COMP2511

Tutorial 2

Last Week's Tutorial

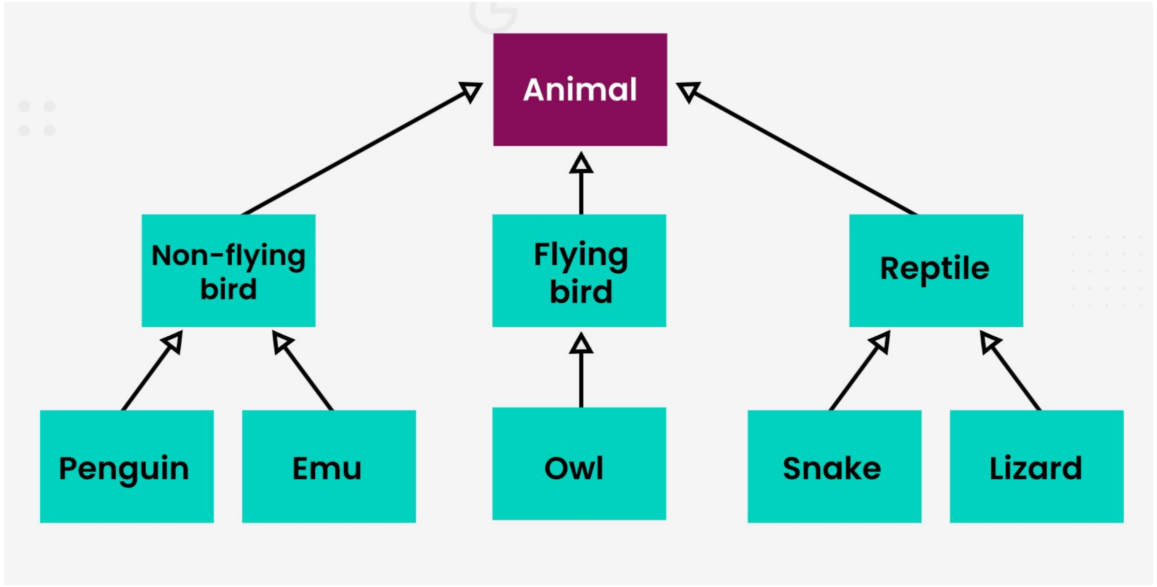
- Course Introduction
- Introduction to Java
- Introduction to Classes

This Week's Tutorial

- Inheritance
- Method Overriding
- Interfaces and Abstract Classes
- Polymorphism

Inheritance

- At its core, inheritance is about reusing existing classes to create new classes. Think of these new classes as **extensions** of another class.

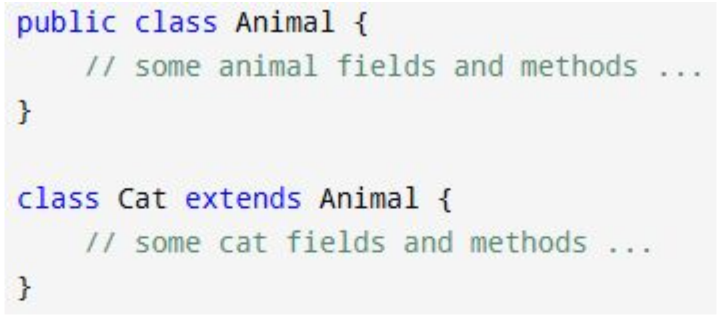


Inheritance

- **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.
    - 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 doubles-up as a way for us to reuse code *and* extend upon existing systems.
        - If you were modelling two related classes (say A and B) with a lot of shared attributes, you could capture all of the shared attributes in a parent class and make A and B inherit this parent class.
        - If you had to add behaviour on top of an existing class, you could create a new subclass of the existing class containing the new functionality.

Inheritance

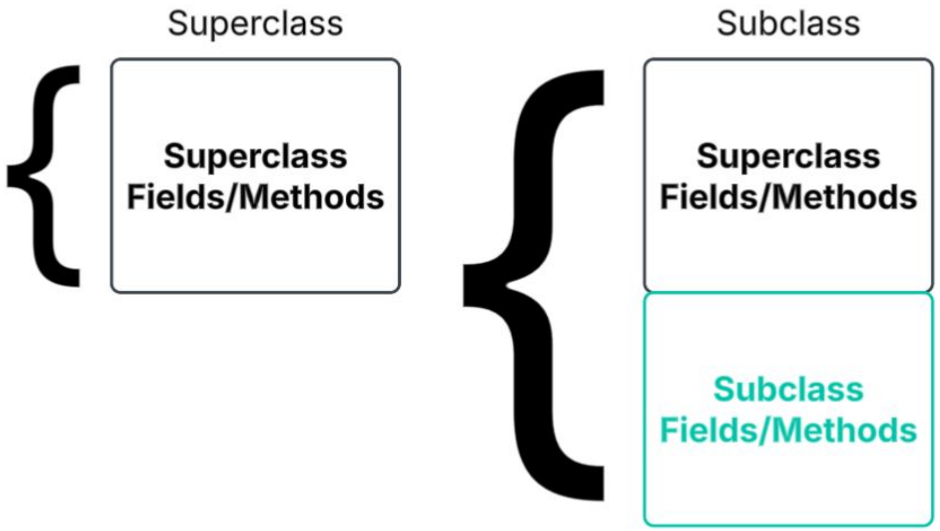
- Inheritance enforces an **‘is-a’** relationship between a subclass and its superclass.
  - **If B is a subclass of A, then an instance of B is also an instance of A**. Use this as a litmus test to determine if inheritance is appropriate! e.g. a Cat is an Animal.
  - A subclass should be able to do the same things as its superclass, most likely more.
- All classes in Java are subclasses of the `Object` class.
- In Java, the **`extends`** keyword makes a class inherit from another.



```
public class Animal {
    // some animal fields and methods ...
}

class Cat extends Animal {
    // some cat fields and methods ...
}
```

Inheritance

- Think of parent classes and subclasses like this; the subclass is the parent class with more stuff 'attached' on top.



Quiz: Inheritance

- Suppose `Cat` is a subclass of `Animal`. Are each of the following valid?

```java
public static void main(String[] args) {
    Animal a = new Cat();
}
```

Yes; all Cats are Animals.

```java
public static void main(String[] args) {
    Cat c = new Animal();
}
```

No; not all Animals are Cats.

```java
public static void main(String[] args) {
    List<Animal> l = new ArrayList<Animal>();
    l.add(new Cat());
}
```

Yes; all Cats are Animals.

Live Example: JavaDoc and 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?
  - What is meant by the term "self-documenting code"?
- 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.

Type-Checking in Java

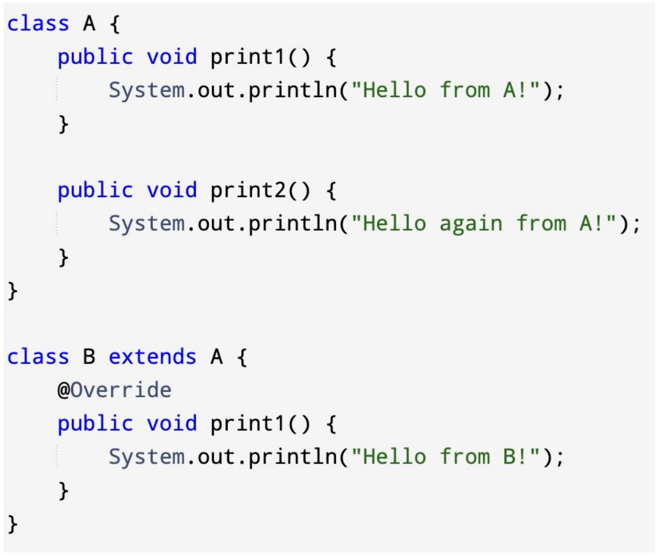
- **Remember!** If we have a class A which has a subclass B, instances of B are instances of A.
  - This also applies for inheritance that goes deeper down (*transitivity*). For example, if C was a subclass of B, then instances of C are also instances of A (and B, and C).
- Keeping this in mind, if we 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 instances of the same exact class, and false otherwise.

Method Overriding

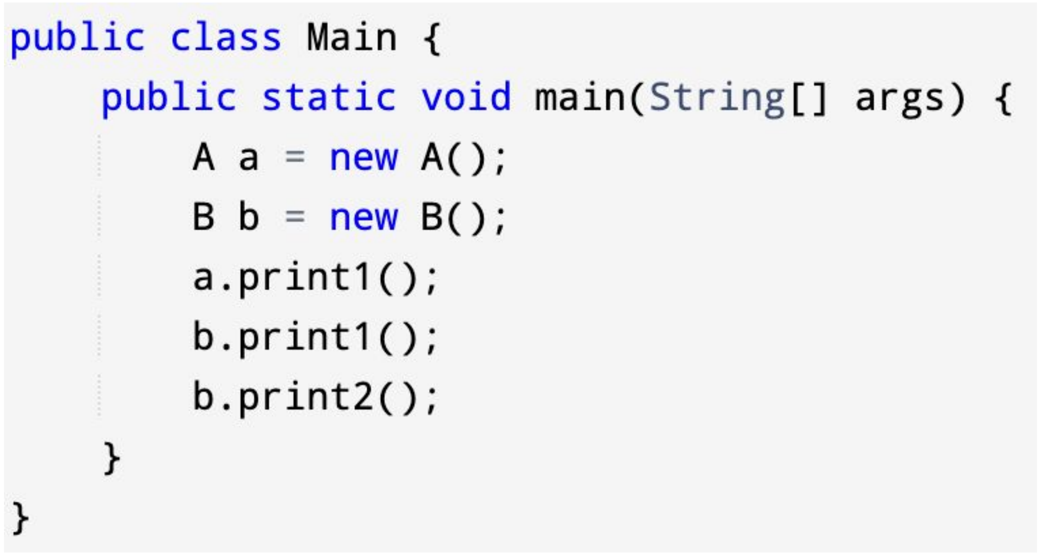
- **Important!** A subclass inherits **all** of its superclass' fields and methods. Private fields/methods cannot be accessed, but are technically still there.
    - If class A defines a (public) method `doSomething()` and class B extends A, then `doSomething()` can also be invoked from instances of class B.
- 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 (exact 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).

Quiz: Method Overriding

- What does the following code output?



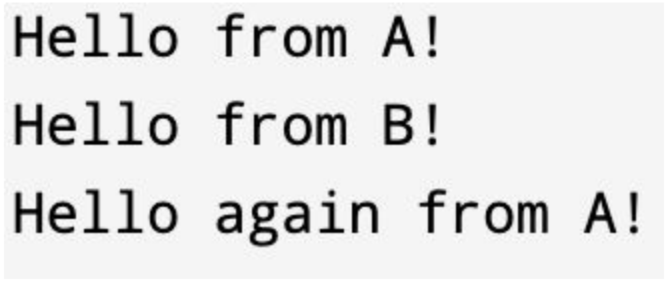
```java
class A {
    public void print1() {
        System.out.println("Hello from A!");
    }

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

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



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

Quiz: Method Overriding

- `a.print1()` prints “Hello from A!”, nothing new.
- `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. How can we reuse code from Employee's `toString()` while writing Manager's `toString()`?
- [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. How can we reuse code from Employee's `equals()` while writing Manager's `equals()`?
- [KEY TAKEAWAYS] Same as Pt. 1, type-checking, safe type-casting.

Abstract Classes

- An **abstract class** is a class that *cannot be instantiated*. They allow methods without concrete implementations to be declared, called abstract methods.
- They essentially act as templates to declare a common structure between any subclasses that derive off of it (remember both fields and methods are inherited down!).
    - This is useful for when you want to create a parent class capturing a bunch of common fields and methods, but this class doesn't make sense as a concrete object (e.g. Animal).
    - Concrete classes that inherit abstract methods must provide the concrete implementations for those methods (otherwise the code cannot compile).
    - Concrete implementations can still be defined within abstract classes.

Interfaces

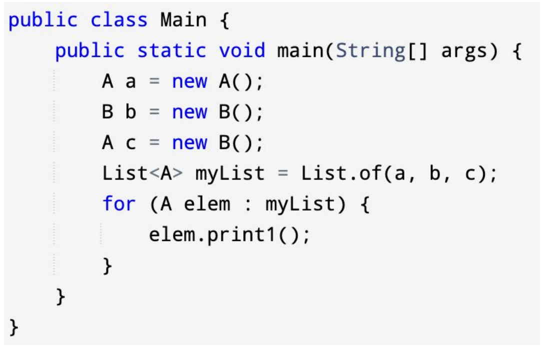
- An **interface** is similar to an abstract class, where all methods are abstract by default.
  - There is a way to define a concrete implementation for a method in an interface, but this is not used often, since interfaces cannot store instance fields (which most methods typically depend on for functionality).
- Interfaces are useful to define a common set of methods that every class that **implements** that interface must provide the concrete implementation for, which provides another way to group related classes.
- Each class in Java can have only one superclass, but can implement as many interfaces as it would like.

Polymorphism

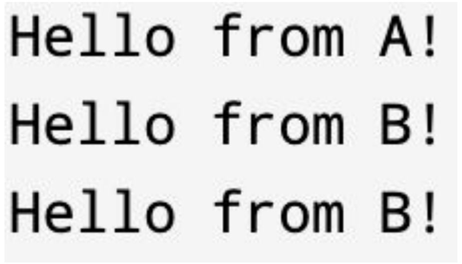
- **Polymorphism** is the ability to use a **common interface** across **different types/classes** to invoke certain functionality**,** regardless of how that functionality is implemented in each of the classes.
    - Here, ‘interfaces’ is a general term, not specifically the interfaces we just talked about. Abstract classes and interfaces provide very useful ways to apply polymorphism by ‘prescribing’ specific methods that should be invokable.
    - In other words, it is the ability to interact with different objects in the same exact way, despite any differences in how they choose to do things.
    - This captures the essence of well designed object-oriented code; polymorphism provides a simple, yet flexible way to interact with the objects of a system.

Code Example: Polymorphism

- Since the A class from earlier defines a method called `print1()`, we know that any objects of type A (or subclasses of A) must also have this method (overridden or not).
- If we store a list of objects of type A (which can also store subclasses of A), `print1()` must be common between all of these objects. Hence, we are guaranteed to be able to call it; the actual behaviour of the method differs across objects, depending on their actual class!



```java
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();
        }
    }
}
```



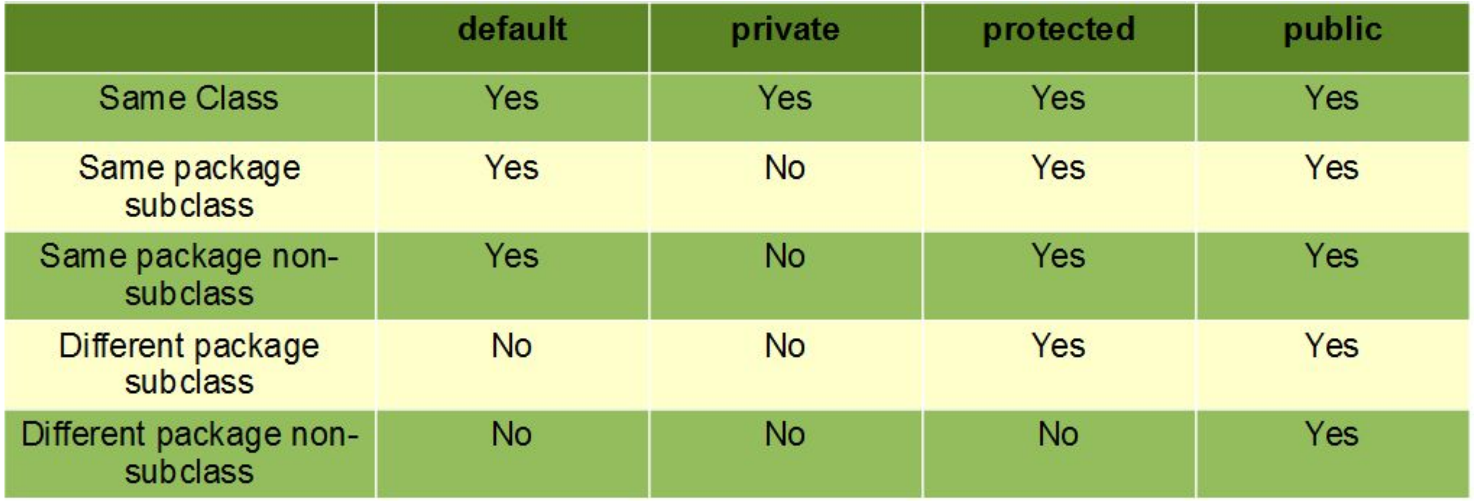
```
Hello from A!
Hello from B!
Hello from B!
```

Live Example: Polymorphism

- Look at the code in the src/languages package, which models multiple students learning different languages.
  - When does it make sense for a class to be abstract?
  - What is the difference between an abstract class and an interface? Why would you use one or the other?
  - Refactor the code to improve its quality.
- [KEY TAKEAWAYS] Interface and abstract class syntax, recognising where the use of either is applicable

Extra Material

Access Modifiers



	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 printed out by calling `Rectangle r = new Rectangle(“red”, 10, 20)`?
    - Is a call to `super(...)` necessary in the constructor of a subclass? If so, why?
    - What is the output of running `r2.getArea()` in main?
    - What is the output of running `Shape.getCount()` in main?

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.
    - It will print 400, as it uses the Square implementation of `getArea()`.
    - It will print 2 as each constructor call updates the static count, which is shared across instances. Hence, the first one updates 0 to 1, and the second updates 1 to 2. This is in contrast to non-static variables which have different values across instances - in that case, the count would be 1 for the two separate instances.