

# Inheritance and Polymorphism

## Inheritance in Java

*Inheritance* is an important feature of object-oriented programming in Java. It allows for one class (*child class*) to inherit the fields and methods of another class (*parent class*). For instance, we might want a child class `Dog` to inherit traits from a more general parent class `Animal`.

When defining a child class in Java, we use the keyword `extends` to inherit from a parent class.

```
// Parent Class
class Animal {

    // Animal class members

}

// Child Class
class Dog extends Animal {

    // Dog inherits traits from Animal

    // additional Dog class members

}
```

## Main() method in Java

In simple Java programs, you may work with just one class and one file. However, as your programs become more complex you will work with multiple classes, each of which requires its own file. Only one of these files in the Java package requires a `main()` method, and this is the file that will be run in the package. For example, say we have two files in our Java package for two different classes:

- `Shape` , the parent class.
- `Square` , the child class.

If the Java file containing our `Shape` class is the only one with a `main()` method, this is the file that will be run for our Java package.

```
// Shape.java file
class Shape {
    public static void main(String[] args)
    {
        Square sq = new Square();
    }
}

// Square.java file
class Square extends Shape {

}
```

## super() in Java

In Java, a child class inherits its parent's fields and methods, meaning it also inherits the parent's constructor. Sometimes we may want to modify the constructor, in which case we can use the `super()` method, which acts like the parent constructor inside the child class constructor.

Alternatively, we can also completely override a parent class constructor by writing a new constructor for the child class.

```
// Parent class
class Animal {
    String sound;

    Animal(String snd) {
        this.sound = snd;
    }
}

// Child class
class Dog extends Animal {

    // super() method can act like the
    // parent constructor inside the child
    // class constructor.
    Dog() {
        super("woof");
    }

    // alternatively, we can override the
    // constructor completely by defining a new
    // constructor.
    Dog() {
        this.sound = "woof";
    }
}
```

## Protected and Final keywords in Java

When creating classes in Java, sometimes we may want to control child class access to parent class members. We can use the `protected` and `final` keywords to do just that.

`protected` keeps a parent class member accessible to its child classes, to files within its own package, and by subclasses of this class in another package.

Adding `final` before a parent class method's access modifier makes it so that any child classes cannot modify that method - it is immutable.

```
class Student {

    protected double gpa;
    // any child class of Student can
    // access gpa

    final protected boolean isStudent() {

        return true;
    }

    // any child class of Student cannot
    // modify isStudent()
}
```

## Polymorphism in Java

Java incorporates the object-oriented programming principle of *polymorphism*.

Polymorphism allows a child class to share the information and behavior of its parent class while also incorporating its own functionality. This allows for the benefits of simplified syntax and reduced cognitive overload for developers.

```
// Parent class
class Animal {
    public void greeting() {
        System.out.println("The animal
greet you.");
    }
}

// Child class
class Cat extends Animal {
    public void greeting() {
        System.out.println("The cat
meows.");
    }
}

class MainClass {
    public static void main(String[] args)
{
    Animal animal1 = new Animal(); //
Animal object
    Animal cat1 = new Cat(); // Cat
object
    animal1.greeting(); // prints "The
animal greet you."
    cat1.greeting(); // prints "The cat
meows."
    }
}
```

In Java, we can easily *override* parent class methods in a child class. Overriding a method is useful when we want our child class method to have the same name as a parent class method but behave a bit differently. In order to override a parent class method in a child class, we need to make sure that the child class method has the following in common with its parent class method:

- Method name
- Return type
- Number and type of parameters

Additionally, we should include the `@Override` keyword above our child class method to indicate to the compiler that we want to override a method in the parent class.

## Child Classes in Arrays and ArrayLists

In Java, polymorphism allows us to put instances of different classes that share a parent class together in an array or `ArrayList`.

For example, if we have an `Animal` parent class with child classes `Cat`, `Dog`, and `Pig` we can set up an array with instances of each animal and then iterate through the list of animals to perform the same action on each.

```
// Parent class
class Animal {
    public void eating() {
        System.out.println("The animal
is eating.");
    }
}

// Child class
class Dog extends Animal {
    // Dog's eating method overrides
    Animal's eating method
    @Override
    public void eating() {
        System.out.println("The dog is
eating.");
    }
}
```

```
// Animal parent class with child
classes Cat, Dog, and Pig.
Animal cat1, dog1, pig1;

cat1 = new Cat();
dog1 = new Dog();
pig1 = new Pig();

// Set up an array with instances of
each animal
Animal[] animals = {cat1, dog1, pig1};

// Iterate through the list of animals
and perform the same action with each
for (Animal animal : animals) {

    animal.sound();

}
```