Introduction to Object-Oriented Programming Concepts

Dr. Martin O'Connor
CA166
www.computing.dcu.ie/~moconnor

Topics

- State and Behaviour
- Data Encapsulation
- Classes and Objects
- Inheritance
- Interfaces
- Packages and API

Object-Oriented Programming (based on the Oracle Java Tutorial)

- An object is a software bundle of related state and behaviour
- Software objects are often used to model the real-world objects that you find in everyday life
- Examples of real world objects: dogs, bicycles, pens, cars, radios, etc.
- Real-world objects share two characteristics:
  - They all have *state* and *behaviour*

Examples of Object - State and Behaviors

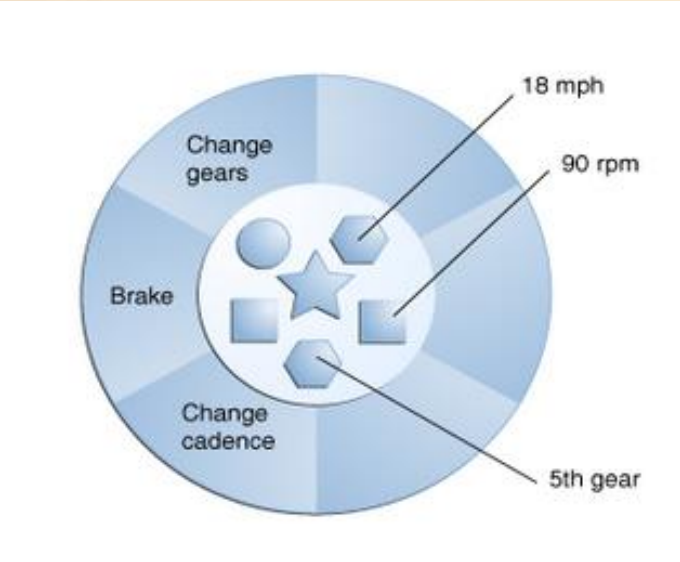
- Dogs
  - *State*: name, colour, breed
  - *Behaviour*: barking, fetching, wagging tail
- Bicycles
  - *State*: current gear, current speed, current pedal cadence
  - *Behaviour*: change gear, change cadence, apply brakes
- Radio
  - *State*: On, off, current volume, current station
  - *Behaviour*: turn on, turn off, change volume, change station, scan

Software Objects

- Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming
- Software objects are conceptually similar to real-world objects: they too consist of state and related behaviour
- A Software objects
  - stores its state in *fields* (also known as variables)
  - exposes its behaviour through *methods*
- Methods operate on an object's internal state
- Methods serve as the primary mechanism for object-to-object communication.
- Classes are a blueprint or template used to create specific objects.

Data Encapsulation

- Hiding the internal states and requiring all interaction to be performed through an object's methods is known as *data encapsulation*
- *Data encapsulation* — is a fundamental principle of object-oriented programming.
- By encapsulating its own state and behaviour, an object remains in control of how the outside world is allowed to use it
- A bicycle modelled as a software object.



Advantages of Data Encapsulation

- Advantages of Data Encapsulation:
  - *Modularity*:
    - The source code for an object can be written and maintained independently of the source code for other objects.
  - *Information-hiding*:
    - By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

Advantages of Data Encapsulation (Continued)

- Advantages of Data Encapsulation:
  - *Code re-use*:
    - If an object already exists (perhaps written by another software developer), you can use that object in your program. Allows for specialists to implement/debug complex, task-specific objects, which you can then trust to run in your own code.
  - *Pluggability and debugging ease*
    - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

What is a Class?

- In the real world, there are many individual objects - all of the same kind.
  - E.g.: 100's of bicycles of the same make and model.
- In object-oriented terminology, we say that a particular bicycle is an *instance* of the *class of objects (bicycles)*
- So, a *class* is the blueprint from which individual objects are created.
- All Java programs consists of at least one class
- When designing a class, ask yourself:
  - What possible states can this object be in?
  - What possible behaviour can this object perform?

An Example of a Bicycle Class

- One possible implementation in Java of a bicycle class:
  (NOTE: class names should always start with a capital letter!)

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

An Example of a Bicycle Class (Continued)

- The fields cadence, speed, and gear represent the object's state
- The methods (changeCadence, changeGear, speedUp etc.) define how the object interacts with the outside world
- NOTE: The Bicycle class **does not** contain a main method.
  - because it is not a complete application; it's just the blueprint for bicycles that may be *used (or created)* by an application
  - The responsibility of creating and using new Bicycle objects belongs to some other class in your application

BicycleDemo Class

- A *BicycleDemo* class that demonstrates how to create two separate Bicycle objects and how to invoke their methods

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

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

Relevant Terminology

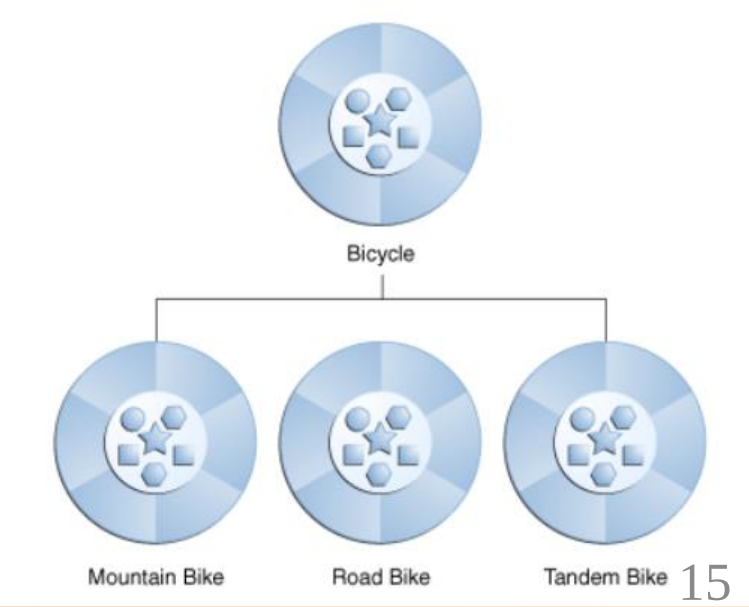
- "**Object Reference**"
  - The identifier of the object
- "**Instantiating an object**"
  - Means creating an object of a class and assigning initial values to the object's fields (states)
  - Objects must be instantiated before they can be used (just like a variable must be declared before it can be used).
- "**Instance of a class**"
  - An object after instantiation.
- "**Members of a class**"
  - The class fields and methods
- "**Method**"
  - The Java statements to manipulate the object's fields or perform some computations.

What is Inheritance?

- Different kinds of objects often have a certain amount in common with each other
  - E.g.: mountain bikes, road bikes, and tandem bikes
- They all share the characteristics of bicycles (current speed, current pedal cadence, current gear)
- Yet each also defines additional features that make them different
  - tandem bicycles have two seats and two sets of handlebars
  - road bikes have drop handlebars
  - mountain bikes may have an additional chain ring, giving them a lower gear ratio, and thus, allowing a cyclist to more easily cycle up hills

What is Inheritance? (Continued)

- Object-oriented programming allows classes to *inherit* commonly used state and behaviour from other classes
- In the example below, Bicycle now becomes the *superclass* of MountainBike, RoadBike, and TandemBike
- Each class is allowed to have one (and only one) direct superclass, and each superclass may have an unlimited number of *subclasses*



What is Inheritance? (Continued)

- The syntax in Java for creating a subclass is simple
  - At the beginning of your class declaration, use the *extends* keyword, followed by the name of the class to inherit from

```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here

}
```

- The class MountainBike has *(inherits)* all of the same fields and methods as the class Bicycle
- However, the class MountainBike can also create new fields and methods for the particular features that make a mountain bike unique.

Example of Inheritance

- On the left, an example of one possible implementation of a MountainBike class and
- On the right, an example of a driver program (called MountainBikeDemo) to demonstrate how to create and use objects of the MountainBike class

```
class MountainBike extends Bicycle {

    int chainRing = 0;

    void changeChainRing(int newValue) {
        chainRing = newValue;
    }

    void printMountainBikeStates() {
        System.out.println("cadence:" +
            cadence + " speed:" +
            speed + " gear:" + gear +
            " chainRing:" + chainRing);
    }
}
```

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

        // Create a MountainBike object
        MountainBike mBike1 = new MountainBike();

        // Invoke methods on the
        // MountainBike object
        mBike1.changeCadence(50);
        mBike1.speedUp(10);
        mBike1.changeGear(2);

        mBike1.changeChainRing(1);
        mBike1.printMountainBikeStates();
    }
}
```

What is an Interface?

- We have learnt that objects define their interaction with the outside world through the methods that they expose
- Thus, methods form the object's *interface* with the outside world
  - E.g.: the power button on a TV is the interface between the user (human) and the electrical wiring inside the TV that powers on/off the TV
- In Java, an interface is a group of related methods with empty bodies
- A bicycle's behaviour, if specified as an interface, might appear as follows:

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

Example of an Interface

- To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ABCDBicycle), and you use the *implements* keyword in the class declaration

```
class ABCDBicycle implements Bicycle {

    // remainder of this class
    // implemented as before
}
```

- Q: Why an Interface?
- Ans: Implementing an interface allows a class to become more formal about the behaviour it promises to provide.
  - Interfaces form a contract between the class and the outside world
  - This contract is enforced at build time by the compiler
  - All methods defined by an interface must appear in the source code of a class implementing that interface before the class will successfully compile.

What is a Package?

- A package is a namespace that organises a set of related classes and interfaces
- Conceptually, packages are similar to different folders on your computer
  - Documents in one folder, Images in another folder, movies in another folder, music in another, and so on.
- Because software written in Java can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

Application Programming Interface (API)

- The Java platform provides an enormous class library (a set of packages) available for use in your own applications.
- This library is known as the "Application Programming Interface", or "API" for short.
  - For example, a String object contains state and behaviour for character strings
  - a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the file system
  - various GUI objects control buttons and checkboxes and anything else related to graphical user interfaces
  - The *Java Platform API Specification* contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java platform (available online at the Oracle Java Website)

How to Reuse a Class

- You don't need to know how the class is written or see the code of the class
- You do need to know the application programming interface (API) of the class.
- The API is published and tells you:
  - how to create objects
  - what methods are available
  - how to call the methods