NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 4: 20th Feb 2024

Static vs Instance Variables

Static Variables

- Declared by using the keyword **static**.
- Static variables are also known as **static member variables**.
- Every instance of the class shares the **same** static variable.
- So if changes are made to that variable, all other instances of that class will see the effect of that change.

Static Variables

It is considered best practice to use the Class name, and not a reference variable to access a static variable.

```
class Dog {

    static String genus = "Canis";

    void printData() {

        Dog d = new Dog();
        System.out.println(d.genus);      // Confusing!
        System.out.println(Dog.genus);    // Clearer!
    }
}
```

Static Variables

An instance isn't required to exist, to access the value of a static variable.

```
class Dog {

    static String genus= "Canis";
}

class Main {

    public static void main(String[] args) {
        System.out.println(Dog.genus);        // No instance of Dog needs to exist, in order to access a static variable
    }
}
```

Static Variables

Static variables aren't used very often, but can sometimes be very useful.

They can be used for

- Storing counters.
- Generating unique ids.
- Storing a constant value that doesn't change, like PI for example.
- Creating, and controlling access, to a shared resource.

Static Variables

```java
class Dog {

    private static String name;

    public Dog(String name) {
        Dog.name = name;
    }

    public void printName() {
        System.out.println("name = " + name);  // Using Dog.name would have made this code less confusing.
    }
}

public class Main {

    public static void main(String[] args) {

        Dog rex = new Dog("rex");                // create instance (rex)
        Dog fluffy = new Dog("fluffy");          // create instance (fluffy)
        rex.printName();                         // prints fluffy
        fluffy.printName();                      // prints fluffy
    }
}
```

Instance Variables

They don't use the static keyword.

They're also known as **fields**, or **member variables**.

Instance variables belong to a specific instance of a class.

Instance Variables

Each instance has its own copy of an instance variable.

Every instance can have a different value.

Instance variables represent the state, of a specific instance of a class.

Instance Variables

```
class Dog {

    private String name;

    public Dog(String name) {
        this.name = name;
    }

    public void printName() {
        System.out.println("name = " + name);
    }
}

public class Main {

    public static void main(String[] args) {

        Dog rex = new Dog("rex");                  // create instance (rex)
        Dog fluffy = new Dog("fluffy");            // create instance (fluffy)
        rex.printName();                           // prints rex
        fluffy.printName();                        // prints fluffy
    }
}
```

Static vs Instance Methods

Static Methods

Static methods are declared using a static modifier.

Static methods can't access instance methods and instant variables directly.

They're usually used for operations that don't require any data from an instance of the class (from '**this**').

If you remember, the **this** keyword is the current instance of a class.

Static Methods

So inside a static method, **we can't use the this** keyword.

Whenever you see **a method that doesn't use instance variables**, that method should probably be declared as a static method.

For example, main is a static method, and it's called by the Java virtual machine when it starts the Java application.

Static Methods Example

```
class Calculator {

    public static void printSum(int a, int b) {
        System.out.println("sum= " + (a + b));
    }
}

public class Main {

    public static void main(String[] args) {
        Calculator.printSum(5, 10);
        printHello();    // shorter from of Main.printHello();
    }

    public static void printHello() {
        System.out.println("Hello");
    }
}
```

static methods are called as **ClassName.methodName();** or **methodName();** only if in the same class

In this example
Calculator.printSum(5,10);
printHello();

Instance Methods

Instance methods **belong to an instance**, of a class.

To use an instance method, we have to instantiate the class first, usually by using the new keyword.

Instance Methods

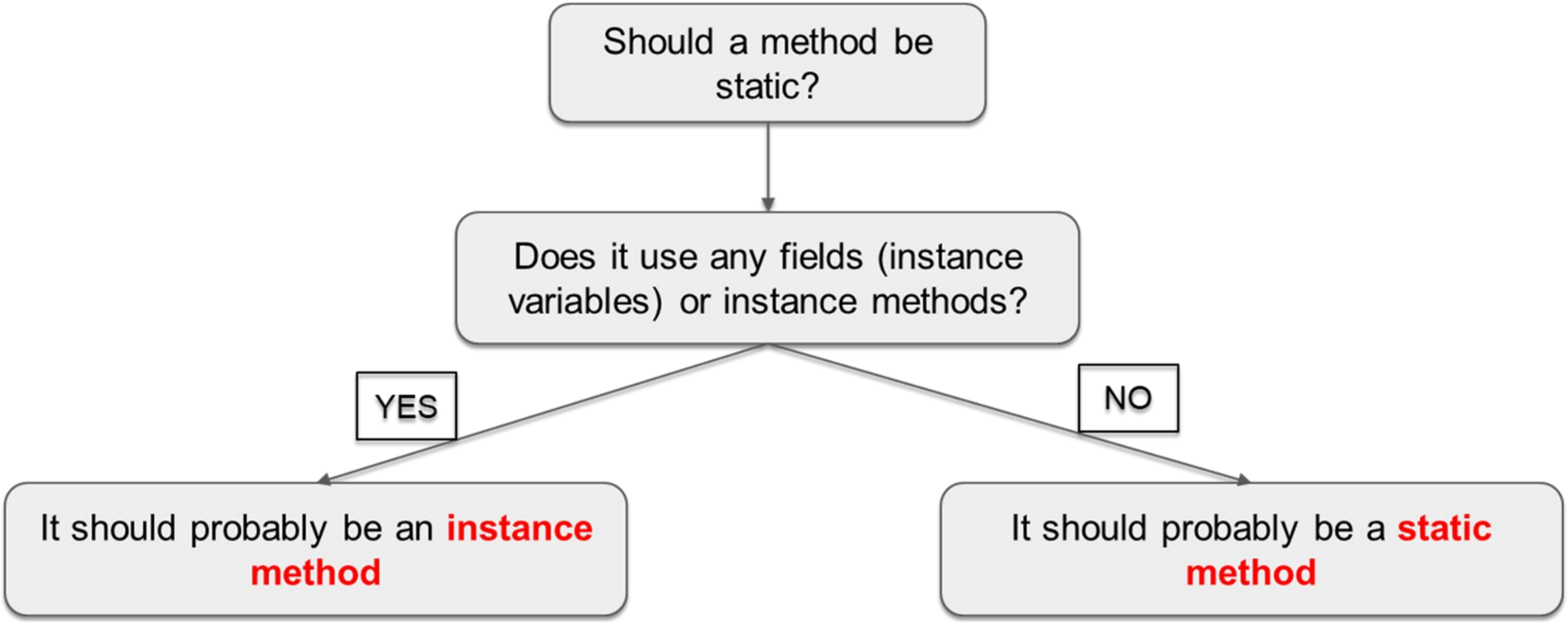
Instance methods can access **instance methods and instance variables** directly.

Instance methods can also access **static methods and static variables** directly.

Instance Method Example

```java
class Dog {

    public void bark() {
        System.out.println("woof");
    }
}

public class Main {

    public static void main(String[] args) {

        Dog rex = new Dog();                            // create instance
        rex.bark();                                     // call instance method
    }
}
```

Static or Instance Method



Should a method be static?
Does it use any fields (instance variables) or instance methods?
YES
NO
It should probably be an **instance method**
It should probably be a **static method**

Plain Old Java Object (POJO)

A plain old Java object (whose acronym is **POJO**) is a class that generally only has instance fields.

It's used to house data, and pass data, between functional classes.

It usually has few, if any methods other than getters and setters.

Many database frameworks use POJO's to read data from, or to write data to, databases, files or streams.

Examples of POJOS

A POJO also might be called a **bean**, or a **JavaBean**.

A JavaBean is just a POJO, with some extra rules applied to it.

A POJO is sometimes called an **Entity**, because it mirrors database entities.

Another acronym is **DTO**, for **Data Transfer Object**.

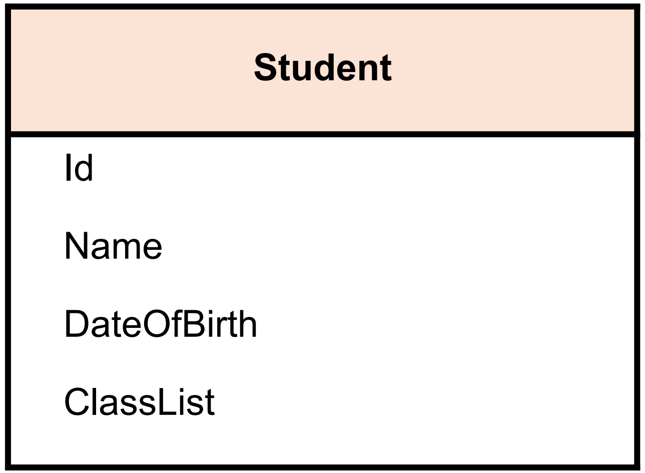
It's a description of an object, that can be modeled as just data.

Support for POJO creation

There are many generation tools, that will turn a data model into generated POJO's or JavaBeans.

You've seen an example of similar code generation in IntelliJ, which allowed us to generate getters, setters, and constructors in a uniform way.

The Entity - The Student Table



Annotation

Annotations are a **type of metadata**.

Metadata is a way of formally describing **additional information** about our code.

Annotations are more structured, and have more meaning, than comments.

This is because they can be used by the compiler, or other types of pre-processing functions, to get information about the code.

Metadata doesn't effect how the code runs, so this code will still run, with or without the annotation.

Overridden Method

An overridden method, is not the same thing as an overloaded method.

An overridden method is a special method in Java, that other classes can implement, if they use a specified method signature.

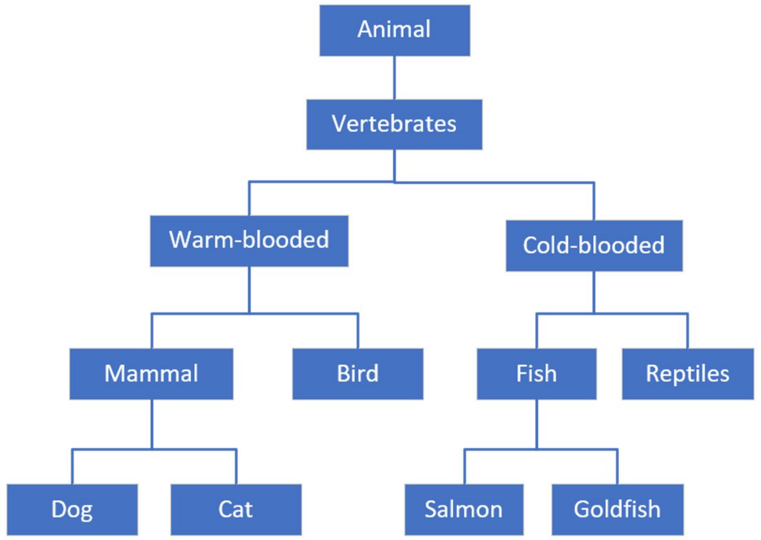
Inheritance

Inheritance

We can look at Inheritance as a form of code re-use.

It's a way to organize classes into a parent-child hierarchy, which lets the child inherit (re-use), fields and methods from its parent.

Inheritance



Each box on this diagram represents a Class.

The most generic, or base class, starts at the top of the hierarchy.

Every class below it is a subclass.

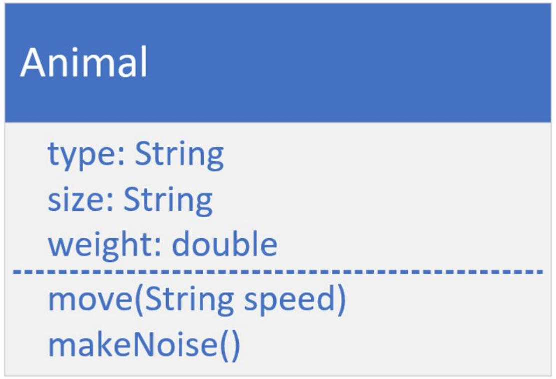
So Animal is the base class.   All the other classes can be said to be subclasses of Animal.

A parent can have multiple children, as we see with Mammal, which is the parent of Dog and Cat.

A child can only have one direct parent, in Java

But it will inherit from its parent class's parent, and so on.

The Animal class

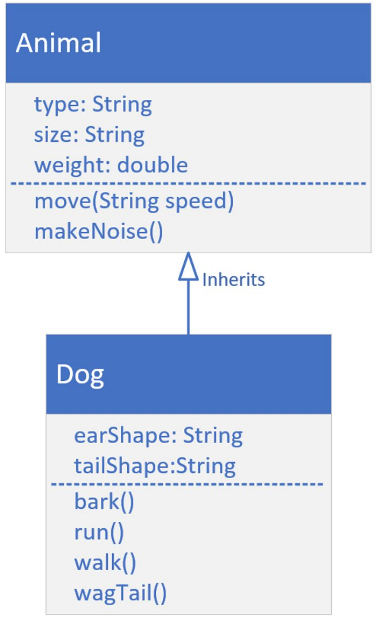


A class diagram, allows us to design our classes before we build them.

This diagram shows the Animal class, with the attributes we think that every kind of animal has.

Below the fields, we have the behavior that animals have in common, move, and makeNoise.

Class Model for Animal and Dog



Dog inherits from Animal.

In other words, Dog '**IS A**' type of Animal.

When we create a Dog object, it will inherit Animal's attributes (type, size and weight).

This is also true for Animal's methods. Dog will inherit these as well.

We can specialize the Dog class with its own fields and behavior.

extends

Using **extends** specifies the superclass (or the parent class) of the class we're declaring.

We can say Dog is a subclass, or child class, of Animal.

We can say Animal is a parent, or super class, of Dog.

A class can specify one, and only one, class in its extends clause.

super()

super() is a lot like this().

It's a way to call a constructor on the super class, directly from the sub class's constructor.

Like this(), it has to be the first statement of the constructor.

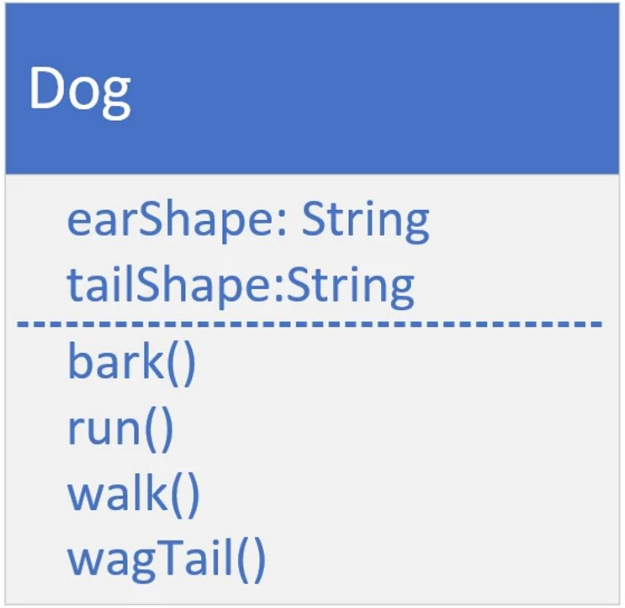
Because of that rule, this() and super() can **never be called from the same constructor.**

super()

If you don't make a call to super(), **then Java makes it for you**, using super's default constructor.

If your super class doesn't have a default constructor, than you must explicitly call super() in all of your constructors, passing the right arguments to that constructor.

The Dog Class



Code Re-use

All subclasses can execute methods, even though the code is declared on the parent class.

The code doesn't have to be duplicated in each subclass.

We can use code, from the parent.

Or we can change that code for the subclass.

Overriding a method

Overriding a method is when you create a method on a subclass, which has the same signature as a method on a super class.

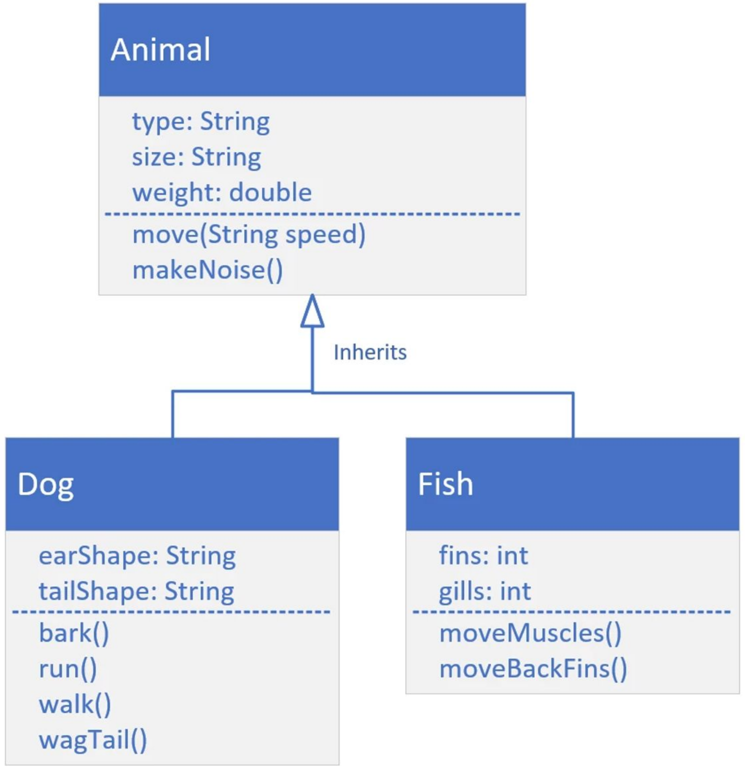
You override a parent class method, when you want the child class to show different behavior for that method.

Overridden method

The overridden method can do one of three things:

- It can implement completely different behavior, overriding the behavior of the parent.
- It can simply call the parent class's method, which is somewhat redundant to do.
- Or the method can call the parent class's method, and include other code to run, so it can extend the functionality for the Dog, for that behavior.

Class Diagram with additional class, Fish



Polymorphism

Polymorphism simply means 'many forms’.

And as you've seen, some advantages of Polymorphism are:

- It makes code simpler.
- It encourages code extensibility.

Inheritance

Inheritance looks kind of interesting, but when would we really use it?

Well, it turns out, in Java, we've been using Inheritance all along.

java.lang.Object

This is because every class you create in Java, intrinsically extends a special Java class.

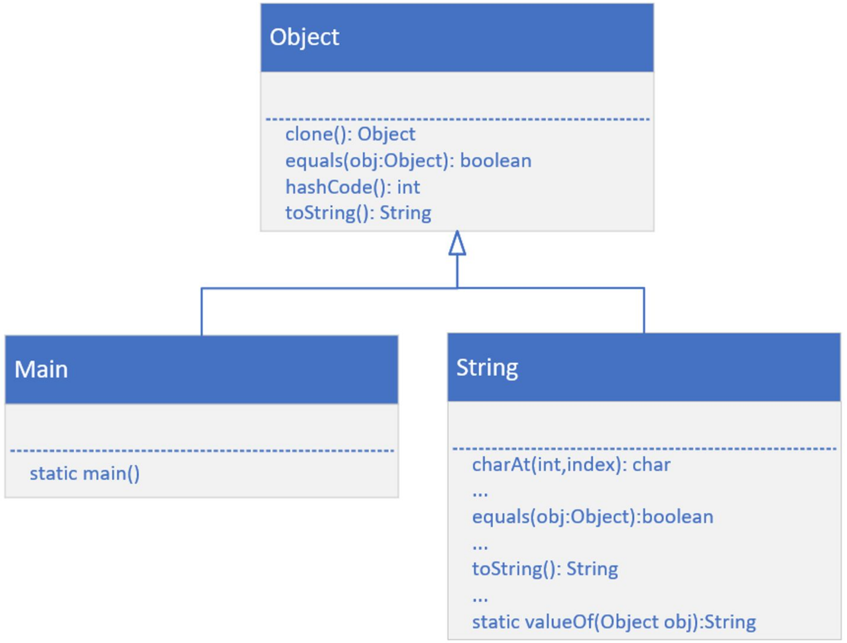
This class is named Object, and it's in the java.lang package.

Ok, that's confusing, a class called Object?

Let's see what Java has to say about this class: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html

We'll use the link to Java's Application Programming Interface (API).

Every Class inherits from Object

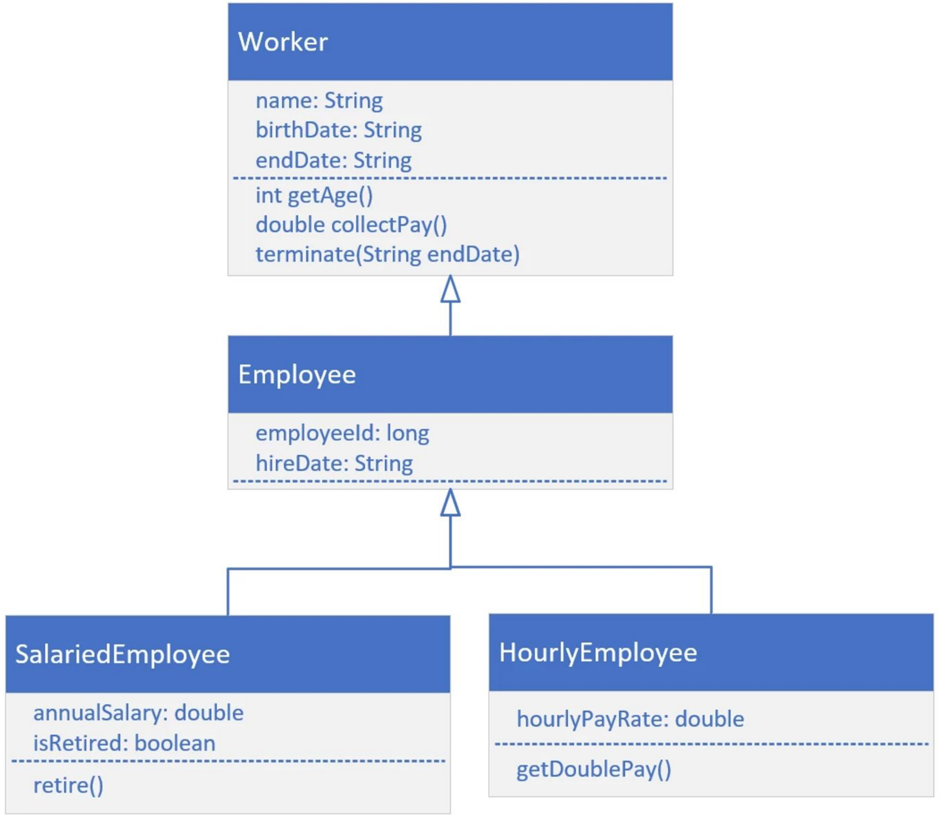


This slide shows that our Main class inherits from, or is a subclass of Object, as also is String.

The String class has over 60 methods!

The String class overrides several methods on Object, two of which are equals(), and toString().

Inheritance Challenge

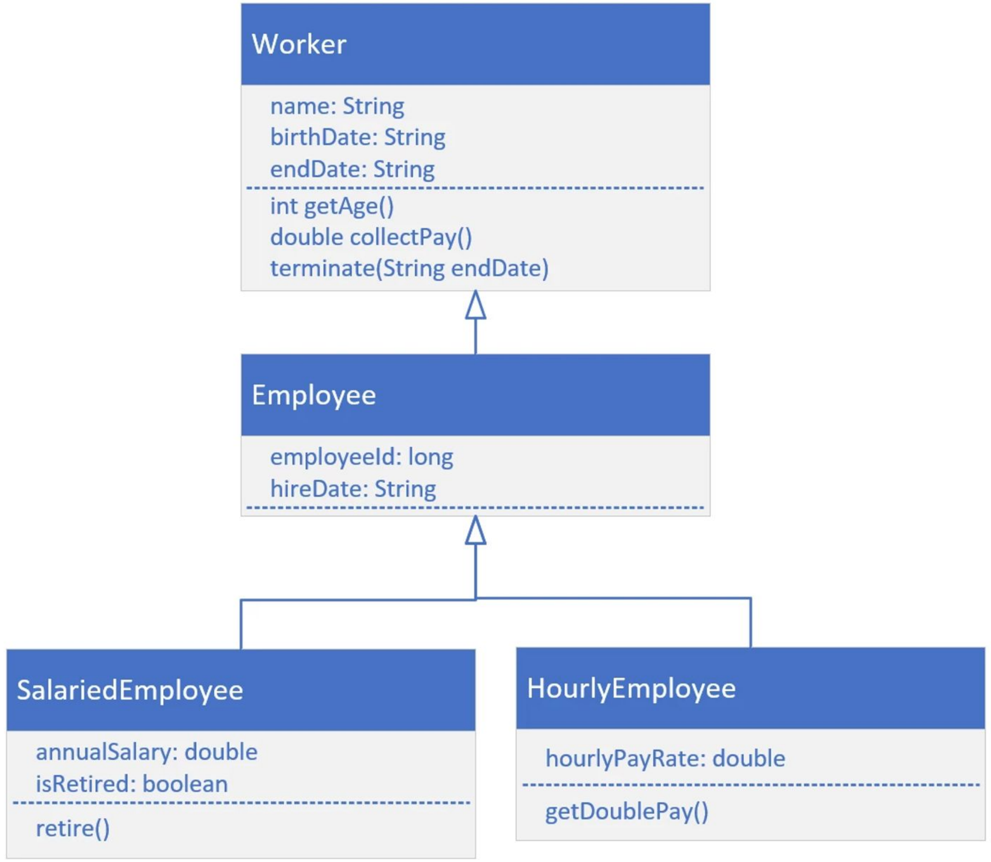


Your challenge is to create the Worker class, the Employee class, and either the SalariedEmployee, or the HourlyEmployee class.

For each class, create the attributes and methods shown on this diagram.

Create a main method that will create either a SalariedEmployee or HourlyEmployee, and call the methods, getAge, collectPay, and the method shown for the specific type of class you decide to implement.

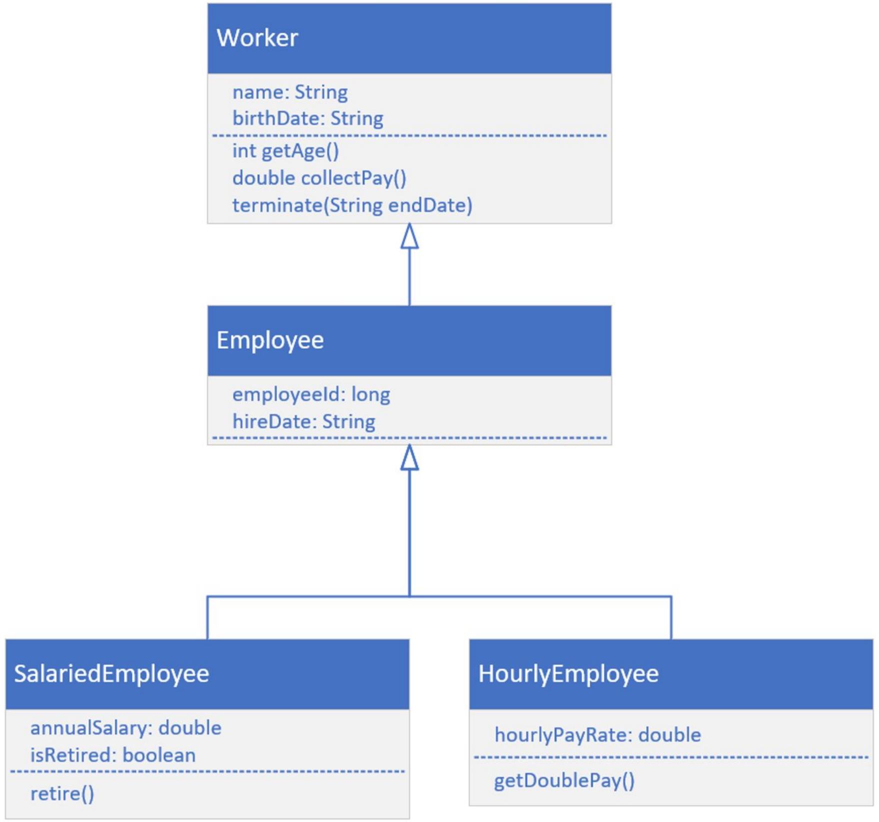
Inheritance Challenge



So if you implement SalariedEmployee, then execute retire().

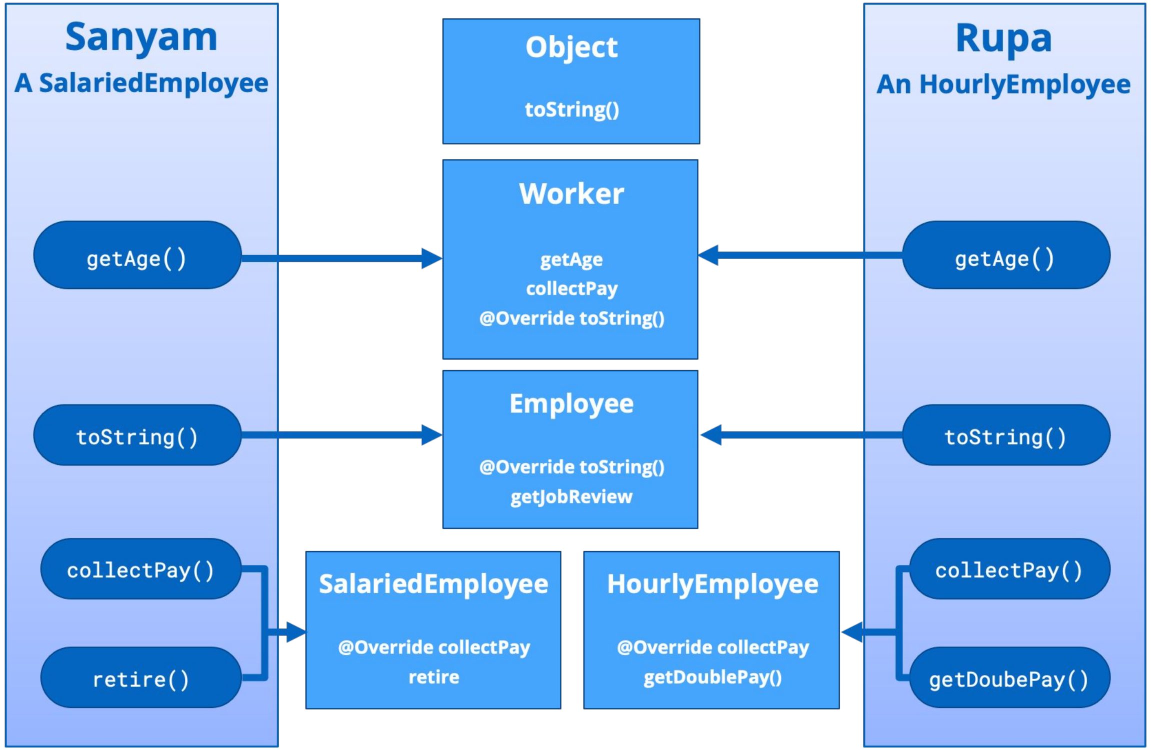
If you implement HourlyEmployee, then execute getDoublePay().

Inheritance Challenge



- A salaried employee, is paid based on some percentage of his or her salary.
- It this person is retired, then the salary may be 100 percent, but it is generally reduced somewhat.
- An hourly employee, is paid by the hours worked, and the hourly rate they agreed to work for.
- An hourly employee may also get double pay, if they work over a certain amount of hours.

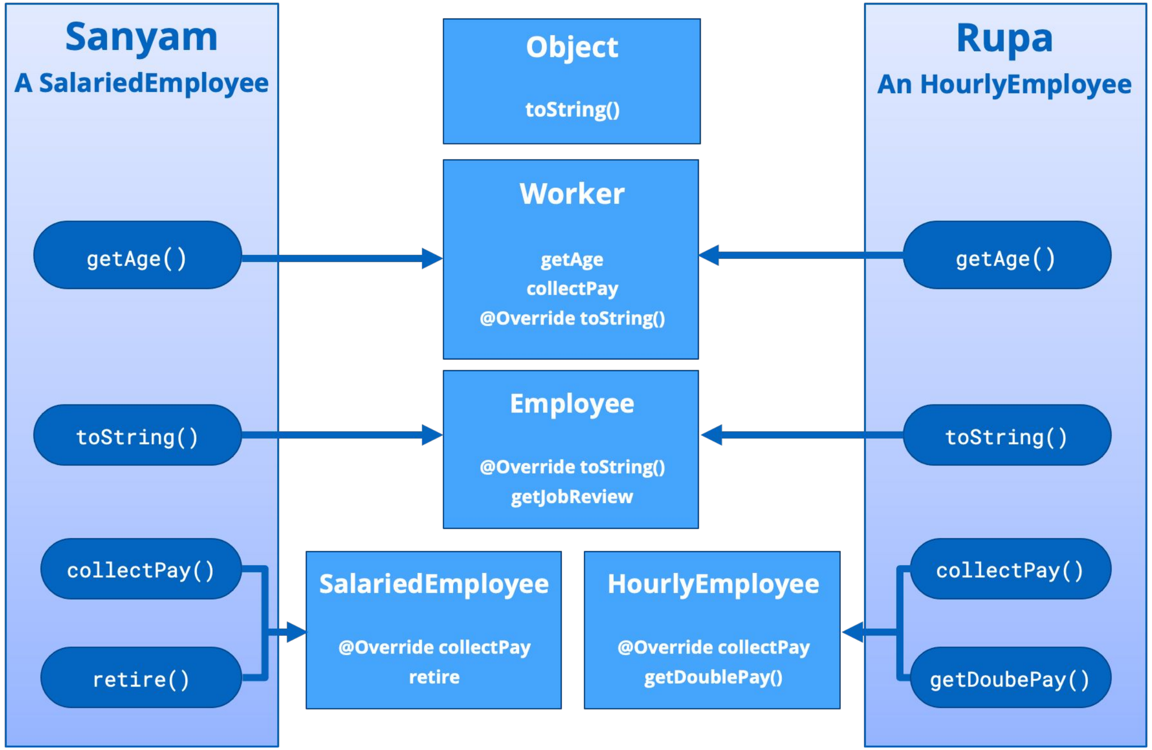
Making the call



Each method call, made on these objects, points to the code that will actually be executed.

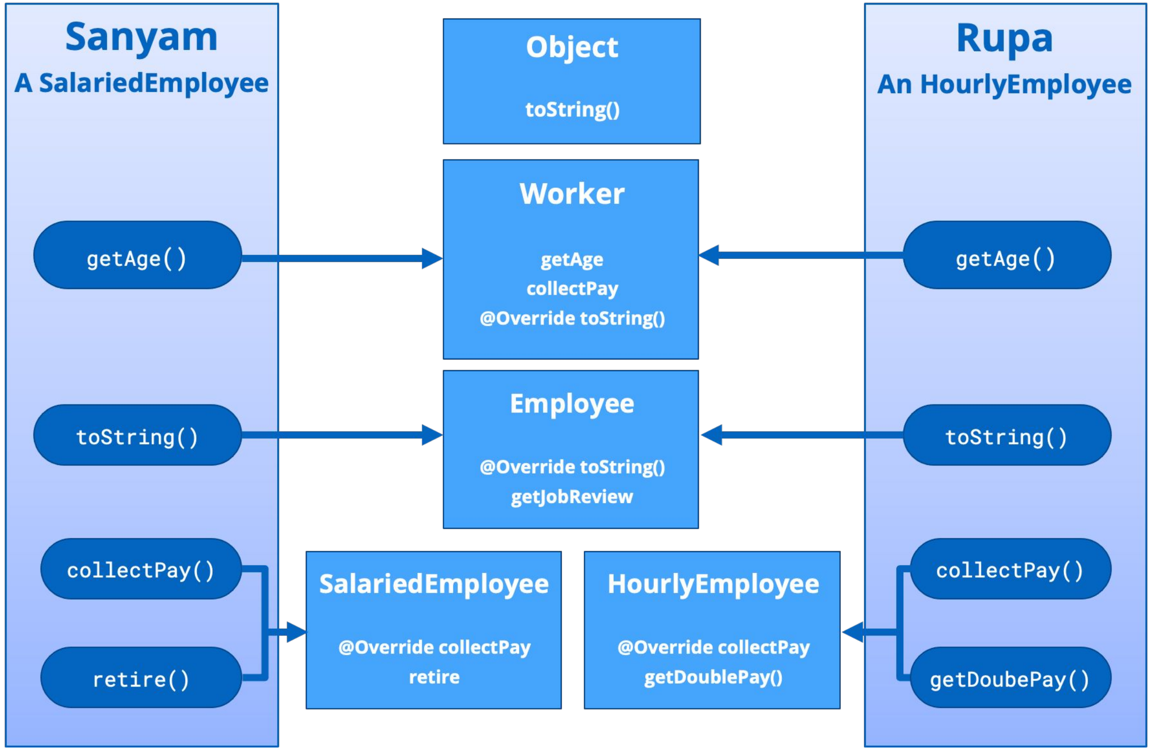
When Sanyam or Rupa call **getAge()**, the method's implementation is on Worker, and not overridden by any other class, so the getAge method on Worker is executed.

Making the call



When Sanyam or Rupa call **toString()**, this method has been overridden twice, first by Worker, and then by Employee. But it wasn't overridden by either SalariedEmployee, or HourlyEmployee, so the method from the Employee class is the one that's used.

Making the call

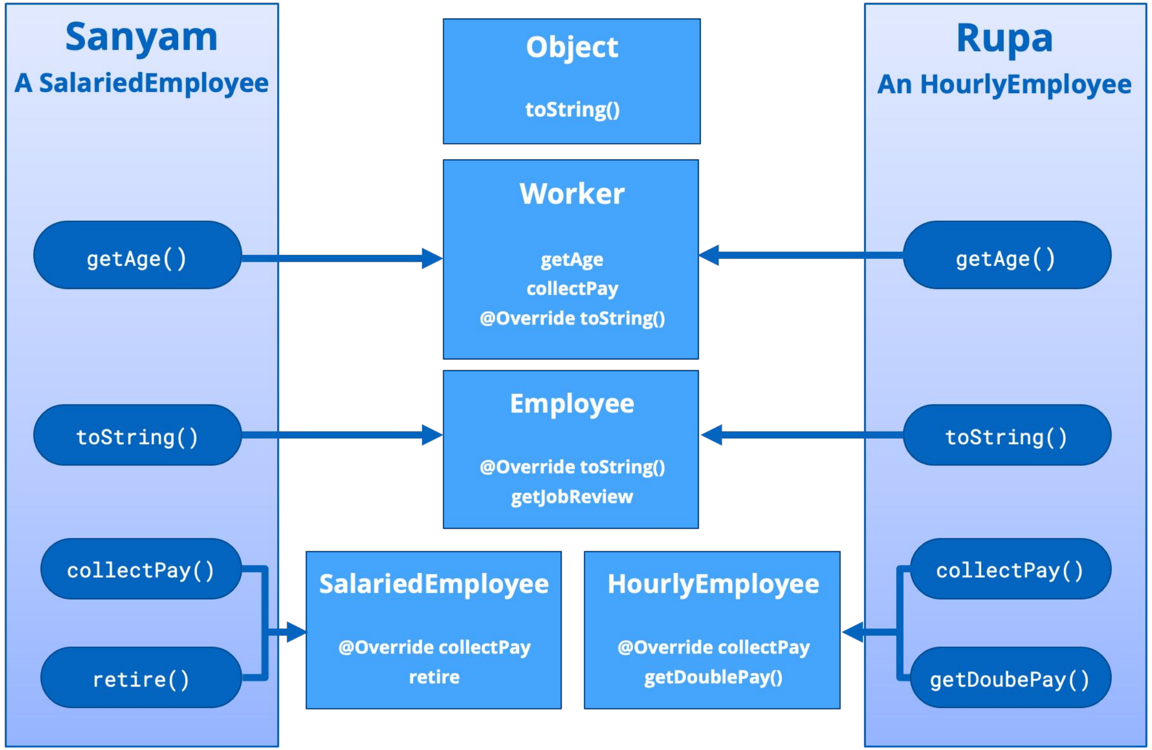


Looking at the **collectPay** method, this method was overridden by both SalariedEmployee, and HourlyEmployee.

Sanyam will execute the method on SalariedEmployee.

Rupa will execute the one on HourlyEmployee.

Making the call



SalariedEmployee has a method, **retire**, that's not overridden, meaning it's only on that class, it's a method specific to a Salaried employee.

HourlyEmployee has its own method, **getDoublePay**, which wouldn't apply to a Salaried employee, so we declared it on this class, and not in any super class.

this vs super

Let's discuss the difference between the **this**, and the **super** keywords.

We'll also find out about the differences between the **this()**, and **super()**, method calls.

this vs super

Let's start with the **super**, and the **this** keywords first.

The keyword **super** is used to *access or call the parent class members* (variables and methods).

The keyword **this** is used to *call the current class members* (variables and methods).

this is required, when we have a parameter with the same name, as an instance variable or field.

NOTE: We can use either of them anywhere in a class, except for static elements, like a static method. Any attempt to do so there, will lead to compile time errors.

this - keyword

```java
public class House {

    private String color;

    public House(String color) {
        // this keyword is required, same parameter name as field
        this.color = color;
    }

    public String getColor() {
        // this is optional
        return color; // same as return this.color;
    }

    public void setColor(String color) {
        // this keyword is required, same parameter name as field
        this.color = color;
    }
}
```

The keyword **this**, is commonly used with constructors and setters, and optionally used in getters.

In this example, we're using the **this** keyword in the **constructor** and **setter**, since there's a parameter with the same name, as the instance or field.

In the getter we don't have any parameters, so there's no conflict, so therefore the **this** keyword is optional there.

super - keyword

```java
class SuperClass {  // parent class aka super class

    public void printMethod() {
        System.out.println("Printed in SuperClass.");
    }
}

class SubClass extends SuperClass {  // subclass aka child class

    // overrides methods from the parent class
    @Override
    public void printMethod() {
        super.printMethod();  // calls the method in the SuperClass (parent)
        System.out.println("Printed in Subclass.");
    }
}

class MainClass {

    public static void main(String[] args) {
        SubClass s = new SubClass();
        s.printMethod();
    }
}
```

The keyword **super**, is commonly used with **method overriding**, when we call a method with the same name, from the parent class.

In this example, we have a method called **printMethod**, that calls **super.printMethod**.

this() vs super() call

In Java we've got the **this()** and **super()** call. Notice the parentheses.

These are known as calls, since it looks like a regular method call, although we're calling certain constructors.

Use **this()** to call a constructor, from another overloaded constructor in the same class.

The call to **this()** can only be used in a constructor, and it must be the first statement in a constructor.

It's used with constructor chaining, in other words when one constructor calls another constructor, and it helps to reduce duplicated code.

The only way to call a parent constructor, is by calling **super()**, which calls the parent constructor.

this() vs super() call

The Java compiler puts a default call to **super()**, if we don't add it, and it's always a call to the no argument constructor, which is inserted by the compiler.

The call to **super()** must be the first statement in each constructor.

A constructor can have a call to **super()** or **this()**, but never both.

Constructors Bad Example

```java
class Rectangle {

    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
    }

    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

Here, we have three constructors.

All three constructors initialize variables.

There's repeated code in each constructor.

We're initializing variables in each constructor, with some default values.

You should never write constructors like this.

Let's look at the right way to do this, by using a **this()** call.

Constructors Good Example

```
class Rectangle {

    private int x;
    private int y;
    private int width;
    private int height;

    // 1st constructor
    public Rectangle() {
        this(0, 0);  // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int width, int height) {
        this(0, 0, width, height);  // calls 3rd constructor
    }

    // 3rd constructor
    public Rectangle(int x, int y, int width, int height) {
        // intialize variables
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

In this example, we still have three constructors.

The 1st constructor calls the 2nd constructor, the 2nd constructor calls the 3rd constructor, and the 3rd constructor initializes the instance variables.

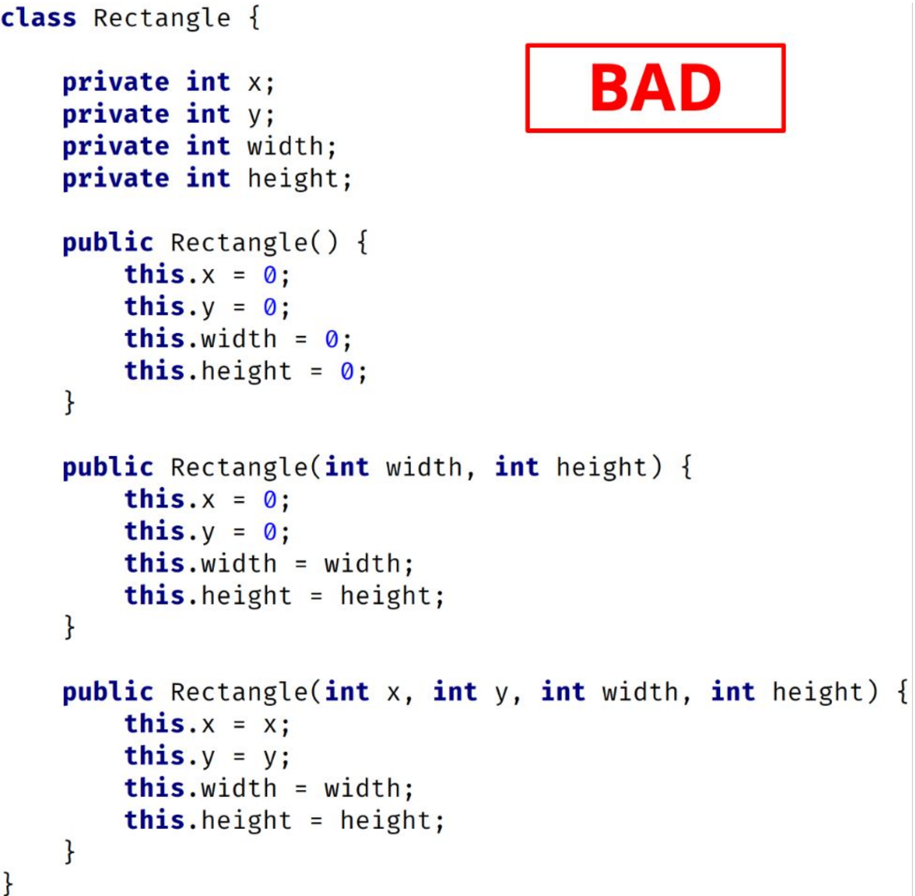
The 3rd constructor does all the work.

No matter what constructor we call, the variables will always be initialized in the 3rd constructor.

This is known as constructor chaining, the last constructor has the **responsibility** to initialize the variables.

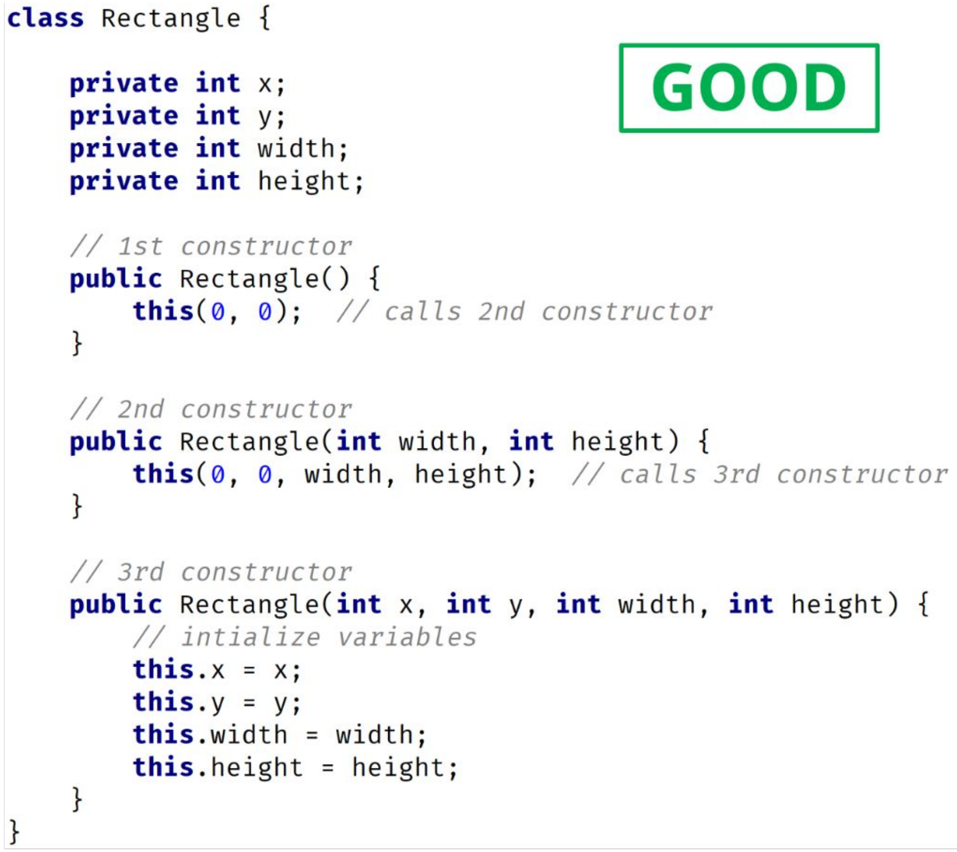
Comparing Both Examples

BAD



```java
class Rectangle {

    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
    }

    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

GOOD



```java
class Rectangle {

    private int x;
    private int y;
    private int width;
    private int height;

    // 1st constructor
    public Rectangle() {
        this(0, 0);  // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int width, int height) {
        this(0, 0, width, height);  // calls 3rd constructor
    }

    // 3rd constructor
    public Rectangle(int x, int y, int width, int height) {
        // intialize variables
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

super() call example

```java
class Shape {

    private int x;
    private int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Rectangle extends Shape {

    private int width;
    private int height;

    // 1st constructor
    public Rectangle(int x, int y) {
        this(x, y, 0, 0);  // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int x, int y, int width, int height) {
        super(x, y);  // calls constructor from parent (Shape)
        this.width = width;
        this.height = height;
    }
}
```

In this example, we have a class **Shape**, with x and y variables, and a class **Rectangle** that extends **Shape**, with variables width and height.

In the Rectangle class, the 1st constructor is calling the 2nd constructor.

The 2nd constructor calls the parent constructor, with parameters x and y.

The parent constructor will initialize the x and y variables, while the 2nd Rectangle constructor will initialize the width and height variables.

Here, we have both the **super()** and **this()** calls.

Method Overloading vs Method Overriding

Method Overloading

Method overloading means providing two or more separate methods, in a class, with the **same name**, but **different parameters**.

Method return type may or may not be different, and that allows us to reuse the same method name.

Overloading is very handy, it **reduces duplicated code**, and we don't have to remember multiple method names.

We can overload static, or instance methods.

Method Overloading

To the code calling an overloaded method, it looks like a single method can be called, with different sets of arguments.

In actuality, each call that's made with a different set of arguments, is calling a separate method.

Java developers often refer to method overloading, as **compile-time polymorphism.**

This means the compiler is determining the right method to call, based on the method name and argument list.

Method Overloading

Usually **overloading** happens within a **single class**.

But methods can also be overloaded by subclasses.

That's because, a subclass inherits one version of the method from the parent class, and then the subclass can have another overloaded version of that method.

Method Overloading Rules

Methods will be considered overloaded if both methods follow the following rules:

- Methods must have the same method name.
- Methods must have different parameters.

If methods follow the rules above:

- They may or may not have different return types.
- They may or may not have different access modifiers.
- They may or may not throw different checked or unchecked exceptions.

Method Overriding

Method overriding, means defining a method in a child class, that already exists in the parent class, with the same signature (the **same name, same arguments**).

By extending the parent class, the child class gets all the methods defined in the parent class (those methods are also known as derived methods).

Method overriding is also known as **Runtime Polymorphism**, or **Dynamic Method Dispatch**, because the method that is going to be called, is decided at runtime, by the Java virtual machine.

Method Overriding

When we **override** a method, it's recommended to put **@Override**, immediately above the method definition.

The @Override statement is not required, but it's a way to get the compiler to flag an error, if you don't actually properly override this method.

We'll get an error, if we don't follow the overriding rules correctly.

We can't override static methods, **only instance methods** can be overridden.

Method Overriding Rules

A method will be considered overridden, if we follow these rules.

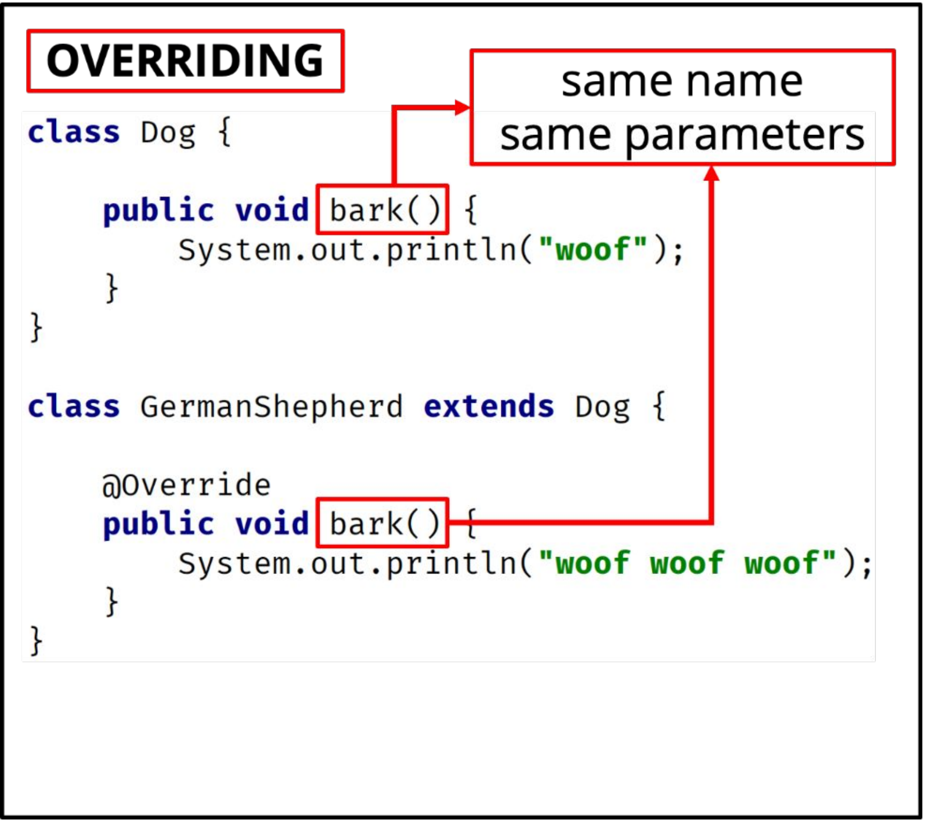
- It must have the same name and same arguments.
- The return type can be a subclass of the return type in the parent class.
- It can't have a lower access modifier. In other words, it can't have more restrictive access privileges.
- For example, if the parent's method is protected, then using private in the child's overridden method is not allowed. However, using public for the child's method would be allowed, in this example.

Method Overriding Rules

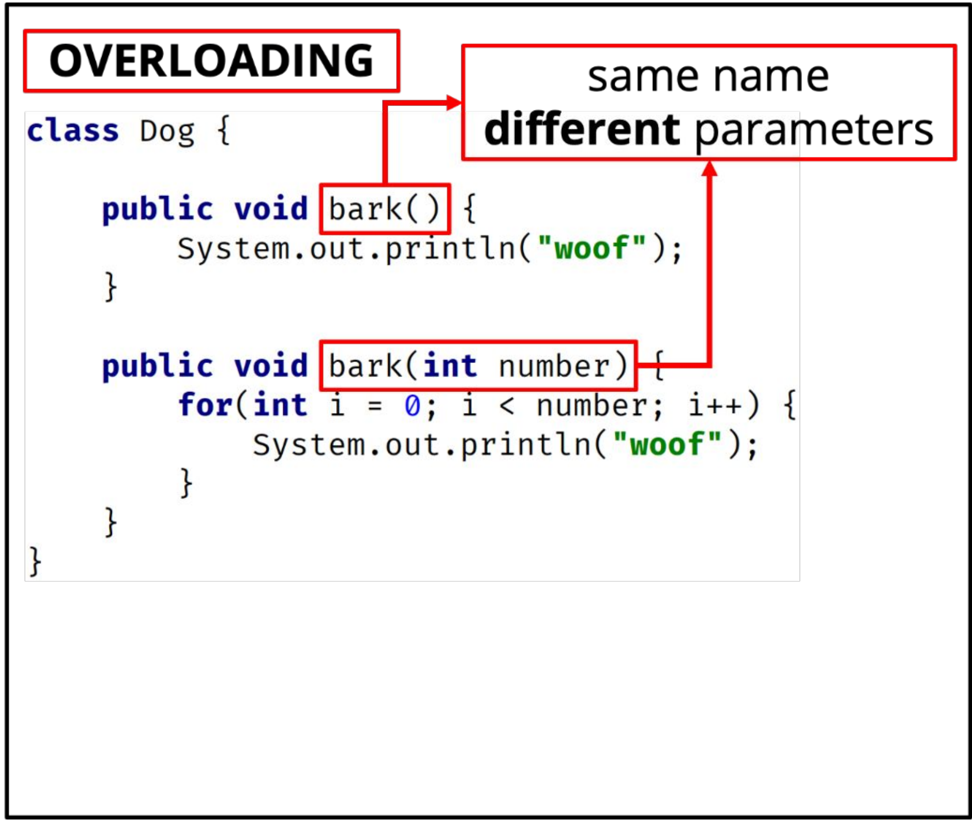
There's also some important points about method overriding to keep in mind.

- Only **inherited methods** can be overridden, in other words, methods can be overridden only in child classes.
- Constructors and private methods cannot be overridden.
- Methods that are final cannot be overridden.
- A subclass can use super.methodName() to call the superclass version of an overridden method.

Method Overriding vs. Overloading



```java
class Dog {

    public void bark() {
        System.out.println("woof");
    }
}

class GermanShepherd extends Dog {

    @Override
    public void bark() {
        System.out.println("woof woof woof");
    }
}
```



```java
class Dog {

    public void bark() {
        System.out.println("woof");
    }

    public void bark(int number) {
        for(int i = 0; i < number; i++) {
            System.out.println("woof");
        }
    }
}
```

Method Overriding vs. Overloading

Method Overloading	Method Overriding
Provides functionality to reuse a method name with different parameters.	Used to override a behavior which the class has inherited from the parent class.
Usually in a single class but may also be used in a child class.	**Always in two classes** that have a child-parent or IS-A relationship.
Must have different parameters.	**Must have** the same parameters and same name.
May have different return types.	**Must have** the same return type or covariant return type(child class).
May have different access modifiers(private, protected, public).	**Must NOT** have a lower modifier but may have a higher modifier.
May throw different exceptions.	**Must NOT** throw a new or broader checked exception.