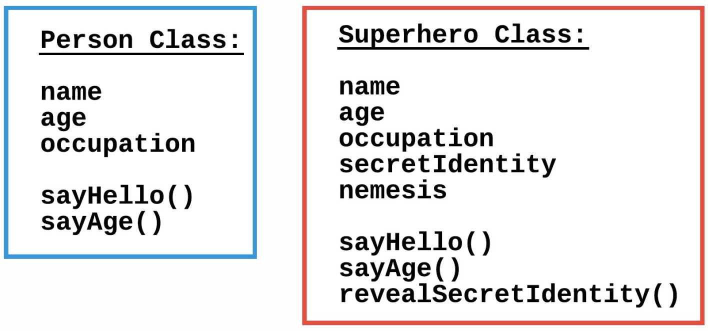
Learning Objectives - Superclass & Subclass

- **Define the terms inheritance, superclass, and subclass**
- **Explain the relationship between the superclass and subclass**
- **Explain the role of the `super` keyword**
- **Create a subclass class from a given superclass**
- **Use `instanceof` to determine an object’s parent class**
- **Define the substitution principle**

What is Inheritance?

Defining Inheritance

Imagine you want to create two Java classes, `Person` and `Superhero`. These respective classes might look something like this:



No_Inheritance

There are some similarities between the `Person` class and the `Superhero` class. If the `Person` class already exists, it would be helpful to “borrow” from the `Person` class so you only have to create the new attributes and methods for the`Superhero` class. This situation describes inheritance — one class copies the attributes and methods from another class.

Inheritance Syntax

In the IDE on the left, the `Person` class is already defined. To create the `Superhero` class that inherits from the `Person` class, add the following code at the end of the program. Notice how `Superhero` definition adds `extends Person`. This is how you indicate to Java that the `Superhero` class inherits from the `Person` class. You can also say that `Person` is the superclass and `Superhero` is the subclass.

```java
//add class definitions below this line

class Superhero extends Person {

}

//add class definitions above this line
```

Now declare an instance of the `Superhero` class and print the value of the `name` and `age` attributes using their getter methods.

```
    //add code below this line

    Superhero s = new Superhero();
    System.out.println(s.getName());
    System.out.println(s.getAge());

    //add code above this line
```

▼ **Why does the program print `null` and `0`?**

The `Person` class does not have a constructor. So when Java creates the attributes, it gives `null` as the initial value for strings and `0` for the initial value of integers.

challenge

Try these variations:

- Print the `occupation` attribute.
- Call the `sayHello` method from the `Superhero` object.

▼ **Solution**

```
        //add code below this line

        Superhero s = new Superhero();
        System.out.println(s.getName());
        System.out.println(s.getAge());
        System.out.println(s.getOccupation());
        s.sayHello();

        //add code above this line
```

Limitations of Inheritance

Java place some rules about how inheritance works. Most importantly, only public methods and attributes in superclass are directly accessible to the subclass. The following code creates an error. While the subclass inherits the `name` attribute, it can only access it through the getter method. For inheritance to work properly, the superclass needs to have getters, setters, and other public methods

```
  //add code below this line

  Superhero s = new Superhero();
  System.out.println(s.name);

  //add code above this line
```

Constructors can also be a bit tricky with inheritance. Java will automatically call the constructor of the superclass when the subclass object is instantiated. The superclass should have a constructor without any parameters. Add the following constructor to the `Person` class. This will avoid the situation above where attributes have values like `null` and `0`.

```
public Person() {
  name = "Sarah";
  age = 37;
  occupation = "VP Sales";
}
```

Now call the `sayHello` and `sayAge` methods from the `Superhero` object.

```
  //add code below this line

  Superhero s = new Superhero();
  s.sayHello();
  s.sayAge();

  //add code above this line
```

challenge

Try these variations:

- Change the constructor of the `Person` class so that the default values are `Rodrigo`, `19`, `student`.

▼ Solution

```
public Person() {
  name = "Rodrigo";
  age = 19;
  occupation = "student";
}
```

- Change the constructor of the `Person` class to look like the following code:

```
public Person(String n, int a, String o) {
  name = n;
  age = a;
  occupation = o;
}
```

▼ Why is there an error?

Java tries to call the constructor of the superclass when instantiating a subclass object. This happens automatically and without any parameters, but the `Person` class expects three parameters. This is why the code generates an error. The next page describes how to call a superclass constructor with parameters.

Super

The super Keyword

The super keyword is used in the subclass to invoke methods in the superclass. This is how constructors with parameters are called in inheritance. The Person class to the left has a constructor without any parameters. Add a second constructor to the Person class that has three parameters.

```
  public Person() {
    name = "Sarah";
    age = 37;
    occupation = "VP Sales";
  }

  public Person(String n, int a, String o) {
    name = n;
    age = a;
    occupation = o;
  }
```

By default, Java will call the constructor with no parameters when the Superhero object is instantiated. Create a constructor for the Superhero class with three parameters. Use the super keyword followed by the parameters. Java will pass the parameters to the constructor for Person because that is the superclass. You are now able to specify the name, age, and occupation attributes for the subclass.

```
//add class definitions below this line

class Superhero extends Person {
  public Superhero(String n, int a, String o) {
    super(n, a, o);
  }
}

//add class definitions above this line
```

Instantiate a `Superhero` object with a name, age, and occupation. Call the `sayName` and `sayAge` methods to verify that the `Superhero` object has the values `Wonder Woman` and `27` for the `name` and `age` attributes.

```
    //add code below this line

    Superhero hero = new Superhero("Wonder Woman", 27,
        "intelligence officer");
    hero.sayHello();
    hero.sayAge();

    //add code above this line
```

challenge

Try this variation:

Add the following constructor to the `Person` class.

```
public Person(String n) {
  name = n;
  age = 0;
  occupation = "";
}
```

Create a constructor for the `Superhero` class that will call this new constructor in the superclass.

▼ **Solution**

```
  public Superhero(String n) {
    super(n);
  }
```

You can call this new constructor with the following object instantiation:

```
    //add code below this line

    Superhero hero = new Superhero("Wonder Woman");
    hero.sayHello();

    //add code above this line
```

Every Class has a Superclass

The `Superhero` class has a superclass because it extends the `Person` class. But what about the `Person` class, does it have a superclass? In Java, every class has a superclass. The code below creates `Person` object. The variable `c` represents the class of the `Person` object, and the variable `sc` represents the superclass of the `Person` object.

```
    //add code below this line

    Person p = new Person();
    Class c = p.getClass();
    Class sc = c.getSuperclass();
    System.out.println("Class: " + c);
    System.out.println("Superclass: " + sc);

    //add code above this line
```

The second line of output should be `class java.lang.Object`. That means the `Object` class is the superclass for every object in Java. However, you do not need to declare that a user-defined class extends the `Object` class. Java assumes this automatically.

Inheritance Hierarchy

Inheritance Hierarchy

You have seen how the `Superhero` class becomes the subclass of the `Person` class through inheritance. The relationship between these two classes is called inheritance (or class) hierarchy. Java has a built-in operator to help you determine the hierarchy of classes.

Start by creating four classes, where `ClassA` is the superclass to `ClassB` and `ClassC` is the superclass to `ClassD`. These classes do not need to do anything, so do not create attributes, a constructor, or methods.

```java
//add class definitions below this line

class ClassA {}
class ClassB extends ClassA {}
class ClassC {}
class ClassD extends ClassC {}

//add class definitions above this line
```

Now, create an instance for each class.

```java
    //add code below this line

    ClassA a = new ClassA();
    ClassB b = new ClassB();
    ClassC c = new ClassC();
    ClassD d = new ClassD();

    //add code above this line
```

The `instanceof` operator returns a boolean when comparing an object and a class. It returns `true` if the object is an instance of a class, and it returns `false` if the object is not an instance of a class. The code below prints the class for object `b` which is `ClassB`. It then prints `true` because object `b` is an instance of `ClassB`. Surprisingly, it also prints `true` when asking if `b` is an instance of `ClassA`. This is because of inheritance. Because `ClassB` inherits from `ClassA`, `b` is considered to be an instance of `ClassA`.

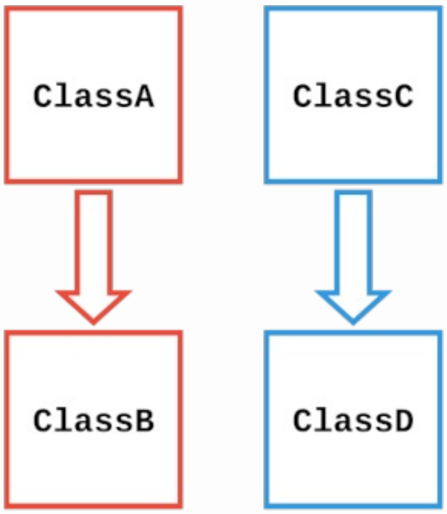
```
    //add code below this line

    ClassA a = new ClassA();
    ClassB b = new ClassB();
    ClassC c = new ClassC();
    ClassD d = new ClassD();

    System.out.println(b.getClass());
    System.out.println(b instanceof ClassB);
    System.out.println(b instanceof ClassA);

    //add code above this line
```

Think of inheritance hierarchy as a downward flow chart. The superclass is on top while the subclass is below. The hierarchy is a one-way relationship. Inheritance always flows from the superclass to the subclass.



.guides/img/inheritance/inheritance_herarchy

challenge

Try these variations:

Add the following code to your program.
* `System.out.println(a instanceof ClassB);`
* `System.out.println(d instanceof ClassC);`

Substitution Principle

Substitution Principle

When one class inherits from another, Java considers them to be related. They may have different data types, Java allows a subclass to be used in place of the superclass. This is called the substitution principle. In the code below, `ClassB` inherits from `ClassA`. Both classes have the `greeting` method which prints a greeting specific to the class.

```
//add class definitions below this line

class ClassA {
  public void greeting() {
    System.out.println("Hello from Class A");
  }
}

class ClassB extends ClassA {
  public void greeting() {
    System.out.println("Hello from Class B");
  }
}

//add class definitions above this line
```

According to the substitution principle, an object of `ClassB` can be used in a situation that expects an object of `ClassA`. The `substitution` method explicitly calls for an argument of `ClassA`.

```
  //add method definitions below this line

  public static void substitution(ClassA a) {
    a.greeting();
  }

  //add method definitions above this line
```

Instantiate an object of `ClassB` and pass it to the `substitution` method. Even though the object `b` has the wrong data type, the code should still work due to the substitution principle. Because `ClassB` extends `ClassA`, object b can be used in place of an object of type ClassA. Run the code to verify the output.

```
//add code below this line

ClassB b = new ClassB();
substitution(b);

//add code above this line
```

challenge

Try this variation:

- Remove extends ClassA from the ClassB declaration and run the code again.

  ▼ **Why did this produce an error?**

  Deleting extends ClassA means that ClassB no longer inherits from ClassA. Therefore, the substitution principle no longer applies. Java now says there is a type mismatch error.

The Substitution Principle is a One-Way Relationship

Like inheritance hierarchy, the substitution principle is a one-way relationship. Add a third class ClassC that inherits from ClassB. We now have an inheritance chain that flows from ClassA to ClassB to ClassC.

```java
//add class definitions below this line

class ClassA {
  public void greeting() {
    System.out.println("Hello from Class A");
  }
}

class ClassB extends ClassA {
  public void greeting() {
    System.out.println("Hello from Class B");
  }
}

class ClassC extends ClassB {
  public void greeting() {
    System.out.println("Hello from Class C");
  }
}

//add class definitions above this line
```

The substitution principle states that the substitution will work with any subclass. Since `ClassB` is a subclass of `ClassA` and `ClassC` is a subclass of `ClassA`, then `ClassC` can be substituted for `ClassA`. Create an object of type `ClassC` and pass it to the `substitution` method.

```java
    //add code below this line

    ClassC c = new ClassC();
    substitution(c);

    //add code above this line
```

Change the method header for `substitution` so that the parameter is of type `ClassB`.

```java
  //add method definitions below this line

  public static void substitution(ClassB b) {
    b.greeting();
  }

  //add method definitions above this line
```

Now create an object of type `ClassA` and pass it to the `substitution` method. This should now cause an error because the substitution principle no longer applies because the `substitution` expects `ClassB` or a subclass. `ClassA` is a superclass, so it cannot be substituted for `ClassB`.

```java
    //add code below this line

    ClassA a = new ClassA();
    substitution(a);

    //add code above this line
```