1 Objectives

At the end of this topic you should have an understanding of what object-oriented programming (or OOP) is, what its main advantages are, and what purpose it serves. You will also learn to determine what to put in a class, how to implement a class, and how to create objects from classes.

2 What is object-oriented programming?

Object-oriented programming is a programming paradigm (a programming style) based on the specification, implementation, and use of objects.

2.1 What is an object?

An *object* is a data type with structure and state. Each object defines methods that can access and manipulate the state. These methods exist to be called by other objects. In other words, an object is an entity that performs an action in response to a request from another entity.

2.2 So, yeah, what's the point?

Suppose you have a car, and suppose your car starts making strange grumbling noises. Unless this is usual behaviour, you may think there is a problem that needs some fixin'. You could try to do the repairs yourself, but if you're like me, you could really screw up. But have no fear! You can still get your car fixed! Like most people, you might bring your car to the shop, where professionals — who know how to deal with strange grumbling noises — can fix your car for you.

In particular, we solved the car problem by finding an appropriate agent (the shop) and sending them a message containing my request (please fix my car). It is the *responsibility* of the shop to fix the car and to satisfy the request. There is some *method* — that is, some algorithm or set of operations — used by the shop to do this. I don't know (and don't even know if I want to know) *how* they are going to go about fixing the grumble. In fact, knowing the details just may confuse me even more, and it is best that I leave the job to them. I really just care that they get the job done, even if the shop sends the car to another shop who in turn sends my beloved piece-of-shit car to a third shop where it is finally fixed and sent back up the chain.

And what is the point of my story? The point is that there are certain people, entitities, or *objects* that are better equipped to perform certain tasks than others. We would like *them* to deal with these problems, and *abstract away* (hide) the implementation details by hiding how they perform their task.

3 Advantages of OOP

Programming using objects has many advantages, some of which are:

1. the user doesn't have to worry about implementation details — they can assume that the object performing the task will do it correctly
2. the programmer doesn't have to worry about telling the user about minute changes to the implementation
3. it organizes the code into chunks that better represent real-life objects, and therefore is logically easier to understand program flow

4. it places related code all in one place, making the program easier to write, read, debug, and maintain
5. it promotes code reuse

4 The Three Main OOP Principles

When people talk about OOP, they use many of the same catch phrases. Three specific principles creating a buzz are:

1. information hiding and encapsulation
2. polymorphism
3. inheritance

We will cover each of these three principles in detail below.

5 Our Running Example

In the following sections we are going build up one example: The Human Being, or Person.

I think we are all familiar with the concept of a person. There are many different kinds of people (duh) such as males and females, students and teachers, employees and employers, adults and children, athletes and couch potatoes, etc. But despite the differences, all people share certain properties common to a 'Person'. For example, a person has a name, a height, a weight, a nationality, a father and mother, a favourite colour, a birthday, a recipient of their first kiss, etc. Furthermore, there are some common actions we would like to perform on a person (mind out of gutter!) such as getting their name, height, etc, or maybe changing their name (for marriage or witness protection program), changing their weight (liposuction), changing your parents, oh wait, you can't do that.

So in OOP terms, our object is going to be a Person. Who are the users of this object? Well, perhaps there is some dating service that would like to know this information to match up partners, or an employment office to find a suitable employee, or maybe it's just me being nosey.

A sample skeleton for class `Person` may be something like

```
class Person
{
    private String   name;
    private int      height;
    private Person   mother, father;
    private Person   firstKiss;
    private bool     likesMushrooms;
    public           Person( String n )     { name = n; }     // the constructor
    public  String   toString()            { return name; }  // a standard method
    public  int      getAge()              { ... }       // another method
    public  int      rateDatingPotential()  { ... }
}
```

We will build further on this example in the next few classes.

6 Information Hiding and Encapsulation

When we think of an object, we should think of it as an *atomic unit*, i.e. and indivisible entity. What this means is the user of an object should not be allowed to see or tamper with the inner workings of the object, just as I did not want to tamper with the engine of my car, or you would not want to dissect how the computer internally manipulates floating point numbers. Instead, the object is the only one with access to its own state, and everyone else has only indirect access through the object's provided methods.

This concept of hiding implementation details, and making certain components of an object inaccessible is called *information hiding*, or *abstraction*. *Encapsulation* is the combination of grouping related data and operations in an common object together with hiding the implementation of this object from its users.

In our `Person` example, we use encapsulation by creating an object to contain related person properties such as name, height, and birthday. We further hide the implementation of this object by hiding these properties from a user so that they can only access them and change them in a valid manner.

So we've determined that we would like to group related data together and hide implementation details. How do we do this? First of all, how do we determine what your object is going to be:

- what are its fields?
- what are its methods?

To help with this, we think of encapsulation as a "has-a" relationship:

- a person *has a* name
- a person *has a* birthday

In other words, the *fields* of the class should be the properties of the object, and the *methods* of the class should be ways to modify and access the values of these properties.

Suppose you are a dating service in the process of matching up your various wacky clients. You may want to find out the 'dating potential' of each person and pair them accordingly. (fyi: I don't pretend to know how dating services actually work, I'm just guessing here). You could do this all yourself by figuring out some formula based on a person's name, weight, hair colour, mushroom preference, etc. but to do this for each person would be very tedious and time consuming. A more efficient solution would be to ask each individual person to calculate their own potential (based on the same formula) and send it back to you:

```
Person    adam     = new Person( "Adam Sandler" );
int       rating   = adam.calculateDatingPotential();
```

And this way the person can sort out their own rating without having to tell the dating service their personal details such as weighing 180 pounds (i.e. they can keep things private), and the process will be more accurate and efficient than if the service were to do it. The bottom line: leave things to the experts.

The next issue is hiding certain implementation details from the users. How do we do this? Fortunately, Java provides certain *modifiers* to help us in our task:

- `public`: allows method to be inherited and accessed from outside the class
- `private`: prohibits method from being inherited or accessed from outside the class
- `protected`: allows method to be inherited but prohibits it to be accessed from outside the class. This is useful because subtypes of a class often need to know some of the internal details of the implementation in order to work efficiently.
- `static`: limits the number of instances of this method to one and allows access without an object. I.e. the one instance is shared by all object instances.

- `final`: cannot override (inherit) this method any more
- ⟨none⟩: same as `public`

For example, each person may wish to keep their weight private, and we would only want qualified people (a doctor or the person themself) to be able to change it (so this variables would be `private`).

7 Polymorphism

The second important principle is *polymorphism*. This is the idea that reference type can reference objects of several different types. When a method is applied to a polymorphic type, the operation that is appropriate to the actual referenced object is automatically selected.

In Java, polymorphism is implemented as part of inheritance. This allows us to implement classes that share common logic.

8 Inheritance

We would like to be able to extend the behaviour of a class without having to reimplement what we have already done. I.e. to upgrade the video card in your computer, you do not send the computer back to the manufacturer. The mechanism used in the object oriented paradigm to express hierarchical relationships is called *inheritance*.

Where encapsulation is known as a "has-a" relationship, inheritance is seen as a "is-a" relationship.

- a female is a person
- a student is a person (when they're not walking, talking zombies)

And we would rather not have to rewrite our age calculation for the female or student. Rather, we would prefer to use what we have already written for the generic person, thus saving us time, hassle, money, etc.

Some other common examples of inheritance are: class `Car`, `Truck`, and `Airplane` could all inherit from class `Vehicle`. Some inherited properties may be seating capacity, number of wheels, type of fuel, but they may differ in that the plane may have a propellor or the truck a cab in the back. Another inheritance example may be that class `MusicVideo` and `Comedy` could inherit from a class `TVProgram`.

8.1 Why Inheritance?

Inheritance has many benefits, including:

- it promotes code reuse by allowing you to change the behaviour of the class without having to rewrite the code of the class.
- it logically organizes your code into simple relationships, with subclasses building on the properties of their super, or parent class

8.2 Inheritance in Java

In Java, every class is implicitly a subclass of `Object`.

You inherit in Java using the `extends` keyword:

```
class Female extends Person
{
    private int      pairsOfShoes;
    private int      bustSize;
    private Person[] wantsToDate;
    public  int      rateCattiness();
}
```

```
class Male extends Person
{
    private int      hrsWatchingSports;
    private boolean  isBoxers;
    private Person[] wantsToDate;
    public  int      rateInsensitivity();
}
```

A class can extend exactly one other class, so if you would like to inherit from more than one superclass, you can't. Why is this? And how can you get around this?

For example, suppose we have we'd like to have a new class `Transvestite` that inherits from both class `Female` and class `Male`:

```
class Transvestite extends Female, Male
{ ... }
```

And now suppose we execute the statements

```
Transvestite t = new Transvestite( "Pat" );
t.rateDatingPotential();
```

If both Female and Male have different implementations of `rateDatingPotential()` (which they should!) then whose method gets called?!? Is it `Female`'s or `Male`'s? Hm.

The solution to this problem is a construct called an *interface*, which we will cover in the next handout.

8.3 Overriding Methods

So far, in discussing inheritance, we have assumed that when object types inherit methods, they do not need to change the behaviour of the inherited methods. This is not always going to be the case, for example, as mentioned above, the `rateDatingPotential()` method should be more extensive for the `Female` class than for just `Person`. So we now need to turn to ways for manipulating methods in object subtypes.

The default behaviour is that a subtype inherits the methods of its super-type. But we can override some of these methods (the non-`final` ones) to change their behaviour. Note that we can override only if:

- both methods have the same name
- both methods are class methods or instance methods (not `final`)
- both have the same number and type of parameters

```
class Female extends Person
{
    public int getWeight()              { return (weight - 10); }
}
```

We will discuss inheritance in further detail when we talk about interfaces.

9 Implementing Objects

A *class* is Java's blueprint, or description, of an entity, and an object is just an instance of a class, i.e., a specific entity. The structure of a class is:

- fields: represent the state of object
- methods: called by dispatcher to perform some action, must be given the message, and can view the state
- constructors: performs any special initialization (setting default values, etc.)
- others: destructors, helpers, etc.

Please see section 5 (Our Running Example) for an example of a class.

10 Using Objects

Recall that an object is an instance of a class. It has its own copies of instance variables and instance methods (fields and methods that are not modified as `static`). It interacts with other objects through method calls. You create an object with `new`:

```
Person p = new Person( "Hans Solo" );
```

And you call methods using the accessor operator (.):

```
String n = p.getName();
```

11 Reference Types

A *reference* is an address of an object, i.e. it is a value representing the location in memory where we can find the object. When an object gets created through a constructor call, the constructor does not actually return an object of the requested type. Rather, it returns the *address*, or "mailbox number" of where the created object has been placed. Java does not allow you to "store" the actual object in a variable, so instead it stores this address. If I need to use the object stored in the variable, Java will automatically go to the stored address and pick up my variable for me. For example, in the following code:

```
Person barbie        = new Person( "Barbie" );
Person ken           = new Person( "Ken" );
Person favouriteDoll = barbie;
```

`barbie`, `ken`, and `favouriteDoll` are all variables of type `Person`, but they store the address of the newly created `Persons`. If you try to print one of the above variables, it will only print out jibberishcrap like `Person@34059`. Furthermore, by changing the contents of the reference variable `favouriteDoll` I've really stored the address of a different object.

And last but not least, when you pass an object as a parameter, you are in fact not passing the object itself but the address of that object, i.e. a reference to that object. Similarly, when you return an object from a method you are really returning a reference to that object.

12 Review

Object oriented programming is the programming style of splitting your code into entities called objects. These objects interact with each other by calling each other's methods.

A class is a structure representing an entity that contains fields and methods relating to this entity. An object is a specific instance of a class. To determine what goes inside a class, and what should actually *be* a

class, it helps to think of the "is-a" and "has-a" relationships (representing inheritance and field definition, respectively).

The 3 main principles of OOP are

- encapsulation & information hiding: grouping related properties together into one object and hiding implementation details from the user
- inheritance: the mechanism used to express hierarchical relationships
- polymorphism: allowing a reference type to reference objects of several different types

Encapsulation is implemented by creating classes based on related data and functions, and hiding implementation details using special modifiers. Inheritance is implemented using either the `extends` or `implements` keywords, and helps us to incrementally add functionality to existing classes without having to rewrite a lot of code. Polymorphism in Java is a side-effect of inheritance and allows us flexibility in what we assign to reference types.