COMP2511

Tutorial 2

Last Week's Tutorial

- Java syntax
  - Declaring variables, writing loops, running the main function, performing I/O, importing libraries.
- Introduction to classes
  - Creation, fields and methods, invoking methods, instantiation, getters and setters, constructors.
- Throughout the tutorial, I will assume that you are comfortable with all of these ideas, but please interrupt me at any time if you need anything clarified. This will be a **very** content-heavy tutorial if you have not watched the lectures!

This Week's Tutorial

- Static fields and methods
  - What if it makes more sense to make something belong to the actual *idea* of a class, rather than a concrete instance of the class?
- Inheritance, method overriding and polymorphism
  - How can we avoid code reuse, and create a relationship between classes that are related by an 'is-a' relationship?
  - How can we specialise the behaviour of a subclass?
- JavaDoc
  - What are the conventions behind code documentation in Java projects?
- Access modifiers
  - How do we enforce specific restrictions on what parts of our classes should be visible to other classes?

Lab 1 Reflections

- In Lab 1, you were asked to complete the Average class, and its associated computeAverage method.
- In order to access this method, you needed an **instance** of the Average class to call the method from.
  - Average a = new Average();
  - float avg = a.computeAverage(nums);
- This seems a bit finicky. If you created another instance b of the Average class, would you ever expect b.computeAverage(nums) to give a different result to a.computeAverage(nums), if the array stays the same?
  - Nope!
  - This suggests that the usage of computeAverage should **not** be tied to a specific instance of the Average class, but rather the Average class *itself*.

Static Fields and Methods

- A **static method** is a method that belongs to the class itself, not to any specific instance of the class.
- Similarly, a **static field** is a field that belongs to the class itself, not to any specific instance of the class.
  - **Q**: What's one instance where a static field may be useful?
  - **A**: Whenever we have constants. For example, Math.PI is how Java stores π, and belongs to the Math class.
- This means that only one 'copy' of that field/method exists at any given point of time.

Static Fields and Methods

- To make a field/method static, simply add the static keyword.
  - For constants, the **final** keyword makes the field immutable, and by convention constants are written in capitalised SNAKE_CASE.

```
public class Average {
    public static float computeAverage(int[] nums) {
        /* do stuff */
    }
    public static void main(String[] args) {
        int[] array = { /* some array of numbers */ };
        float avg = Average.computeAverage(array);
        System.out.println(avg);
    }
}
```

```
public class Example {
    public static final int MAGIC_NUMBER = 42;

    public static void main(String[] args) {
        System.out.println(Example.MAGIC_NUMBER);
    }
}
```

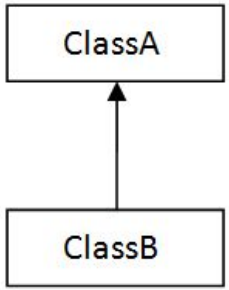
Definitely Not Among Us

- Imagine you're programming a multiplayer murder mystery game, where there are two types of players in each match:
  - **Crewmates**, whose role is to survive for some given amount of time.
  - **Impostors**, whose role is to eliminate all of the crewmates without getting caught.
- You have decided to take an object-oriented approach, so you have created a Crewmate class, and an Impostor class.
- After careful consideration, you have decided to add *medics* into the game, which **are a type of** crewmate, but have the **added ability** to revive an eliminated crewmate once every match.
- One approach to implementing the Medic class is to copy-paste all of the Crewmate logic, and add an extra method to revive crewmates. Does this follow good coding practices? Is there an alternative approach?

Inheritance

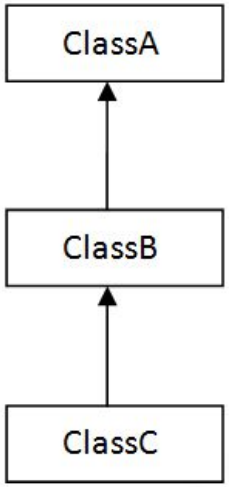
- **Inheritance** refers to the use of an existing class as a basis for the creation of a new class, by making the new class have a copy of **every** field and method from the existing class.
  - This allows us to reuse code within related classes.
  - The class that inherits another class is referred to as the **subclass/child class**, while the class being inherited from is referred to as the **superclass/parent class**.
  - Inheritance enforces an 'is-a' relationship between a subclass and its superclass. If B is a subclass of A, then an instance of B **is also an** instance of A.
  - More fields and methods can be added to the subclass. Hence, a subclass typically has more functionality than its superclass.
- **extends** is the Java keyword to make a class inherit from another (for example, `class B extends A` makes B a subclass of A)
- All classes in Java directly or indirectly extend the **Object** class.

Inheritance Diagram



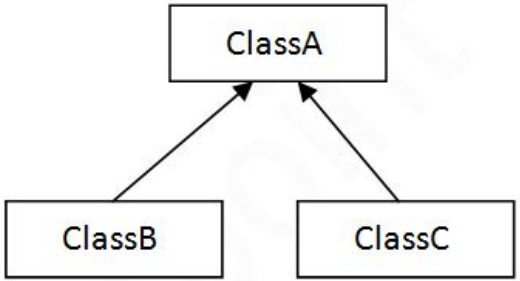
1) Single

ClassB **directly** extends ClassA



2) Multilevel

ClassC **indirectly** extends ClassA



3) Hierarchical

Both ClassB and ClassC **directly** extend ClassA

Inheritance

- **Q:** How can we apply inheritance in the hypothetical situation? Which class should be the superclass, and which should be the subclass?
- **A:** The Medic class should inherit from the Crewmate class, making Medic the subclass and Crewmate the superclass.
- This works because a medic **is-a** crewmate, but with extra functionality that can be implemented by adding more methods in the Medic class.
- This inheritance relationship will mean that `Crewmate c = new Medic();` will work completely fine!
    - This also means that you can add a Medic into a collection of Crewmates, like an ArrayList<Crewmate>.

Method Overriding

- **Reminder!** A subclass can access **all** of its superclass' (non-private) fields and methods.
  - If class A defines a method doSomething() and class B **extends** A, then class B will also have access to doSomething().
- A subclass can provide its own implementation of a method inherited from its superclass, effectively **overriding** its original functionality.
  - The method being overridden by the subclass needs to have the same **method signature** as the one in the superclass (i.e. same method name, exact same order and type of parameters (the naming of parameters is irrelevant)).
- Overridden functions should have the @Override tag on top.
  - This is not strictly enforced by the Java compiler, but helps to explicitly declare your intent and prevent bugs (eg. trying to override a method that cannot be overridden).

Method Overriding Code Example

- What does the following code output?

```
class A {
    public void print1() {
        System.out.println("Hello from A!");
    }

    public void print2() {
        System.out.println("Hello again from A!");
    }
}

class B extends A {
    @Override
    public void print1() {
        System.out.println("Hello from B!");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.print1();
        b.print1();
        b.print2();
    }
}
```

Method Overriding Code Example

- a.print1() prints “Hello from A!”, nothing special
- b.print1() prints “Hello from B!”, since this method has been overridden
- b.print2() prints “Hello again from A!”, since this method has not been overridden

```
Hello from A!
Hello from B!
Hello again from A!
```

Caveats with Inheritance

- Remember, **all** of the attributes and methods of the superclass are carried over to the subclass (access modifiers aside) - we **cannot** cherry-pick only the things that the subclass actually wants!
  - When thinking of when to use inheritance, ensure that it makes sense for the superclass to inherit **everything** from its superclass.
  - If the above is not true, a **composition** relationship may potentially be a more suitable alternative (a **has-a** relationship).
- An instance of a subclass **must** be a valid instance of its superclass (**Liskov Substitution Principle**).
  - Can we make a Square inherit from a Rectangle?

Polymorphism

- What a complicated sounding word!
- **Polymorphism** is the ability to use a **common interface** across **different types/classes,** irrespective of the concrete behaviour of each of the different types.
  - The behaviour can certainly be different depending on the object's *actual* type.
- One example is performing addition on numeric types!
  - The common interface is the addition operator (+)
  - You will always use this operator to add, regardless of if the number on the left is a float, 32-bit integer, ..., and the one on the right is a short, 64-bit integer or similar.

Polymorphism Code Example

- Since the A class (from earlier) defines a method called print1(), we know that any objects of type A (or a subclass of A) must also have this method (overridden or not).
- Hence, if we store a list of objects of type A, print1() is a common interface, so we are guaranteed to be able to call it with no worries.

```
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        A c = new B();
        List<A> myList = List.of(a, b, c);
        for (A elem : myList) {
            elem.print1();
        }
    }
}
```

```
Hello from A!
Hello from B!
Hello from B!
```

Documentation

- Code documentation is the process of providing information about how a piece of code works, or how to use it.
- ‘Self-documenting code’ is code that is inherently readable.
  - Can be accomplished through descriptive variable and function naming, and ensuring that the control flow follows a logical structure.
- Can comments be a **code smell** (i.e. an indication of bad code design)? Why or why not?
  - Having excessive comments can suggest that your design/code is too complex.

JavaDoc

- JavaDoc is a code documentation tool that is very similar to JSDoc, which you are most likely familiar with from COMP1531!
  - Provides details about what a method is doing, as well as its parameters and return value (whenever applicable).
  - We will specify when we want you to write JavaDoc in this course.
  - Type /** and then enter to get a bunch of details auto-filled (assuming your Java extensions are working).
  - Other information can be added however you like, for example using @author to show the reader who wrote the code.

```java
/**
 * Returns the sum of two integers.
 * @param a the first integer to be summed
 * @param b the second integer to be summed
 * @return  the sum of a and b
 */
public int sum(int a, int b) {
    return a + b;
}
```

Live Code Examples

- **src/shapes (Code Review)**
  - Constructors, super(...) vs this(...)
- **src/employee**
  - Class creation, JavaDoc, inheritance, method overriding (toString and equals)
- **src/access (Questions)**
  - Access modifiers