COMP2511

Tutorial 2

Last Week's Tutorial

- Introduction to Java
  - 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, instance vs. static fields/methods.

This Week's Tutorial

- Inheritance
  - How can we reuse existing classes to create other classes related by an 'is-a' relationship?
- Method overriding
  - How can we specialise the behaviour of particular classes?
- Polymorphism, interfaces and abstract classes
  - How can we define and use the methods that are common across classes, regardless of how they are actually implemented?

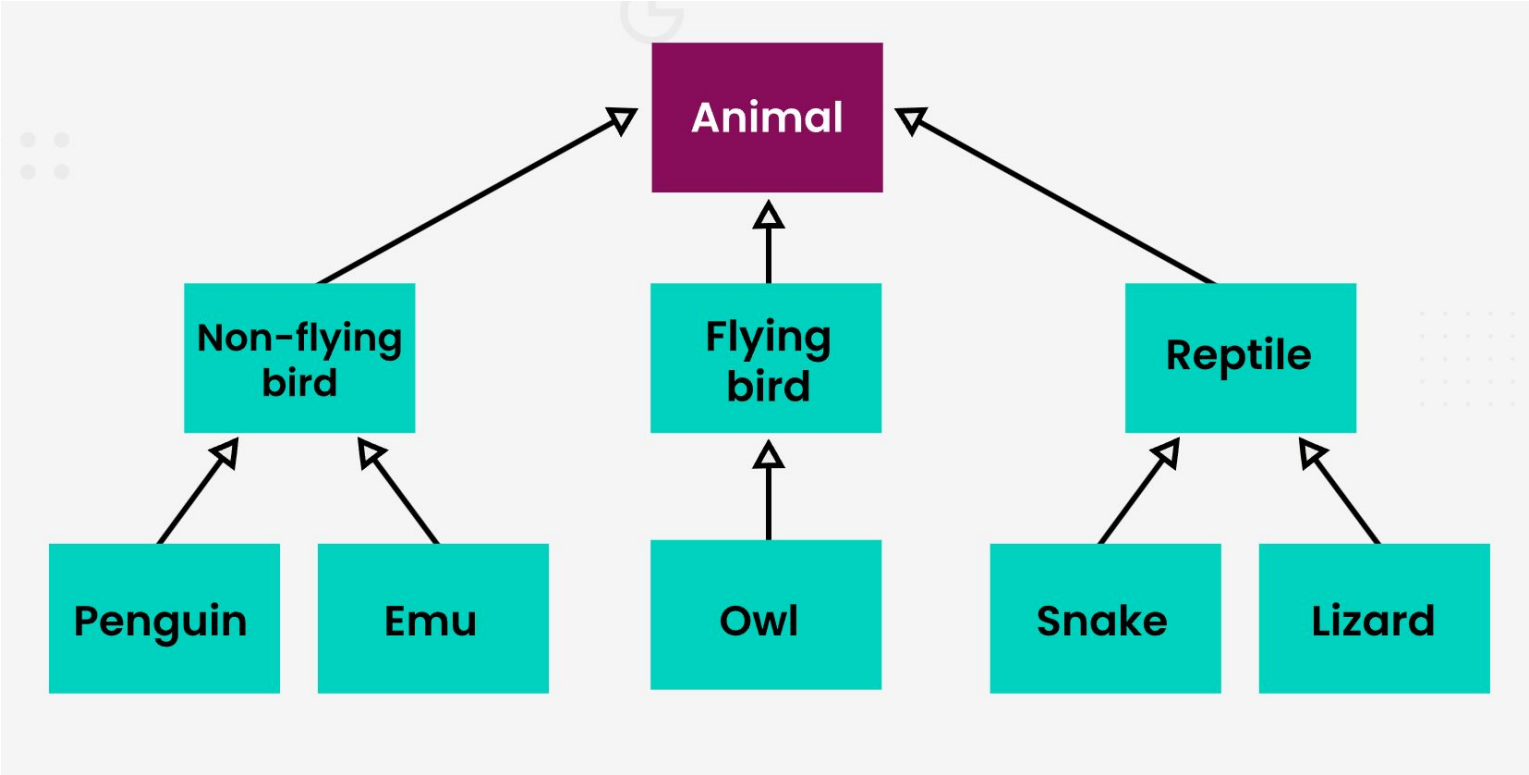
A Suspicious Scenario...

- 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 are subclasses of the **Object** class.

Inheritance Diagram



Inheritance

- How can we apply inheritance in the hypothetical situation? Which class should be the superclass, and which should be the subclass?
  - 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 a List<Crewmate>.

Type-Checking in Java

- **Remember!** If we have a class A which has subclass of B, we still consider instances of B to be instances of A.
  - This also applies for inheritance that goes deeper down. For example, if C was a subclass of B, then instances of C are also instances of A (and B).
- Keeping this in mind, if we want to check if an object is an instance of A **or any subclasses of A**, we use the **instanceof** keyword.
  - For example: `a instanceof A` returns true if the object a is of type A or any of its subclasses, and false otherwise.
- If we want to make an exact comparison on an object's class ignoring subclass relationships, we can compare using the getClass() method.
  - For example: `a.getClass() == b.getClass()` returns true if a and b are of the same exact type, and false otherwise.

Live Example: Inheritance

- Review the **Employee** class in src/employee, which has been documented with JavaDoc.
  - What are the key features of JavaDoc?
  - Should code should always have comments/JavaDoc?
  - Demonstrate how to generate the JavaDoc template on VSCode.
- Create a **Manager** class that is a subclass of Employee and has a field for the manager's hire date.
  - What constructor(s) should we define for the Manager class?
  - Demonstrate how VSCode can generate getters and setters automatically.
  - Is it appropriate to have a getter for the hire date? What about a setter?
  - Why might adding certain getters and setters be bad design?
- [KEY TAKEAWAYS] Writing JavaDoc, subclass creation, thinking about abstraction.

Method Overriding

- **Important!** 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 and parameters).
- All overridden methods should have the `@Override` tag on top.
  - This is not strictly enforced by the Java compiler, but is best practice. It helps to explicitly declare your intent to override a method and prevent bugs (eg. notifying you if you are trying to override a method that does not exist, or using the wrong method signature).

Code Example: Method Overriding

- 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();
    }
}
```

Code Example: Method Overriding

- 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!
```

Live Example: Method Overriding (Pt. 1)

- Recall that all classes in Java are subclasses of the Object class, so it inherits all of Object's methods.
- One of these inherited methods is `toString()`.
  - What does Object's implementation of toString() do?
  - What would actually be useful to include in the result of toString()?
- Override the toString() method in the Employee and Manager classes defined earlier. Also consider whether we can reuse code from the Employee method while writing the Manager method.
- [KEY TAKEAWAYS] Overriding methods, reusing superclass methods.

Live Example: Method Overriding (Pt. 2)

- What is a suitable criterion for two objects to be considered equal?
  - If two objects are instances of the same exact class and have all corresponding fields equal, we can consider them to be equal.
  - There are other ways to define equality, but we will take the above as the definition.
- Does the `==` operator abide by this definition of equality between objects? If not, how does it actually determine equality?
- Another method all classes inherit from Object is `equals()`.
  - What does Object's implementation of equals() do?
  - What would we want our implementation of equals() to do?
- Override the equals() method in the Employee and Manager classes defined earlier. Also consider whether we can reuse code from the Employee method while writing the Manager method.
- [KEY TAKEAWAYS] Same as Pt. 1, type-checking, safe type-casting.

Polymorphism

- **Polymorphism** is the ability to use a **common interface** across **different types/classes** to invoke certain functionality**,** regardless of how that functionality is implemented in each of the classes.
  - This is basically saying that it is the ability to interact with different objects in the same exact way, despite any differences in how they choose to do things.
  - For example, think about the way you turn on your electronics, like your phone, monitor, PC or console. Generally speaking, they all have a power button that you can press to turn it on or off, regardless of what these devices are actually doing under the surface.
  - Another example is if you asked a group of people to say “Hello” in their native language (assuming they all understand what you’re asking). You’ve asked the same question to everyone, but you can certainly expect to hear many different responses back!

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 (remember we can actually store subclasses of A as well), print1() is a common interface, so we are guaranteed to be able to call it, regardless of how each of the objects implement it.

```
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!
```

Abstract Classes

- An **abstract class** is a class that *cannot be instantiated* and allows the declaration of methods without concrete implementations.
- They essentially act as templates to declare a common structure between any subclasses that derive off of it (both fields and methods are inherited down).
    - An example use case would be declaring an abstract class called Mammal containing fields and methods that every mammal should share, and then having separate Dog, Cat, Horse classes derive off it.
    - Concrete classes that inherit an abstract method must provide the concrete implementation for that method.
    - Methods with concrete implementations can still be defined within an abstract class.
- Methods that are declared without concrete implementations are referred to as **abstract methods**.
- Keeping in mind what we have just covered, why would these classes be useful? **(Hint: polymorphism!)**

Interfaces

- An interface is similar to an abstract class, but can only store static constant fields (i.e. **static** and **final**), and all methods are abstract by default.
  - There is a way to define a concrete implementation for a method in an interface, but this is seldom used due to the fact that instance fields cannot belong to an interface.
- Interfaces are used to define a common set of methods that every class that **implements** that interface must provide the concrete implementation for.
  - Hence, they are another useful tool for polymorphism!
- Importantly, each class in Java can only have one superclass, but can implement as many interfaces as it would like.

Live Example: Abstract Classes and Interfaces

- Look at the code in the src/dogs package, including the main method in Chihuahua.java.
  - What is the purpose of using an interface in this code?
  - What are some downsides of the use of the interface here?
  - What is the difference between an abstract class and an interface? Why would you use one or the other?
  - If you have time, refactor the code to improve its quality, perhaps using an abstract class.
- [KEY TAKEAWAYS] Interface and abstract class syntax, recognising where the use of either is applicable

Extra Material

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 **has-a** relationship may be a more suitable alternative (i.e. one class stores an instance of another class and delegates some functionality to it).
- 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? We require that a Square can be treated as a valid instance of a Rectangle.

Access Modifiers (The Definitive Version)

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Code Review

- Open src/shapes, and review the Shape and Rectangle classes.
- Answer the following questions (answers on the following slide):
  - What is the difference between `super` and `this`?
  - What about `super(...)` and `this(...)`?
  - What will the code print and why?
  - Is a call to `super(...)` necessary in the constructor of a subclass? If so, why?

Code Review Answers

- Answers:
  - `super` refers to the parent object, while `this` refers to the current object.
  - `super(...)` refers to a constructor for the parent class, while `this(...)` refers to a constructor for the current class.
  - “Inside Shape Constructor”, “Inside Rectangle constructor with one argument”, “Inside Rectangle constructor with three arguments”; backtrack the method calls.
  - It is necessary. In order to construct a subclass, its parent class needs to be constructed first. If no explicit call to a superclass constructor is made, Java will implicitly try to call `super()`, which is implicitly defined if the user has not defined any constructors explicitly.