelerate(double v)
    {
        speed = speed + v;
    }

    public double getCurrentSpeed() { return speed; }

    public String toString()
    {
        return "Vehicle - " + name + ": " + speed;
    }

    // Further methods that apply to all forms of Vehicle...
}
```


Boat Class

```
public class Boat extends Vehicle
{
    private double   displacement;
    private int      numSails;

    public Boat(String n, double displace, int sails)
    {
        name = n;
        displacement = displace;
        numSails = sails;
    }

    public String toString()
    {
        return "Boat - " + name + " : Displaces " + displacement;
    }
}
```

Aircraft Class

```
public class Aircraft extends Vehicle
{
    private int      numWings;
    private double   wingLift;

    public Aircraft(String n, int wings, double lift)
    {
        name = n;
        numWings = wings;
        wingLift = lift;
    }

    public String toString()
    {
        return "Aircraft - " + name + " : Lift " + wingLift;
    }
}
```


Transport Manager Class

```
import java.util.ArrayList;

public class TransportManager
{
    private ArrayList<Vehicle> vehicles;

    public TransportManager()
    {
        vehicles = new ArrayList<Vehicle>();
    }

    public void addVehicle(Vehicle v)
    {
        vehicles.add(v);
    }

    public void describeTransport()
    {
        for (Vehicle v : vehicles)
        {
            System.out.println(v.toString());
        }
    }
}
```

TransportTest Class

```
public class TransportTest
{
      TransportManager manager = new TransportManager();

      public static void main(String[] args)
      {
            TransportTest test = new TransportTest();

            test.go();
      }

      public void go()
      {
            Vehicle v = new Vehicle("The Vehicle");
            Boat  b = new Boat("Maltese Falcon",1240,15);
            Aircraft a = new Aircraft("Concorde",2, 5000);

            manager.addVehicle(v);
            manager.addVehicle(b);
            manager.addVehicle(a);
            manager.describeTransport();
      }
}
```


TransportTest – Output

```
Vehicle - The Vehicle: Speed 0.0
Boat - Maltese Falcon : Displaces 1240.0
Aircraft - Concorde : Lift 5000.0
```

Inheritance is your friend...

- You are already benefitting from inheritance when you build your own Java GUIs
  - You get most of the behaviour you need for free
  - You only add/adjust the bits you want to be different
  - Look at API guide to see how much you are getting
  - It's also the reason why you do not write a `main` method when developing your GUI code. You are inheriting from a class that already has the main method in it.