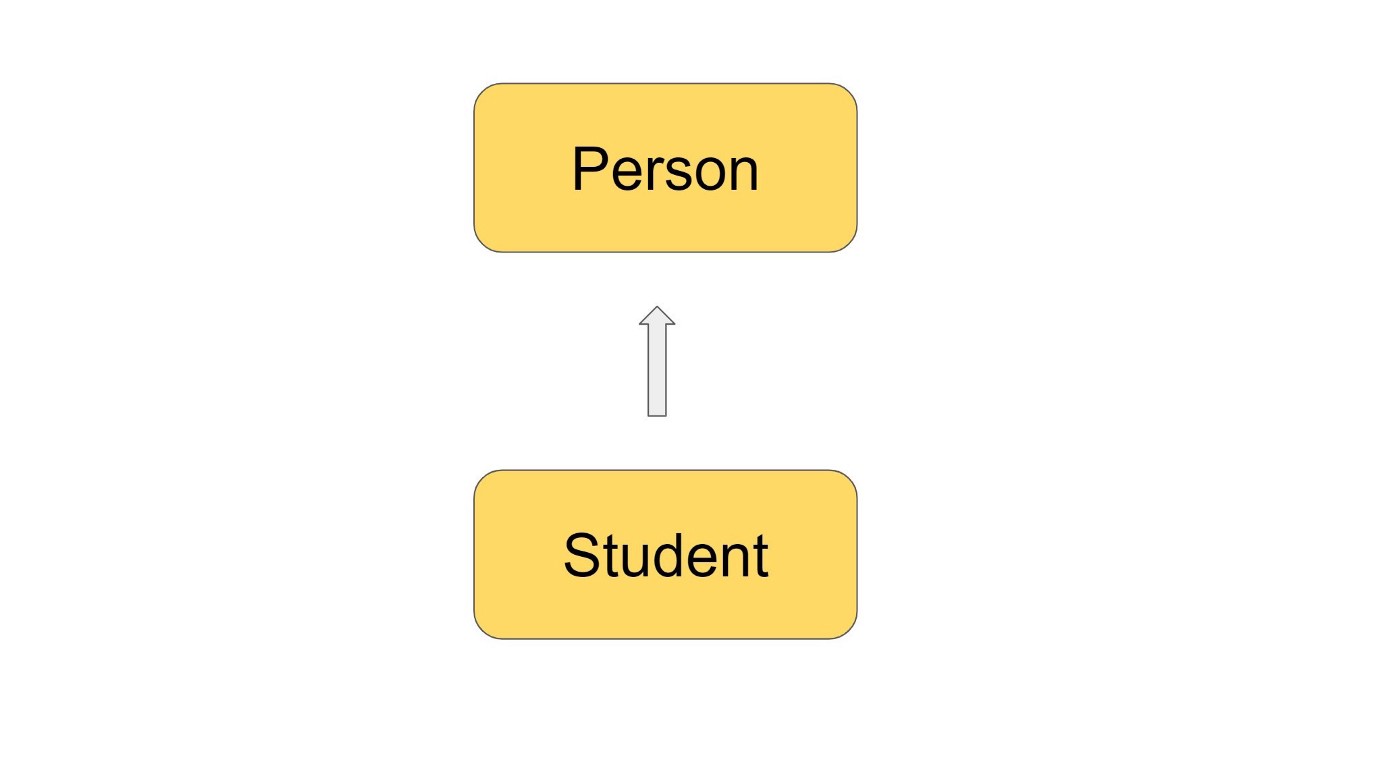
**OOP: Inheritance in Java**

1. Inheritance.
2. Method overriding and variables hiding.
3. this and super keywords.

**Inheritance**

Inheritanceis a process where one class can inherit visible properties and methods from another class — the parent-child relationship between two classes (or superclass and subclass).

// in Person.java file  
public class Person {  
 public String name;  
 public String address;  
 public int age;  
   
 public void walk() {  
 System.out.println(name + " is walking.");  
 }  
}// in Student.java file  
public class Student extends Person {  
 public static void main(String[] args){  
 Student student = new Student();  
 student.name = "John Doe";  
 student.address = "101 Main St";  
 student.age = 22;  
 student.walk();  
 }  
}



Student extends Person

In the above example, the Student class extends the Person class so our Student class is a child class of Person class. Student class will extend all visible (depends on access modifiers of variables and methods) variables and methods.

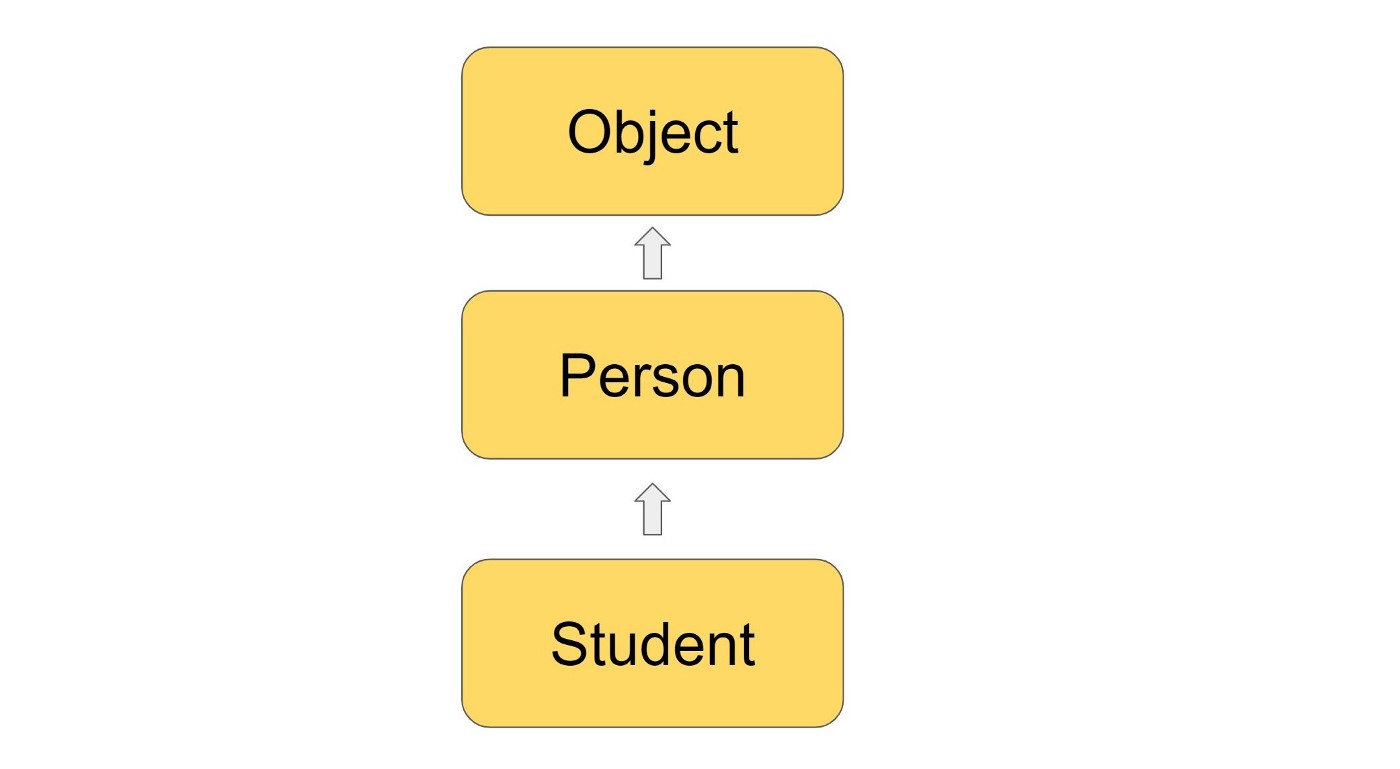
Inheritance is useful for code reusability for example we can have one generic class that will have common properties and methods with default behaviors and the child classes can just extend it and reuse a lot of code. If the child class wants to have its own implementation for methods defined in the parent class, we can always override these methods in the child class.

One good example is *java.lang.Object* class. Object class is the parent class for all classes in java. Java automatically will inject extends Object syntax after every class declaration. Why every class needs to extend from super java.lang.Object class? So from every class in java, we can potentially create an object. It can be Person or it can be Student or it can be Car and so on and if we think about these classes they are all objects. Java wants to give generic behaviors for every object that ever will be created in java. The [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html" \t "_blank)has 11 methods(Java 8) so every class will inherit these methods.

[**equals**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) obj)

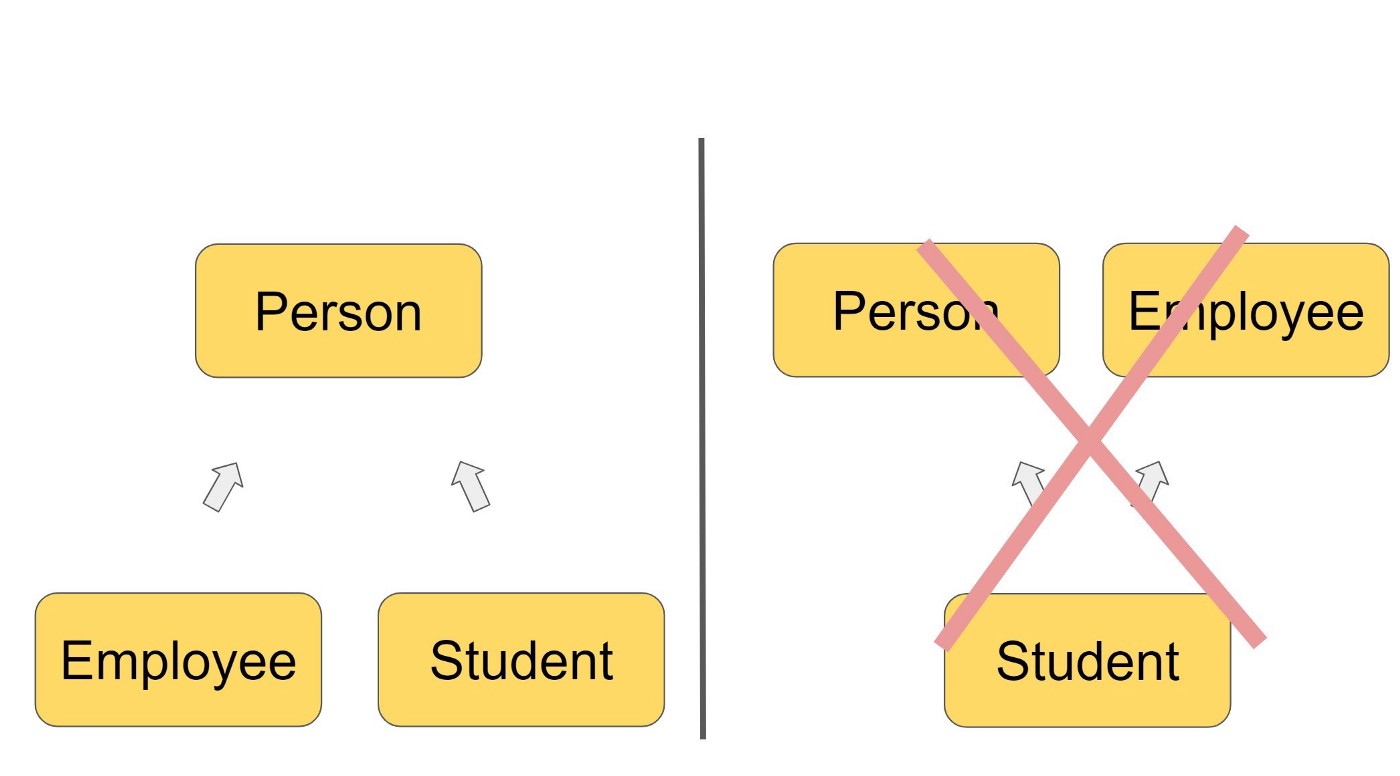
This is one of the methods that will come from the Object class. We need the equals method to compare two objects of the same class on equality. So our superclass is giving as equals method to do so. By default, it will not compare two object properties, it will compare if two references are pointing to the same object or not(same as ==). We need to override the equals method and write the logic of how exactly we want to compare our objects.

It’s good to have some common methods for all objects because other libraries can assume that in order to compare your objects they can use the equals method. The same logic for other methods as well.



In our case Object is grandpa for Student class :)

We are saying that every class extends Object but in this example, our Student class extends our Person class, not the Object class. Yes, the Student class will extend the Object class via the Person class.



Java allows only a single inheritance type

Java allows only a single inheritance type. Multiple classes can inherit from a single class but one class cannot inherit multiple classes at the same time.

**Method overriding and variables hiding**

When a child class wants to use its own implementation of the method instead of the parent class method, it can always override it.

// in Car.java file  
public class Car {  
 public void drive() {  
 System.out.println("Car is driving");  
 }  
}

// in BMW.java file  
public class BMW extends Car {  
 @Override  
 public void drive() {  
 System.out.println("BMW is driving");  
 }

public static void main(String[] args) {  
 BMW bmw = new BMW();  
 bmw.drive(); // BMW is driving  
 }  
}

* in the above example, the child class BMW overrides the method of a parent class.
* @Override annotation will check if we are really overriding the method. If it finds some issues, it gives a compiler error.

Rules of overriding:

* Method name and number, order, and type of arguments should be exactly the same as the parent’s method.
* Return type should be the same or [covariant](https://en.wikipedia.org/wiki/Covariant_return_type) with the parent method.
* The access modifier should be the same or more visible than the parent method.
* If exception declaration exists in the parent method, the child method can have the same type of exception declaration or a smaller type.

Why some people refer to overriding as runtime polymorphism?

Car car = new BMW();  
car.drive(); // BMW is driving

In the above example, we have Car as our reference type and it will decide what method and variables are available for this reference. The actual object of car is BMW . During the compilation, java thinks that it will call drive() method from Car class, but during the runtime, it figures out that BMW actually overrides the method and it will call the method from BMW

Now, for variables, if child class will have variables with exact same name as parent class does. Child class variables will hide the parent class variables.

Hiding is different than overriding, there is no runtime polymorphism.

// in Car.java file  
public class Car {  
 public String name = "CAR";  
}

// in BMW.java file  
public class BMW extends Car {  
 public String name = "BMW";

public static void main(String[] args) {  
 BMW bmw = new BMW();  
 System.*out*.println(bmw.name); // BMW  
  
 Car car = new BMW();  
 System.*out*.println(car.name); // CAR  
 }  
}

**this and super keywords**

this keyword is used to refer to the current object of a class.

super keyword is used to refer to the parent class members.

// in Car.java file  
public class Car {  
 public String name = "CAR";  
}

// in BMW.java file  
public class BMW extends Car {  
 public String name = "BMW";

public void printName() {  
 System.*out*.println(this.name); // BMW  
 System.*out*.println(super.name); // CAR  
 }

public static void main(String[] args) {  
 BMW bmw = new BMW();  
 bmw.printName();

}  
}

* As you can see in the printName() method we are using super to get the name of a parent class.
* It works similarly with methods.

To call a constructor we can also use this and for parent class constructor super . Let’s see examples of calling empty constructors.

this(); // calls empty constructor of current class  
super(); // calls empty constructor of parent class