Introduction

Welcome to section 7 of this course.

Tim Buchalka here, and this is part one of a two-part section, about Object-Oriented Programming.

In part one, you're going to learn the fundamentals of Object-Oriented Programming, starting with the structures you'll be using, specifically Classes, Objects, and Constructors.

Along with these structures, you'll be learning some fundamental features of Object-Oriented Programming.

Namely Inheritance, Encapsulation, Polymorphism, and Composition.

{LP} LearnProgramming .academy

Introduction

In this section, I'll cover Inheritance, and in the next one, we'll explore the other 3 features.

So, let's make a start on Object-Oriented Programming.

{LP} LearnProgramming .academy

Introduction to Classes and Objects

In this video, I'm going to start talking about **Object-Oriented Programming**.

{LP} LearnProgramming .academy

What is Object Oriented Programming

What is Object-Oriented Programming?

Object-Oriented Programming is a way to model real-world objects as software objects which contain both data and code.

OOP is a common acronym for Object-Oriented Programming.

{LP} LearnProgramming .academy

Class-based Programming

Class-based programming starts with classes which become the blueprints for objects.

But what does this really mean?

To start, we need to understand what objects are.

They're really the key to understanding this object-oriented terminology.

{LP} LearnProgramming .academy

Real World Object Exercise

What I'd like you to do is just have a look around in the area you're sitting in right now.

And if you do that, you'll find that there's many examples of real-world objects.

For example, I'm sitting here, and I can see:

- A computer.
- I can see a keyboard.
- I can see a microphone.
- I can see shelves on the wall.
- I can see a door.

All of these are examples of real-world objects.

State and Behavior

Real-world objects have two major components:

- state.
- and behavior.

State (computer)

State, in terms of a computer object, might be:

- The amount of RAM it has.
- The operating system it's running.
- The hard drive size.
- The size of the monitor.

These are characteristics about the item that can describe it.

State (ant)

I could also describe animate objects like people, or animals, or even insects like an ant.

For an ant, the state might be:

- The age.
- The number of legs.
- The conscious state.
- Whether the ant is sleeping or is awake.

Behavior (computer)

In addition to state, objects may also have behavior or actions that can be performed by the object, or upon the object.

Behavior, for a computer, might be things like:

- Booting up.
- Shutting down.
- Beeping or outputting some form of sound.
- Drawing something on the screen, and so on.

All of these could be described as behaviors for a computer.

Behavior (ant)

For an ant, behavior might be:

- Eating.
- Drinking.
- Fighting.
- Carrying food, those types of things.

{LP} LearnProgramming .academy

State and Behavior

Modelling real-world objects as software objects is a fundamental part of Object-Oriented Programming.

A software object stores its state in fields, which can also be called variables or attributes.

Objects expose their behavior with methods which I've talked about before.

So, where does a class fit in?

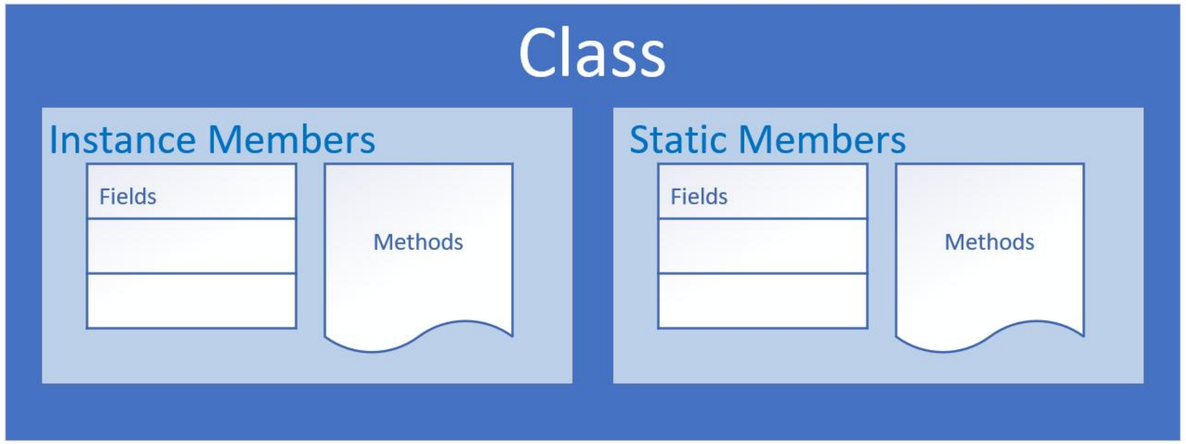
Well, think of a class as a template or a blueprint for creating objects.

Let's take another look at the class.

The class as the blueprint

The class describes the data (fields), and the behavior (methods), that are relevant to the real-world object we want to describe.

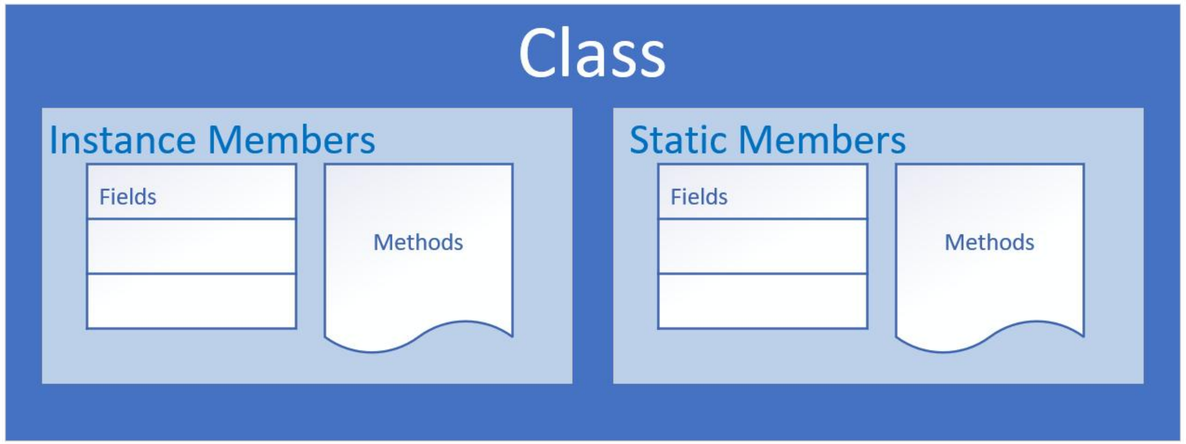
These are called class members.



A class member can be a field or a method, or some other type of dependent element.

If a field is static, there is only one copy in memory, and this value is associated with the class or template itself.

The class as the blueprint



If a field is not static, it's called an instance field, and each object may have a different value stored for this field.

A static method can't be dependent on any one object's state, so it can't reference any instance members.

In other words, any method that operates on instance fields needs to be non-static.

Organizing classes

Classes can be organized into logical groupings which are called packages.

You declare a package name in the class using the package statement.

If you don't declare a package, the class implicitly belongs to the default package.

{LP} LearnProgramming .academy

Access modifiers for the class

A class is said to be a top-level class if it is defined in the source code file and not enclosed in the code block of another class, type, or method.

A top-level class has only two valid access modifier options: public or none.

Access keyword	Description
public	public means any other class in any package can access this class.
	When the modifier is omitted, this has special meaning, called package access, meaning the class is accessible only to classes in the same package.

Access modifiers for class members

An access modifier at the member level allows granular control over class members.

The valid access modifiers are shown in this table from the least restrictive to the most restrictive

Access keyword	Description
public	public means any other class in any package can access this class.
protecte d	protected allows classes in the same package, and any subclasses in other packages, to have access to the member.
	When the modifier is omitted, this has special meaning, called package access, meaning the member is accessible only to classes in the same package
private	private means that no other class can access this member

Encapsulation

Encapsulation in Object-Oriented Programming usually has two meanings.

One is the bundling of behavior and attributes on a single object.

The other is the practice of hiding fields and some methods from public access.

{LP} LearnProgramming .academy

What is null?

null is a special keyword in Java, meaning, the variable or attribute has a type but no reference to an object.

This means that no instance or object is assigned to the variable or field.

Fields with primitive data types are never null.

Default values for fields on classes

Fields on classes are assigned default values automatically by Java, if you don't assign values yourself.

Data type	Default value assigned
boolean	false
byte short int long char	0
double float	0.0

What are getters and setters? Why should we use them?

What are getters and setters?

A getter is a method on a class that retrieves the value of a private field and returns it.

A setter is a method on a class that sets the value of a private field.

The purpose of these methods is to control and protect access to private fields.

this

this is a special keyword in Java.

What it really refers to is the instance that was created when the object was instantiated.

So, **this** is a special reference name for the object or instance, which it can use to describe itself.

And we can use **this** to access fields on the class.

Classes Challenge Exercise

In the previous videos, I've been talking about classes.

It's now challenge time. I want to challenge your understanding of those previous videos.

So, here's what I want you to do.

{LP} LearnProgramming .academy

Object Oriented Challenge

Create a new class for a bank account.

Create fields for account characteristics like:

- account number.
- account balance.
- customer name.
- email.
- phone number.

{LP} LearnProgramming .academy

Object Oriented Challenge

Create getters and setters for each field.

Create two additional methods:

- one for depositing funds into the account.
- one for withdrawing funds from the account.

{LP} LearnProgramming .academy

Object Oriented Challenge

A customer should not be allowed to withdraw funds if that withdrawal takes their balance negative.

Create a new project called ClassesChallenge with the usual Main class with the usual main method.

You'll create an instance of an Account class and then test your withdraw and deposit methods.

You'll print information to the console that confirms what the balance is after the methods are called.

Object Oriented Challenge

You want to make this class encapsulated, so you'll make all your fields private and set up getter and setter methods for your fields.

In addition, you'll have two behavioral methods. One for depositing funds, the other for withdrawing funds.

In addition to this class, you'll set up a main class with a main method that creates at least one instance of the bank account class and simulates depositing and withdrawing money from the account.

{LP} LearnProgramming .academy

Constructor

A **constructor** is used in the creation of an object.

It is a special type of code block that has a specific name and parameters, much like a method.

It has the same name as the class itself, and it doesn't return any values.

You never include a return type from a constructor, not even void.

You can, and should, specify an appropriate access modifier to control who should be able to create new instances of the class, using this constructor.

```
public class Account {  // This is the class declaration

    public Account() {  // This is the constructor declaration
        // Constructor code is code to be executed as the object is created.
    }
}
```


{LP} LearnProgramming .academy

The default constructor

If a class contains no constructor declarations, then a default constructor is implicitly declared.

This constructor has no parameters and is often called the no-args (no arguments) constructor.

If a class contains any other constructor declarations, then a default constructor is NOT implicitly declared.

Constructor overloading

Constructor overloading is declaring multiple constructors with different parameters.

The number of parameters can be different between constructors.

Or if the number of parameters is the same between two constructors, their types, or order of the types must differ.

{LP} LearnProgramming .academy

Constructor chaining with this()

Constructor chaining is when one constructor explicitly calls another overloaded constructor.

You can only use constructor chaining, within constructors.

You must use the special statement this() to execute another constructor, passing it arguments if required.

And this() must be the first executable statement if it's used from another constructor.

{LP} LearnProgramming .academy

Constructor Challenge Exercise

In this video, I'm going to give you a challenge to review what you have learned about constructors so far.

{LP} LearnProgramming .academy

Constructor Challenge

So for this challenge, you'll want to:

1. Create a new class, called Customer, with three fields:
   - name
   - credit limit
   - email address.
2. Create the getter methods only for each field. You don't need to create the setters.

Constructor Challenge

3. Create three constructors for this class:

- First, create a constructor for all three fields which should assign the arguments directly to the instance fields.
- Second, create a no-args constructor that calls another constructor, passing some literal values for each argument.
- And lastly, create a constructor with just the name and email parameters, which also calls another constructor.

Reference vs Object vs Instance vs Class

In this video, I'm going to discuss, and hopefully clear up any confusion about references, vs. objects, vs. instances, vs. classes.

{LP} LearnProgramming .academy

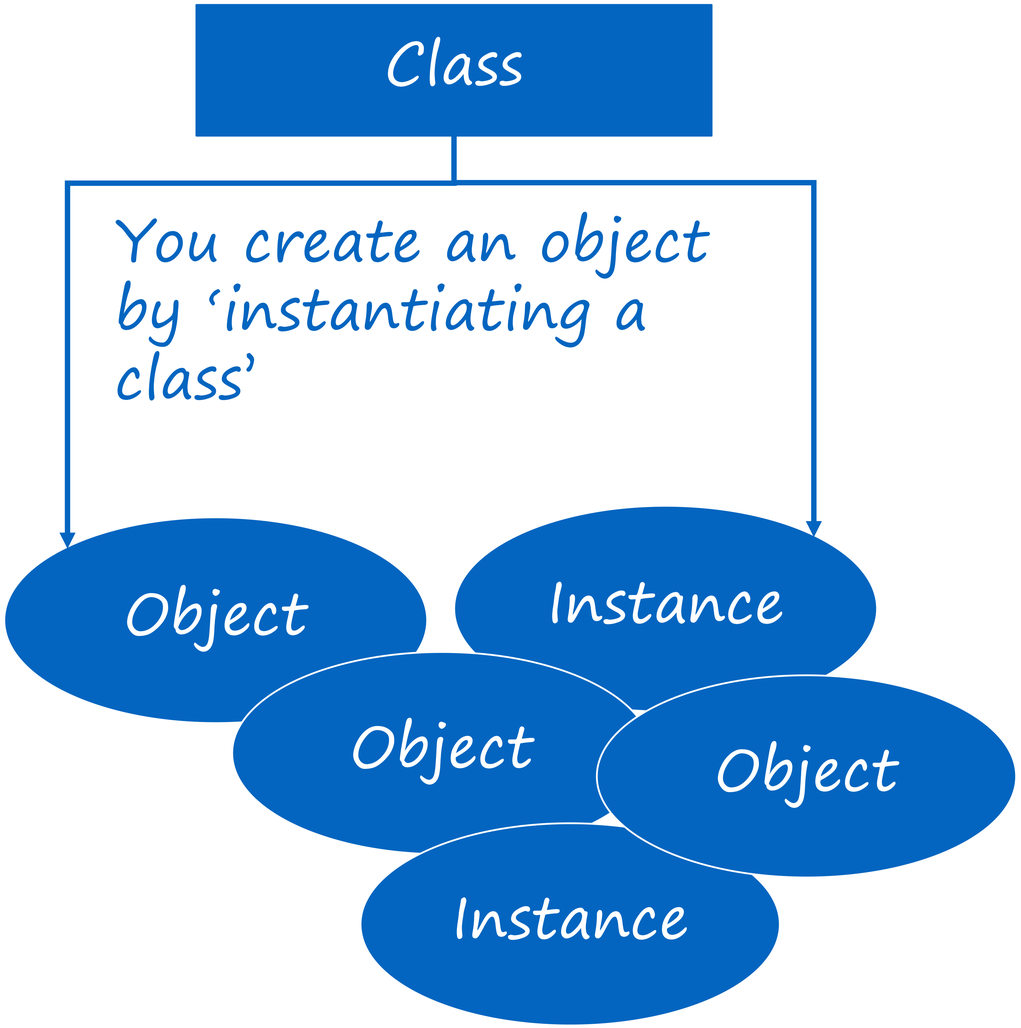
Reference vs Object vs Instance vs Class

By now, you've probably noticed that I use the words **reference**, **object**, **instance** and **class** frequently.

These new concepts may well be confusing at first.

In this video, I'm going to go through all these terms, and show you exactly what each of these words mean in the context of Java programming.

Object vs Instance vs Class



You can create many objects using a single class. Each may have unique attributes or values

Object and instance are interchangeable terms

Reference vs Object vs Instance vs Class

Let's use the analogy of building a house to understand classes.

A class is basically a blueprint for the house.

Using the blueprint, we can build as many houses as we like based on those plans.

Each house we build (in other words, going back to programming terms, each house we instantiate using the new operator) is an object.

This object can also be known as an instance. Often, we'll say it's an instance of the class. So, we would have an instance of house in this example.

Getting back to the physical world, each house we build has an address (it's built at a physical location).

In other words, if we want to tell someone where we live, we give them our address (perhaps written on a piece of paper). This is known as a reference.

Reference vs Object vs Instance vs Class

We can copy that reference as many times as we like, but there is still just one house that we're referring to.

In other words, we're copying the paper that has the address on it, not the house itself.

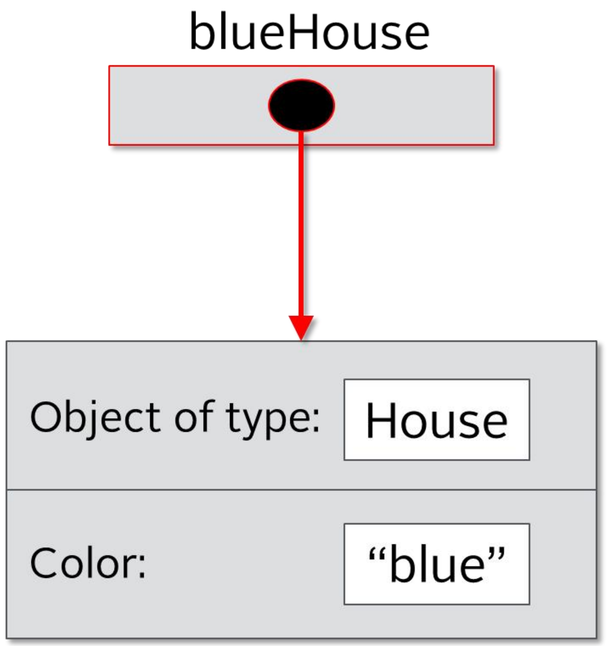
We can pass references as parameters to constructors and methods.

Reference vs Object vs Instance vs Class

```java
public class House {

    private String color;

    public House(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("red");
        System.out.println(blueHouse.getColor()); // red
        System.out.println(anotherHouse.getColor());// red

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //red
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

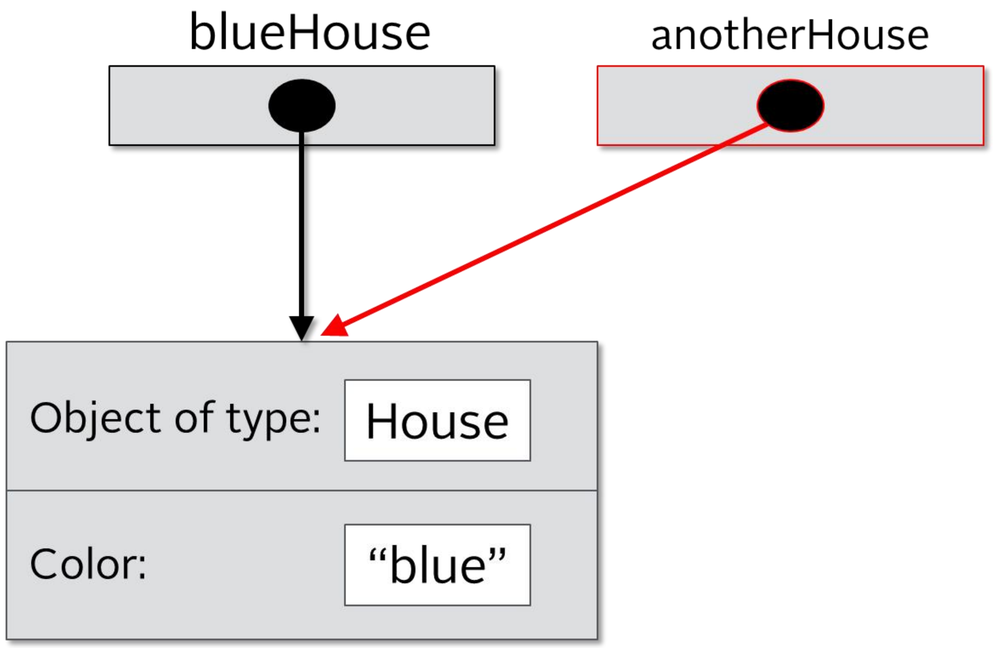
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("red");
        System.out.println(blueHouse.getColor()); // red
        System.out.println(anotherHouse.getColor());// red

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //red
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

*The line **House blueHouse = new House(“blue”);** creates a new **instance** of the House **class**. Remember House is a blueprint, and we are assigning it to the blueHouse **variable**. In other words it is a **reference** to the **object** in memory. The image on the left hopefully makes sense to you now.*

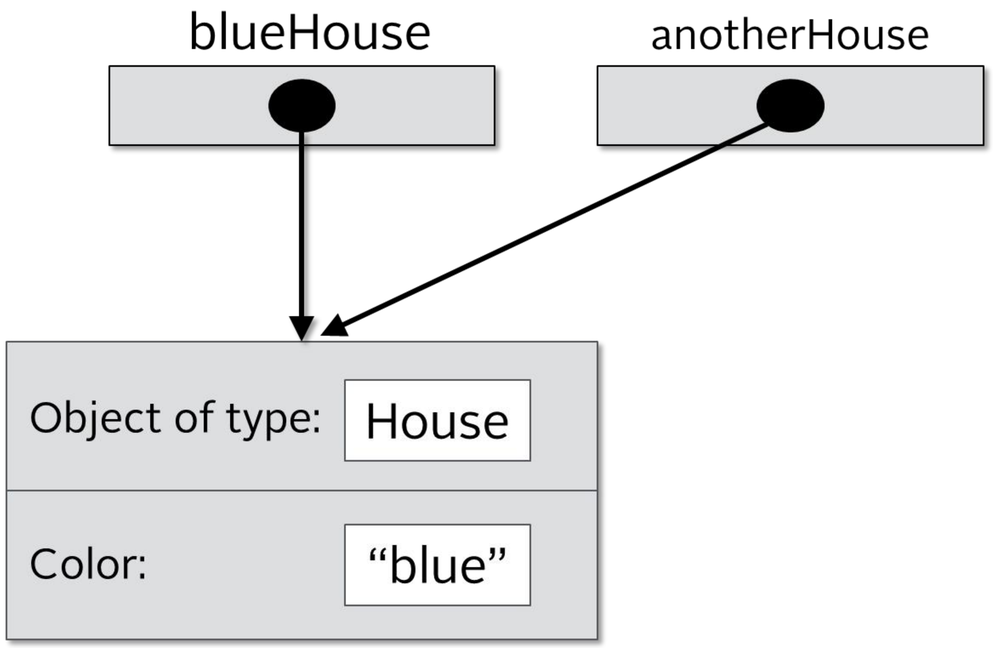
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("red");
        System.out.println(blueHouse.getColor()); // red
        System.out.println(anotherHouse.getColor());// red

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //red
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

The next line **House anotherHouse = blueHouse;** creates another **reference** to the same **object** in memory. Here we have two **references** pointing to the same object in memory. There is still one house, but two **references** to that one **object**. In other words we have two pieces of paper with the physical address of where the house is built (going back to our real world example).

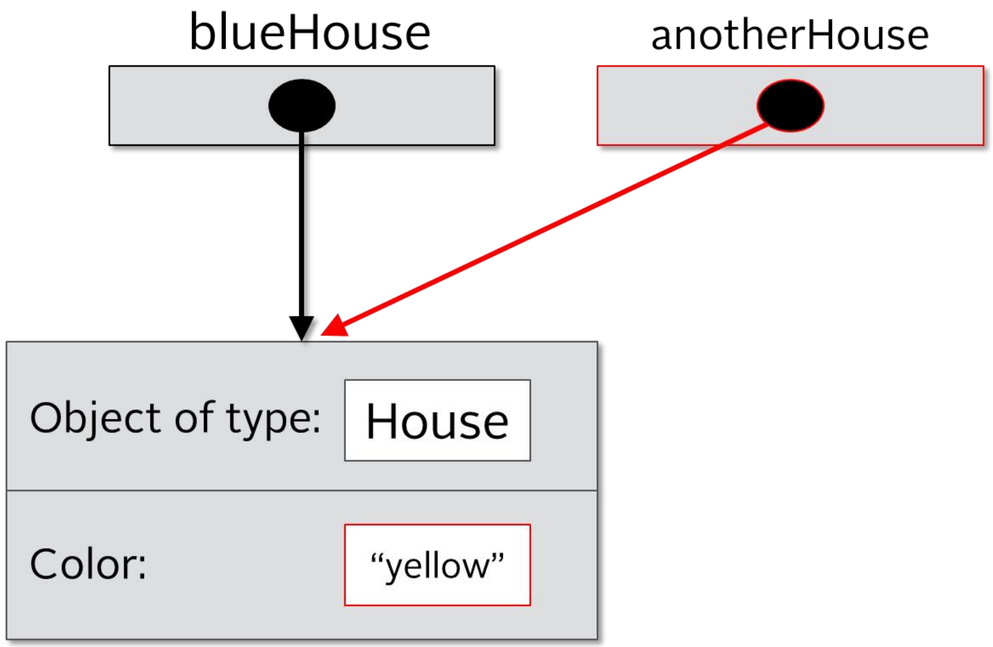
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("red");
        System.out.println(blueHouse.getColor()); // red
        System.out.println(anotherHouse.getColor());// red

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //red
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

*Next we have two println statements that print the blueHouse color and anotherHouse color. Both will print “blue” since we have two **references** to the same **object**.*

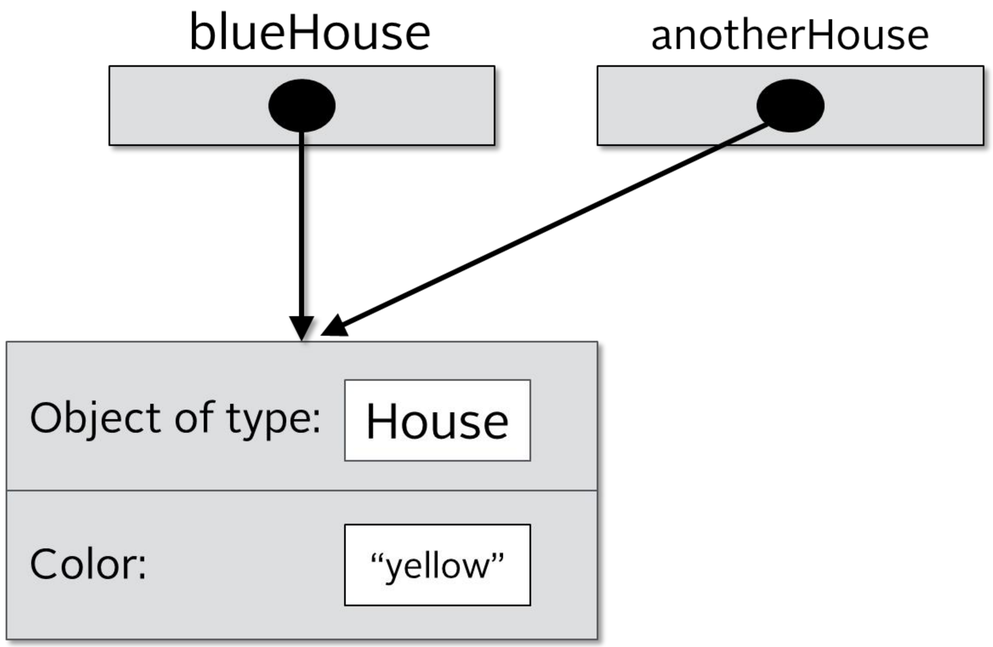
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("yellow");
        System.out.println(blueHouse.getColor()); // yellow
        System.out.println(anotherHouse.getColor());// yellow

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //yellow
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

*The next line calls the method setColor and sets the color to yellow. To the left you can that both blueHouse and anotherHouse have the same color now. Why? Remember we have two **references** that point to the same **object** in memory. Once we change the color, of one, **both references** still point to the same **object**. In our real world example, there is still just one physical house at that one address, even though we have written the same address on two pieces of paper.*

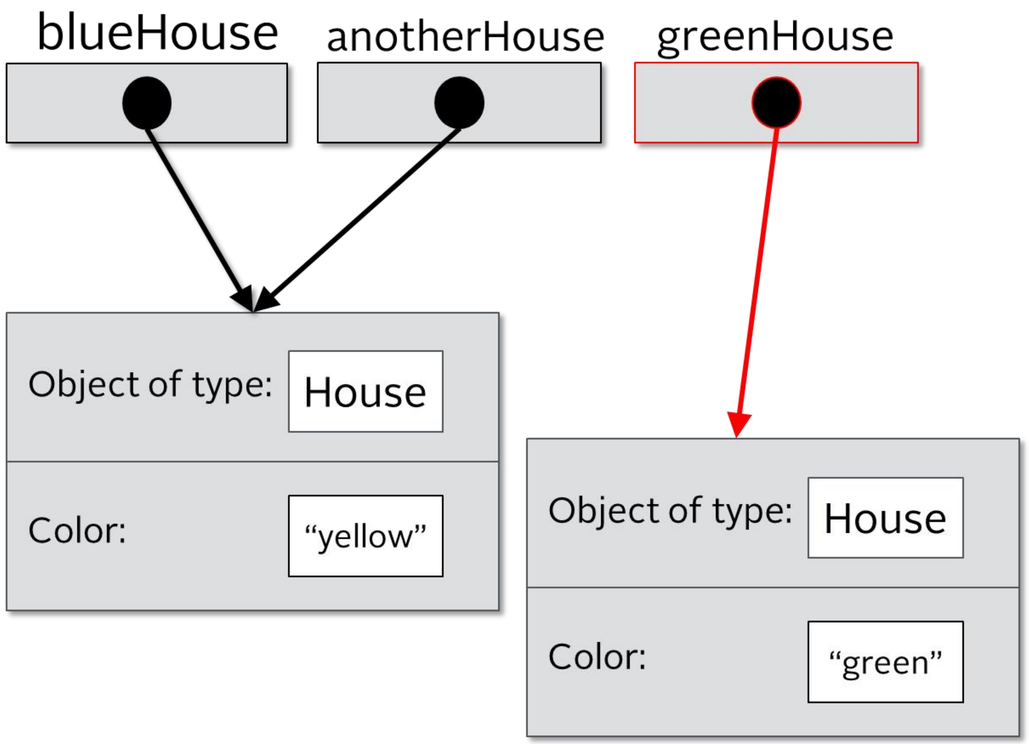
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("yellow");
        System.out.println(blueHouse.getColor()); // yellow
        System.out.println(anotherHouse.getColor());// yellow

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //yellow
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

Here we have two println statements that are printing the color. Both now print “yellow” since we still have two **references** that point to the same **object** in memory. Notice the arrows on the left hand side.

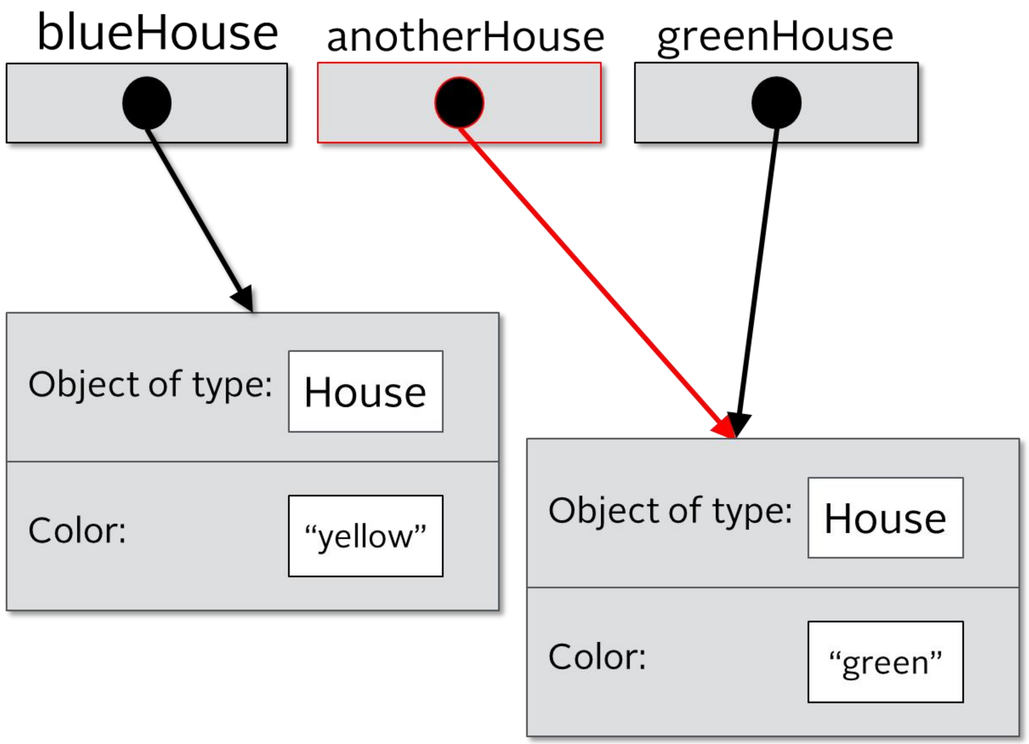
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("yellow");
        System.out.println(blueHouse.getColor()); // yellow
        System.out.println(anotherHouse.getColor());// yellow

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //yellow
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

*Here we are creating another new instance of the House class with the color set to "green". Now we have two **objects** in memory but we have three **references** which are blueHouse, anotherHouse and greenHouse. The variable (**reference**) greenHouse points to a different **object** in memory, but blueHouse and anotherHouse point to the same object in memory.*

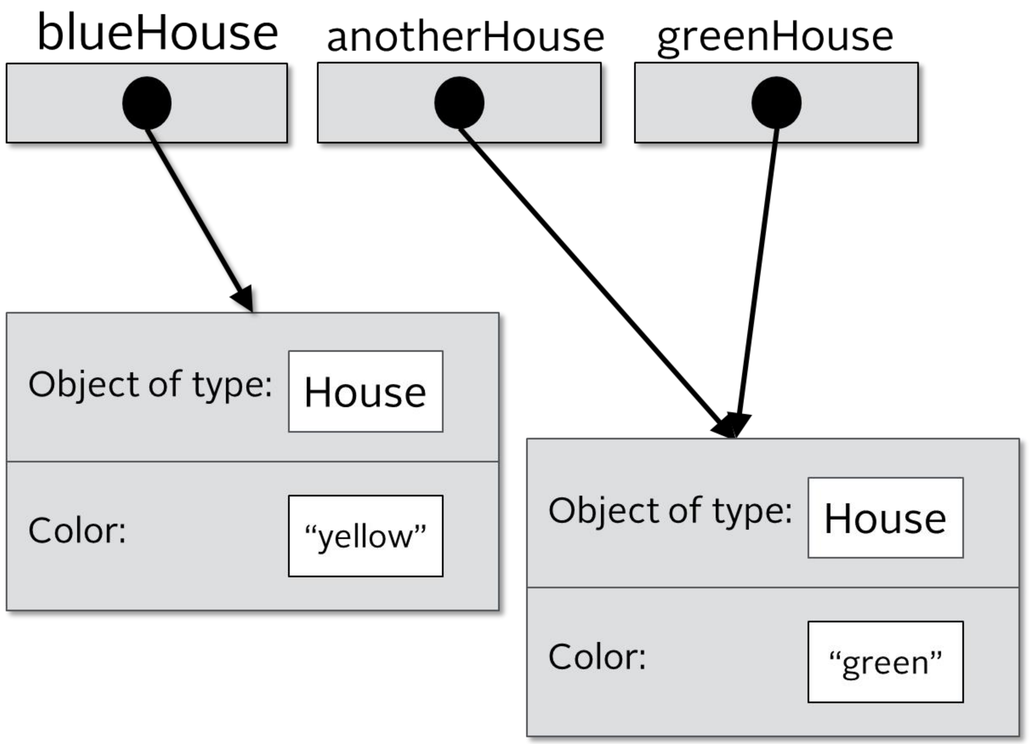
Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("yellow");
        System.out.println(blueHouse.getColor()); // yellow
        System.out.println(anotherHouse.getColor());// yellow

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //yellow
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

*Here we assigns greenHouse to anotherHouse. In other words we are dereferencing anotherHouse. It will now point to a different **object** in memory. Before it was pointing to a house that had the "yellow" color, now it points to the house that has the "green" color. In this scenario we still have three **references** and two **objects** in memory but blueHouse points to one **object** while anotherHouse and greenHouse point to the same **object** in memory.*

Reference vs Object vs Instance vs Class




```java
public class Main {

    public static void main(String[] args) {

        House blueHouse = new House("blue");
        House anotherHouse = blueHouse;

        System.out.println(blueHouse.getColor()); // prints blue
        System.out.println(anotherHouse.getColor());// blue

        anotherHouse.setColor("yellow");
        System.out.println(blueHouse.getColor()); // yellow
        System.out.println(anotherHouse.getColor());// yellow

        House greenHouse = new House("green");
        anotherHouse = greenHouse;

        System.out.println(blueHouse.getColor()); //yellow
        System.out.println(greenHouse.getColor());// green
        System.out.println(anotherHouse.getColor());// green
    }
}
```

Finally we have three println statements. The first will print "yellow" since the blueHouse **variable(reference)** points to the **object** in memory that has the "yellow" color, while the next two lines will print "green" since both anotherHouse and greenHouse point to same **object** in memory.

The reference vs The object

Consider the code on this slide.

```
1   new House("red");                    // house object gets created in memory
2   House myHouse = new House("beige");  // house object gets created in memory
                                         // and it's location (reference) is
                                         // assigned to myHouse
3   House redHouse = new House("red");   // house object gets created in memory
                                         // and it's location (reference) is
                                         // assigned to redHouse
```

On the first line, I create a new House and make it red.

But I am not assigning this to any variable.

The reference vs The object

```
1    new House("red");                    // house object gets created in memory
```

This compiles fine and you can do this.

This object is created in memory, but after that statement completes, my code has no way to access it.

The object exists in memory, but I can't communicate with it after that statement is executed.

That's because I didn't create a reference to it.

The reference vs The object

On the second line, I do create a reference to the house object I created.

```
2    House myHouse = new House("beige");      // house object gets created in memory
                                              // and it's location (reference) is
                                              // assigned to myHouse
```

My reference, the variable I call myHouse, lets me have access to that beige house as long as my variable, myHouse, stays in scope. Or until it gets reassigned to reference a different object.

The reference vs The object

I'm creating a red house again, but this is a different object altogether from the red house I created on line one.

```
3   House redHouse = new House("red");      // house object gets created in memory
                                            // and it's location (reference) is
                                            // assigned to redHouse
```

This third statement is creating yet another house object in memory which has no relationship to the one I created on the first line.

```
1   new House("red");                       // house object gets created in memory
2   House myHouse = new House("beige");     // house object gets created in memory
                                            // and it's location (reference) is
                                            // assigned to myHouse
```

The reference vs The object

```
1   new House("red");                          // house object gets created in memory
2   House myHouse = new House("beige");        // house object gets created in memory
                                               // and it's location (reference) is
                                               // assigned to myHouse
3   House redHouse = new House("red");         // house object gets created in memory
                                               // and it's location (reference) is
                                               // assigned to redHouse
```

This code has three instances of house but only two references.

That first object is said to be eligible for garbage collection immediately after that first statement.

It's useless to the code because It's no longer accessible.

The reference vs The object

```
1   new House("red");                          // house object gets created in memory
2   House myHouse = new House("beige");        // house object gets created in memory
                                               // and it's location (reference) is
                                               // assigned to myHouse
3   House redHouse = new House("red");         // house object gets created in memory
                                               // and it's location (reference) is
                                               // assigned to redHouse
```

There are times we might want to instantiate an object and immediately call a method on it.

99 percent of the time, we'll want to reference the objects we create.

Static vs Instance Variables

Let's discuss the differences between static variables and instance variables.

{LP} LearnProgramming .academy

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.

If changes are made to that variable, all other instances of that class will see the effect of that change.

{LP} LearnProgramming .academy

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
    }
}
```


{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

Instance Variables

```java
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 vs. Instance Methods

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

Static Methods

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.

{LP} LearnProgramming .academy

Static Methods Example

```java
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();

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

Instance Methods

Instance methods can access instance methods and instance variables directly.

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

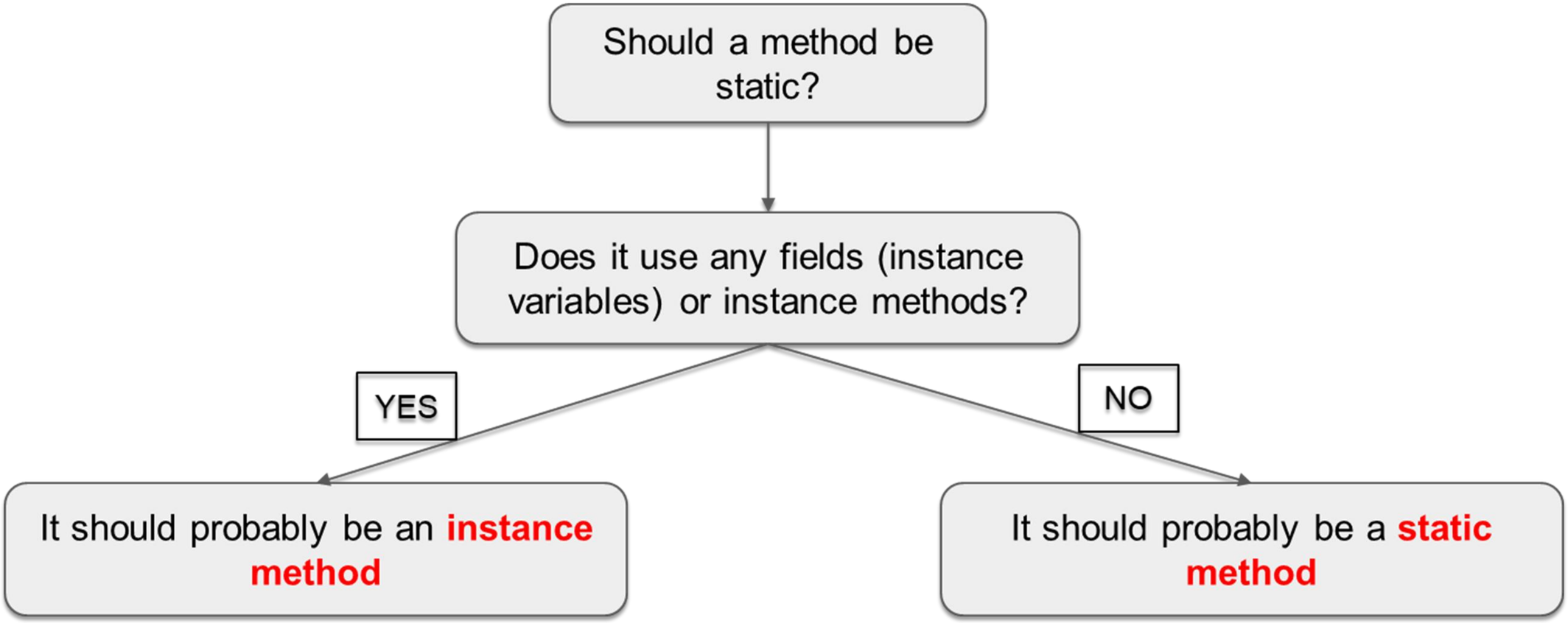
{LP} LearnProgramming .academy

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
    }
}
```


{LP} LearnProgramming .academy

Static or Instance Method



Plain Old Java Object

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 no other, or very few methods, other than getters and setters for the instance fields.

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 me to generate getters, setters, and constructors in a uniform way.

The Entity - The Student Table

Student
Id Name DateOfBirth ClassList

{LP} LearnProgramming .academy

Annotation

Annotations are a type of metadata.

Metadata is a way to formally describe 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 affect 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.

{LP} LearnProgramming .academy

The POJO vs. The Record

In the last video, I talked about the Plain Old Java Object, and I showed you how it comes with a lot of boilerplate code.

It's the code that's repetitive and follows certain rules.

Once created, this code is rarely looked at, or modified.

In fact, there are tools that'll just regenerate all of this code if your underlying data or domain model changes.

The Record type

The record was introduced in JDK 14 and became officially part of Java in JDK 16.

It's purpose is to replace the boilerplate code of the POJO but to be more restrictive.

Java calls them "plain data carriers".

The Record type

The record is a special class that contains data that's not meant to be altered.

In other words, it seeks to achieve immutability for the data in its members.

It contains only the most fundamental methods, such as constructors and accessors.

Best of all, you, the developer, don't have to write or generate any of this code.

Implicit or Generated Code that Java provides

What does Java tell us about what is implicitly created when we declare a record as I did in this code?

```
public record LPAStudent(String id, String name, String dateOfBirth, String classList) {}
```

First, it's important to understand that the part that's in parentheses, is called the record header.

The record header consists of record components, a comma-delimited list of components.

Implicit or Generated Code that Java provides

```
public record LPAStudent(String id, String name, String dateOfBirth, String classList) {}
```

For each component in the header, Java generates:

- A field with the same name and declared type as the record component.
- The field is declared private and final.
- The field is sometimes referred to as a component field.

Implicit or Generated Code that Java provides

```
public record LPAStudent(String id, String name, String dateOfBirth, String classList) {}
```

Java generates a toString method that prints out each attribute in a formatted String.

In addition to creating a private final field for each component, Java generates a public accessor method for each component.

This method has the same name and type of the component, but it doesn't have any kind of special prefix like get or is, for example.

The accessor method for id, in this example, is simply id().

Why have an immutable record?

Why is the record built to be immutable?

There are more use cases for immutable data transfer objects and keeping them well encapsulated.

You want to protect the data from unintended mutations.

POJO vs. Record

If you want to modify data on your class, you won't be using the record.

You can use the code generation options for the POJO as I showed in the earlier video.

But if you're reading a whole lot of records from a database or file source and simply passing this data around, then the record is a big improvement.

Java's new type, the record

I've only touched on some of the features of the record to give you an introduction.

When I do talk more about the final keyword and immutability of data, especially as it may be affected by concurrent threads, we'll be revisiting this type.

I'll also be showing it to you in action, when we get to the Database and I/O sections of this course.

Inheritance

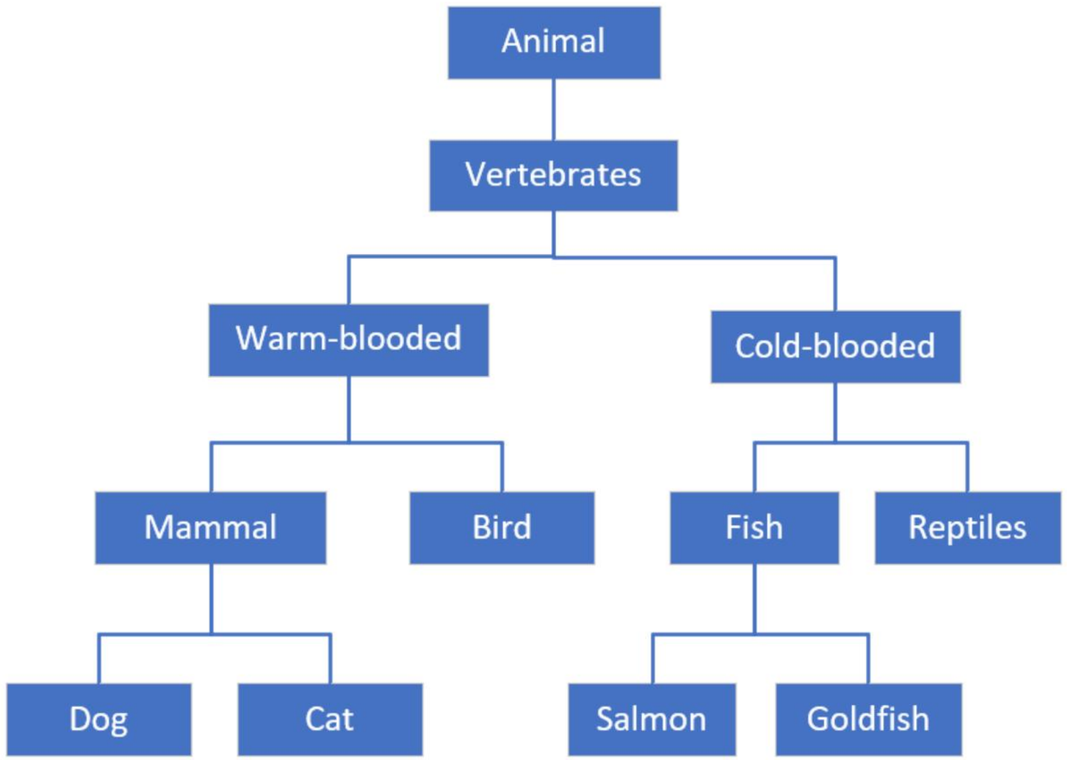
What is Inheritance, and why is it so powerful?

We can look at Inheritance as a form of code reuse.

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

{LP} LearnProgramming .academy

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.

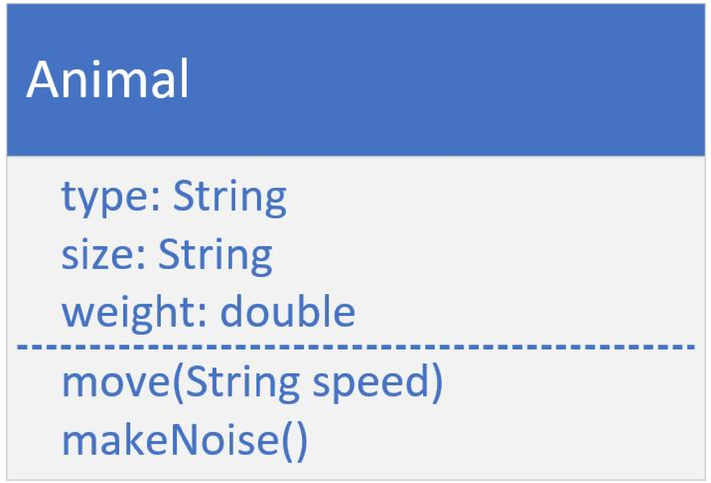
So Animal is the base class. All the other classes can be said to be subclasses of Animal.

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

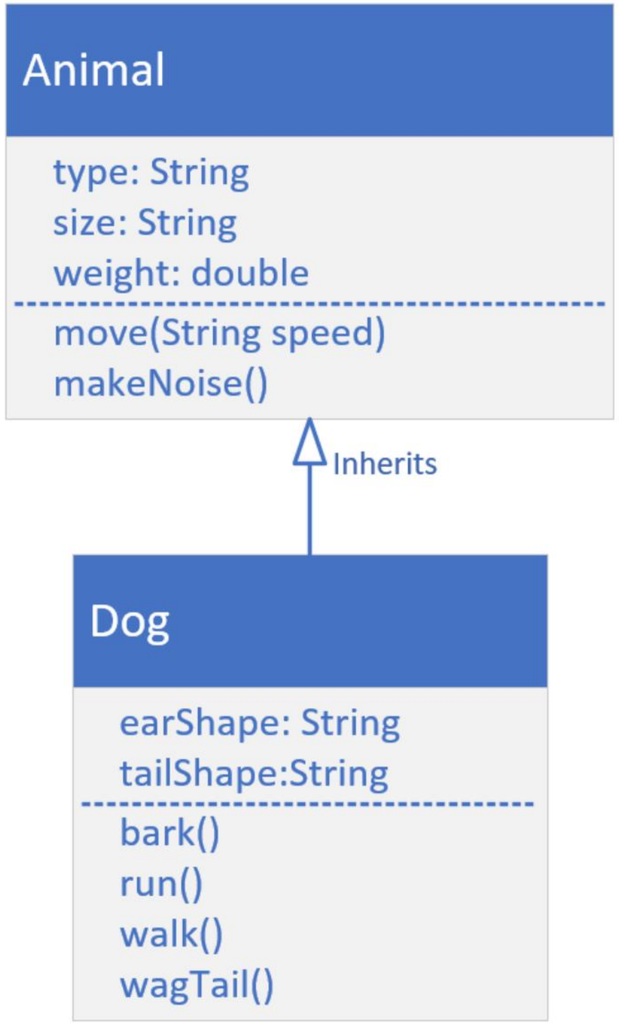


A class diagram allows us to design our classes before we build them.

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

Below the fields, I 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 I 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.

I 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.

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

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

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

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

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, then you must explicitly call super() in all of your constructors, passing the right arguments to that constructor.

Inheritance in Java

I introduced you to the concept of Inheritance and showed you how to implement it in Java using the **extends** keyword.

I talked about using the statement super() to call the constructor on the super class.

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

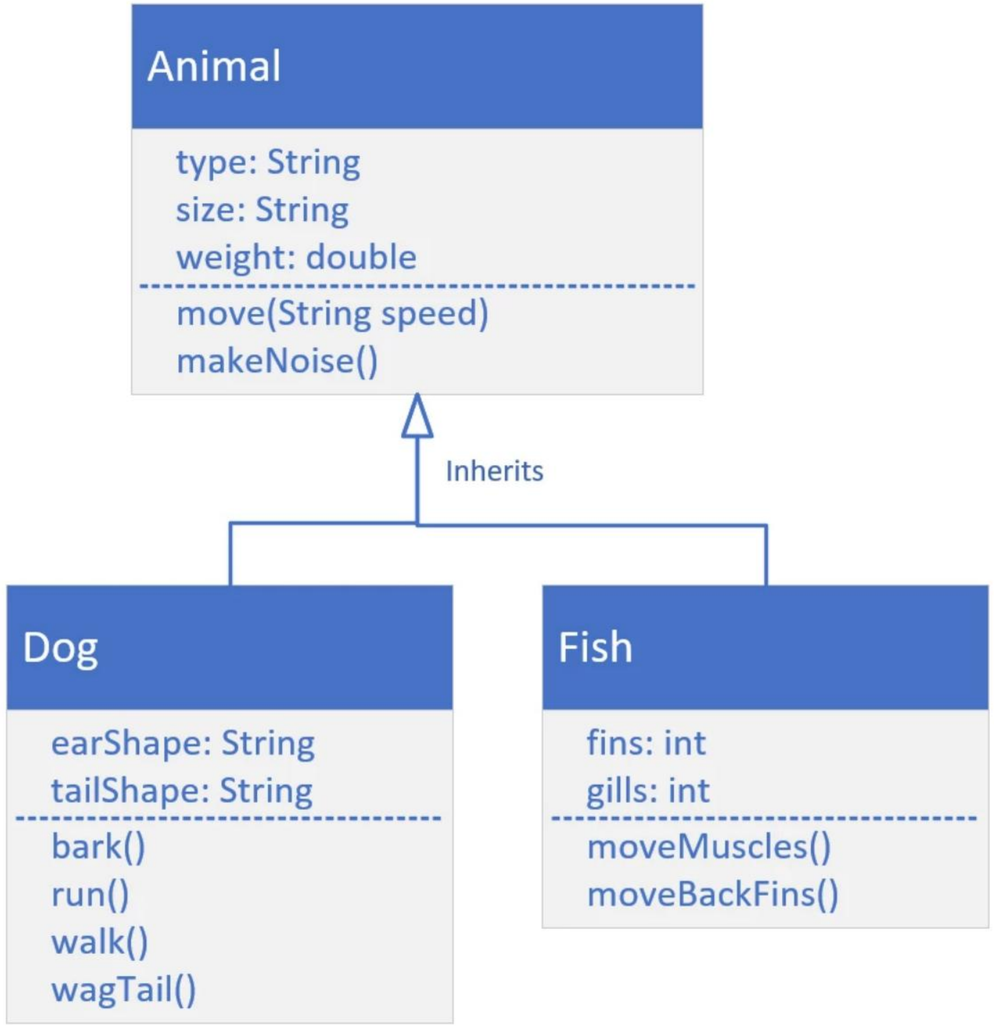
Overridden method

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 for Dog and Animal

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.

{LP} LearnProgramming .academy

Inheritance

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

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

{LP} LearnProgramming .academy

java.lang.Object

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

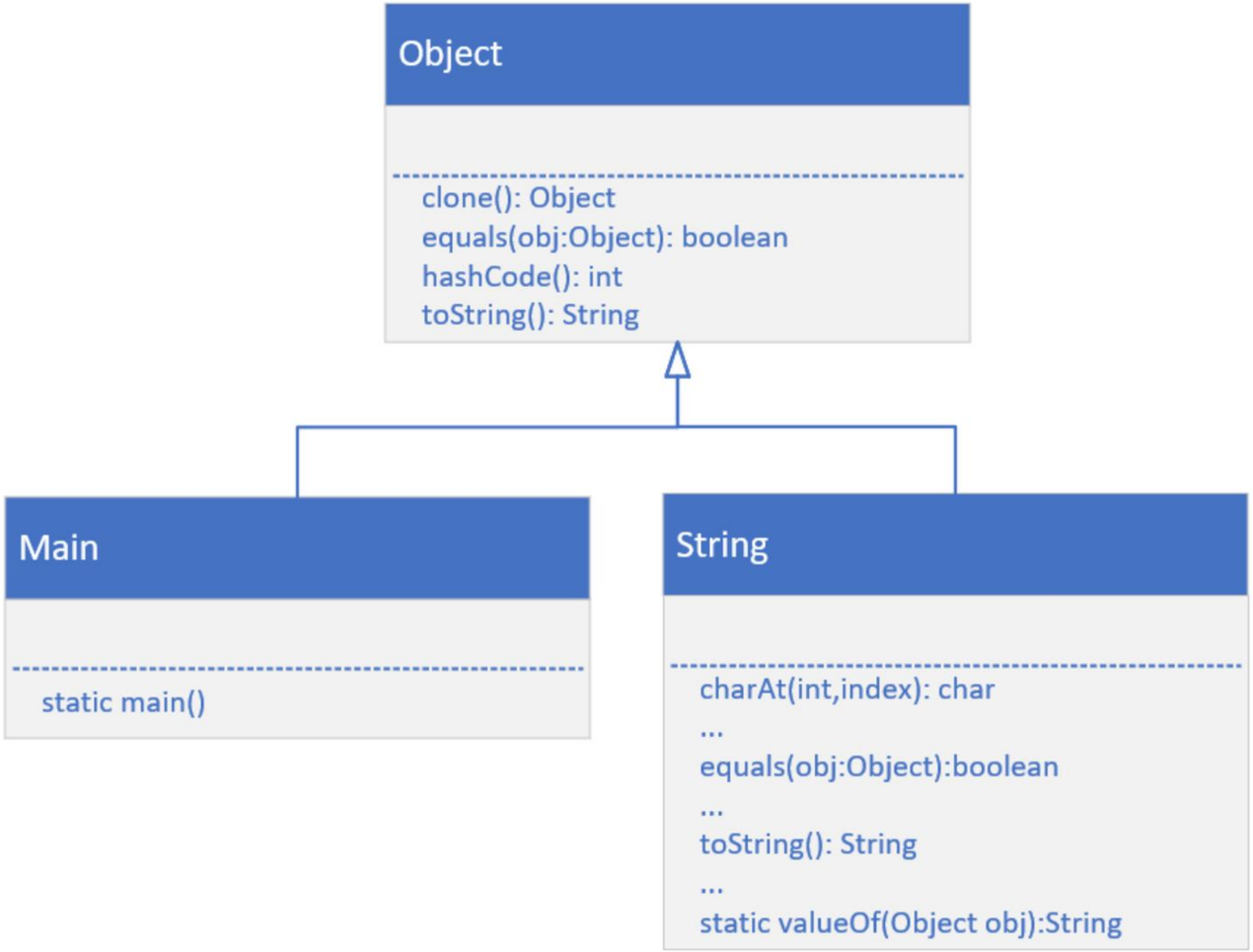
That class is named Object, and it's in the java.lang package.

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

Let's see what Java has to say about this class:

We'll use the link to Java's Application Programming Interface (API) for this class, which you can find in the resources section of this video.

Every Class inherits from Object



This slide shows that my Main class inherits from, or is a subclass of Object, as is also the case with String.

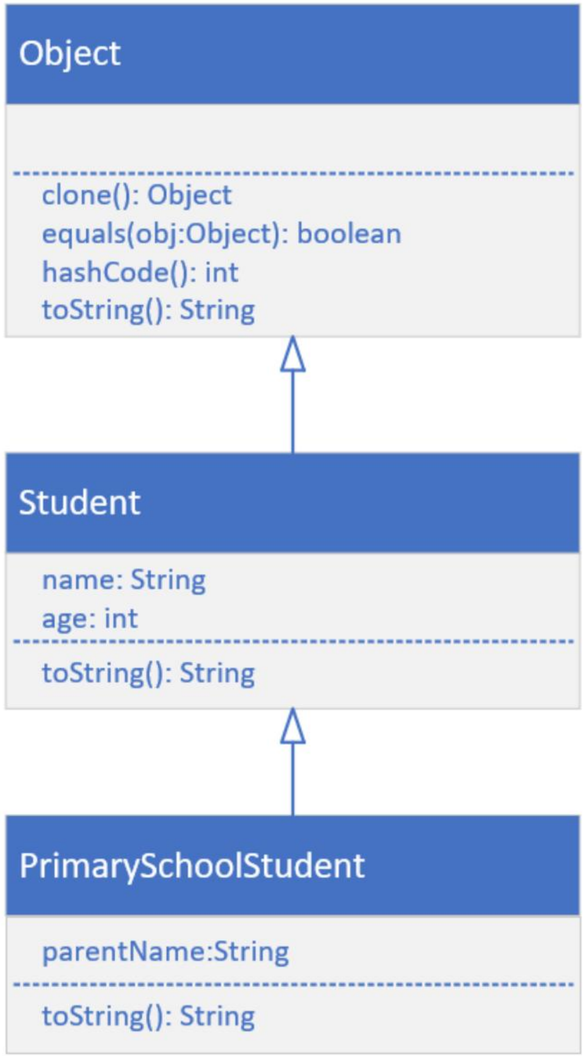
The String class has over 60 methods!

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

Class Diagram for Student and PrimarySchoolStud

I've already built Student which inherits from Object implicitly.

Next, I'll build PrimarySchoolStudent, which will inherit from Student.

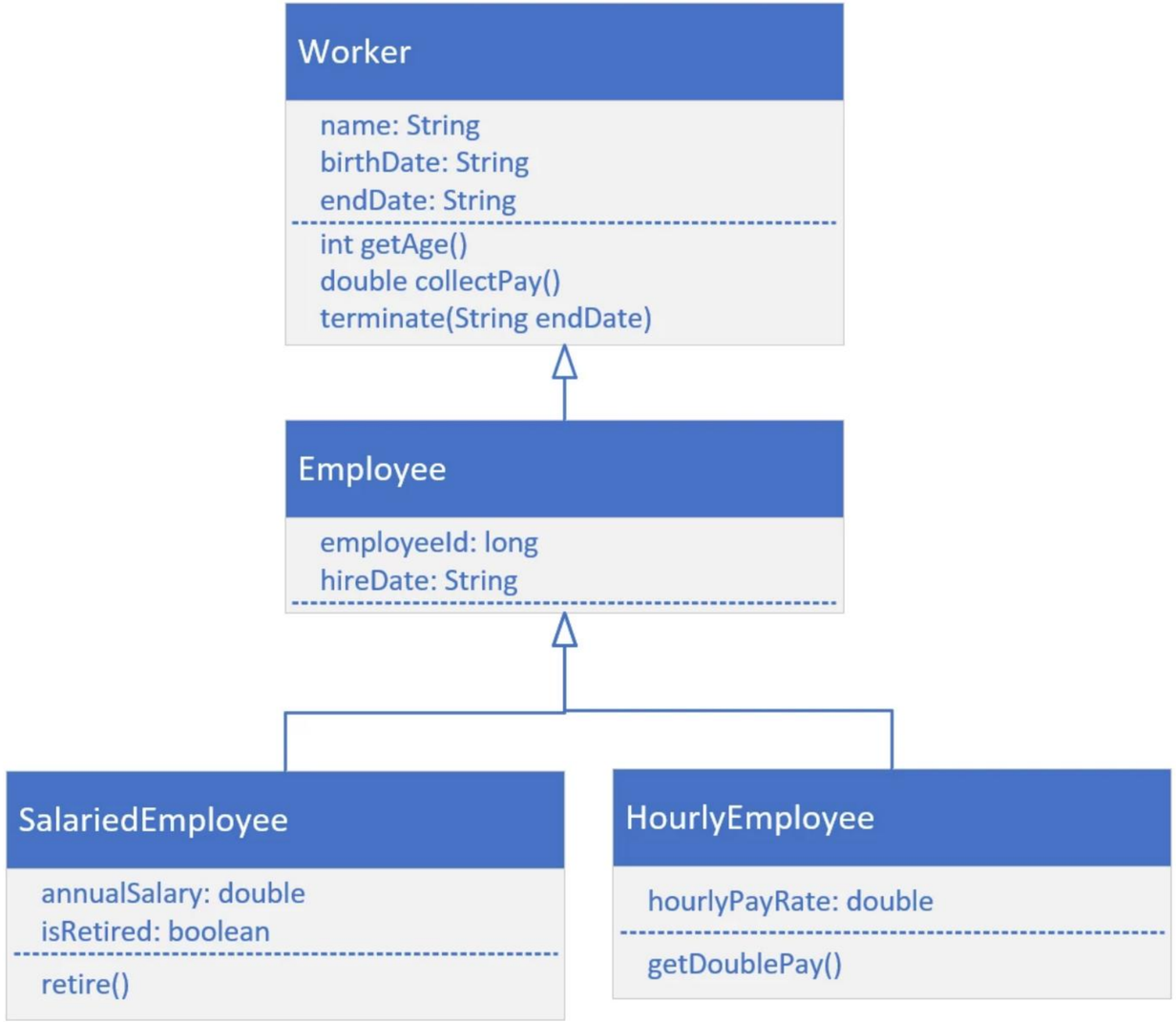


Inheritance Challenge

It's time now for a challenge, to solidify your understanding of what inheritance is.

For this challenge, I'm going to show you a class diagram like we worked with in previous videos.

Inheritance Challenge

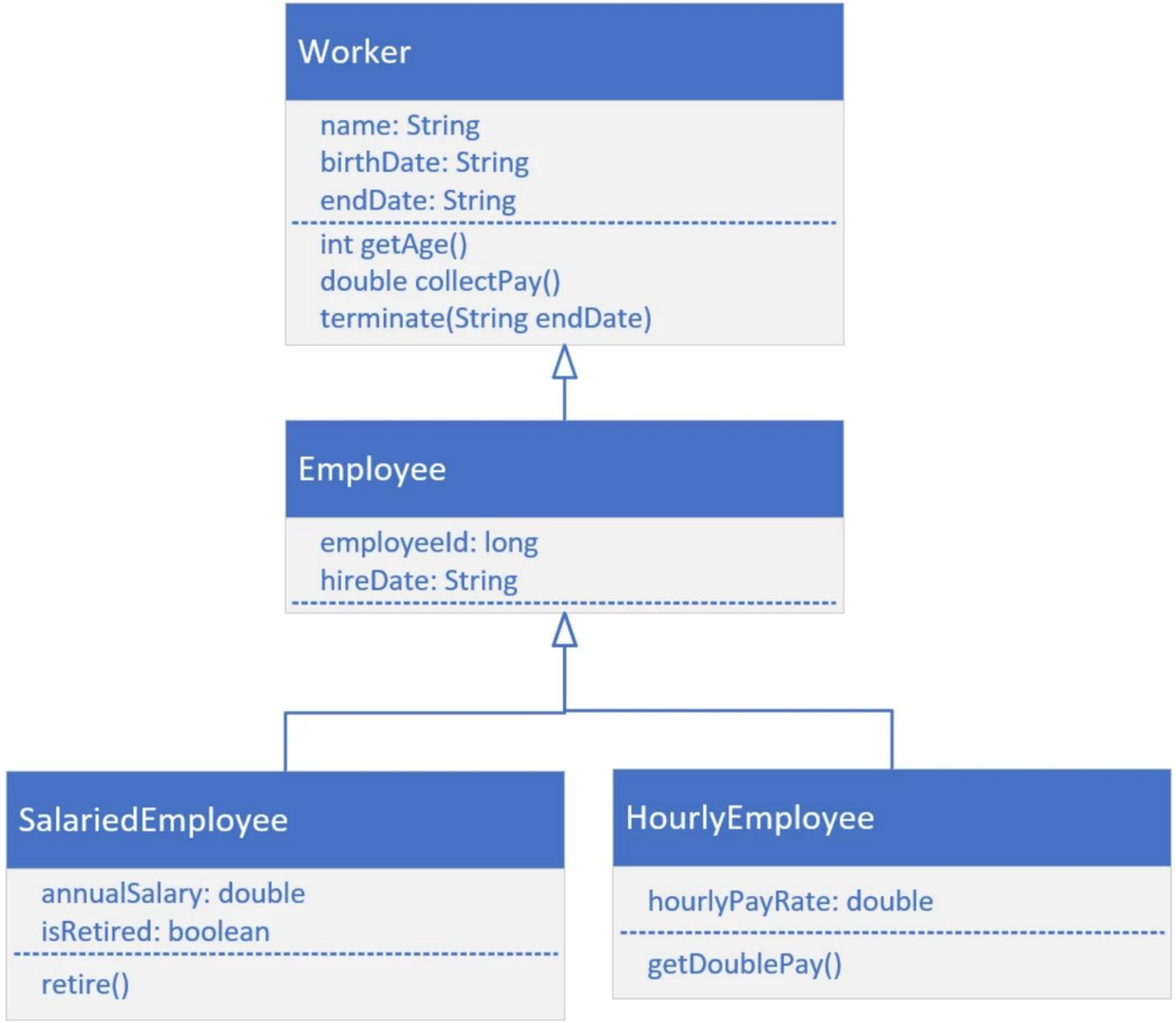


Your challenge is to create the Worker class, the Employee class, and either the SalariedEmployee or the HourlyEmployee class.

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().

Worker class

Worker

name: String
birthDate: String
endDate: String

---

int getAge()
double collectPay()
terminate(String endDate)

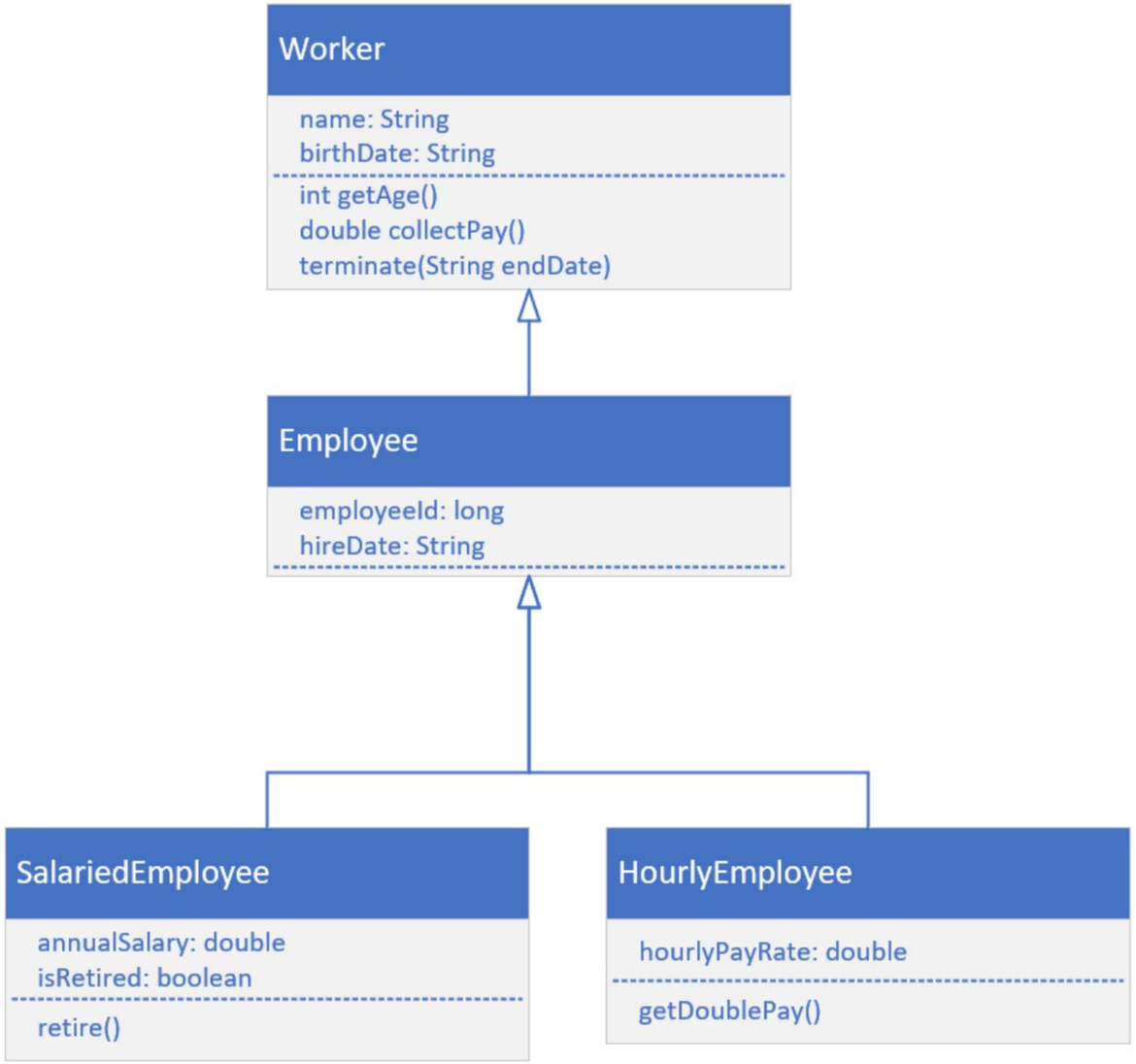
{LP} LearnProgramming .academy

the Employee Class

For this class, I have specific Employee attributes, employeeId, and hireDate.

For simplicity's sake, I haven't included any methods specific to an Employee.

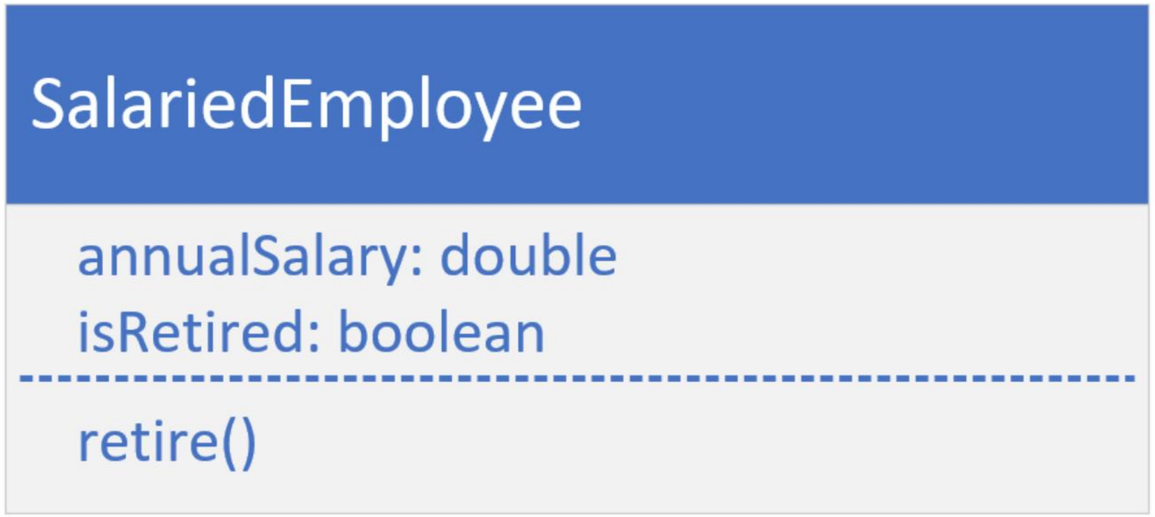
Inheritance Challenge, Continued



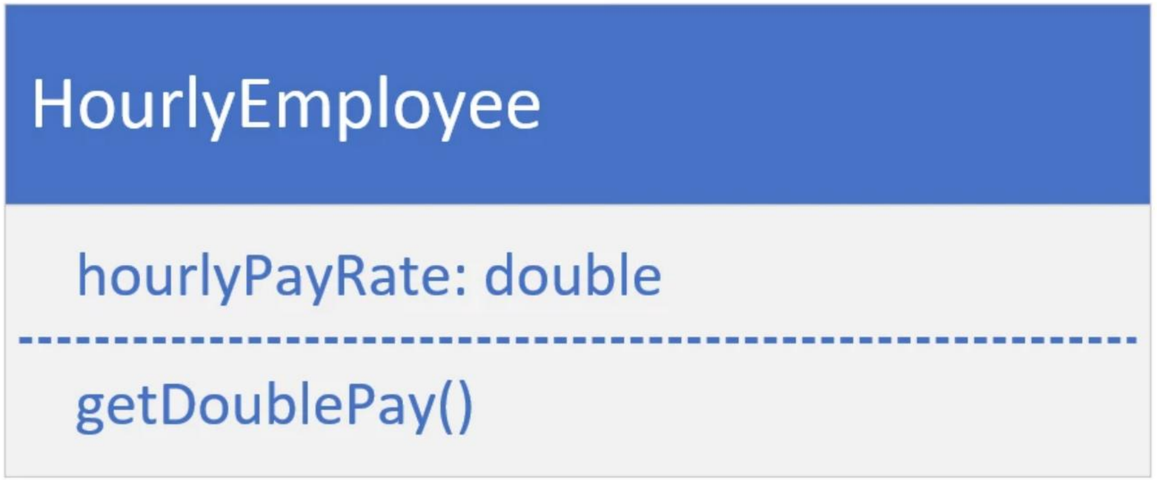
It's time to build a more specific type of Employee, one that's Salaried or one that's Hourly.

- A salaried employee is paid based on some percentage of his or her annual salary.
- If this person is retired, then the salary may be 100 percent of this amount, 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 number of hours.

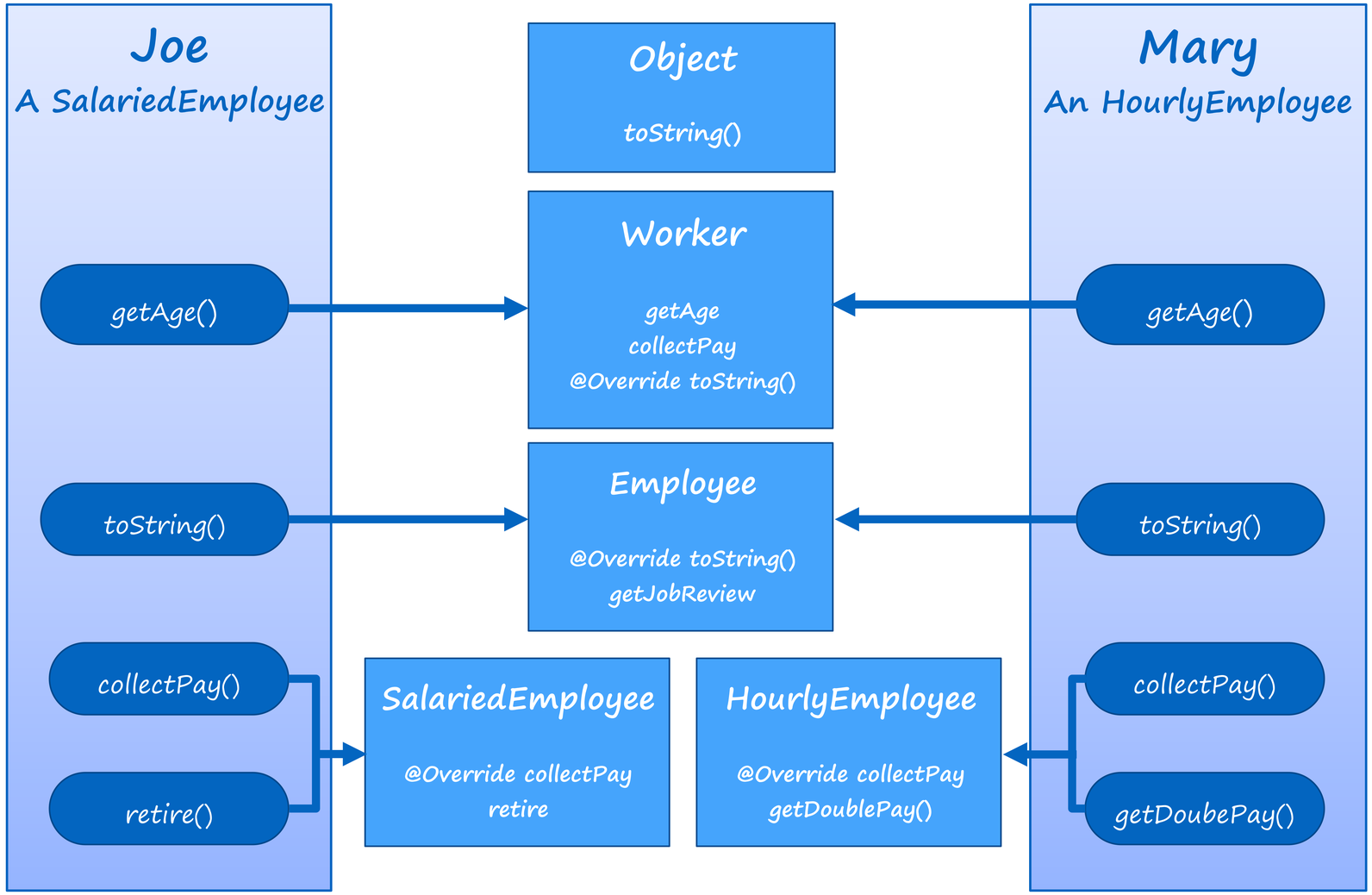
SalariedEmployee



HourlyEmployee class



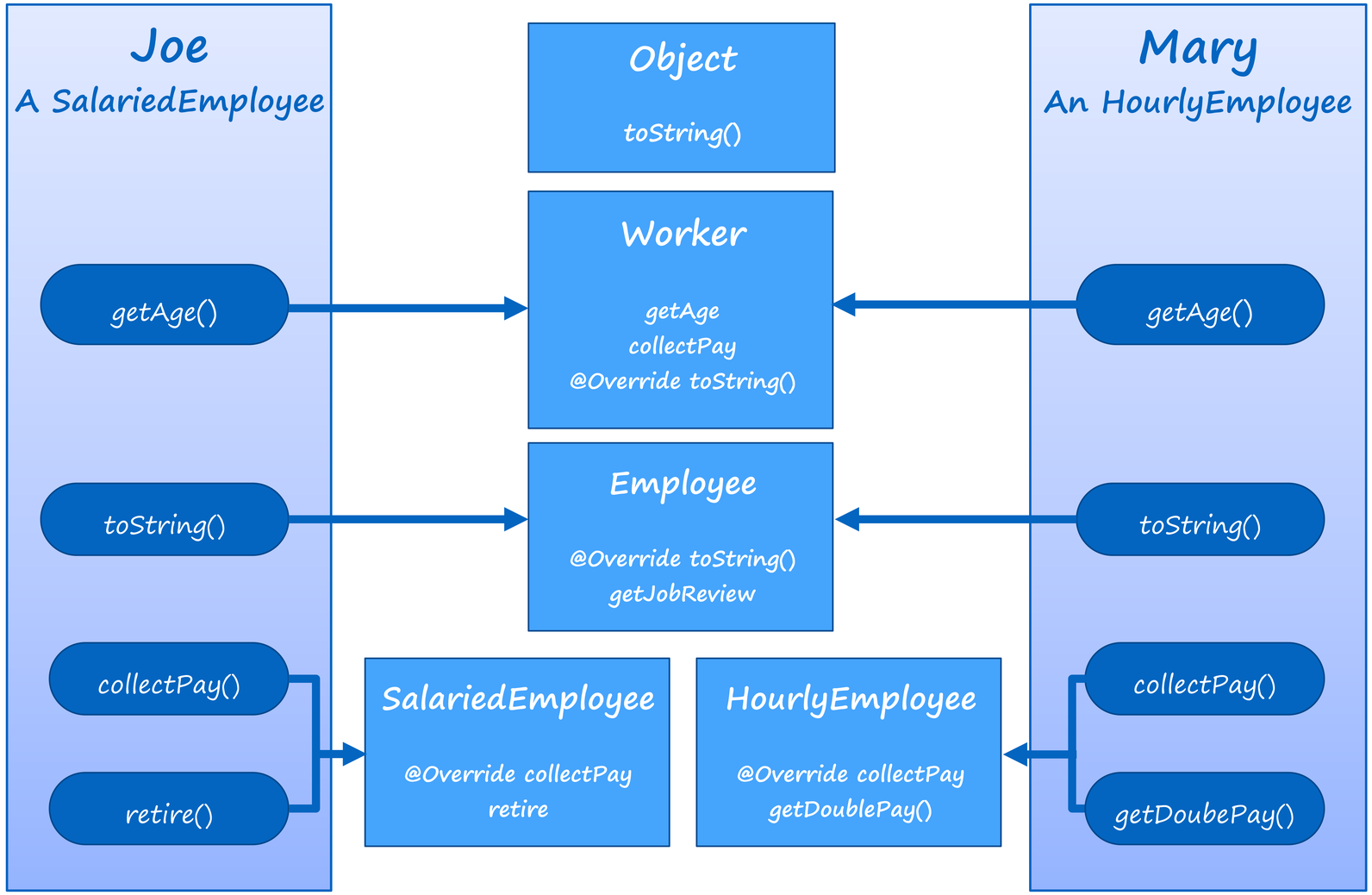
Making the call



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

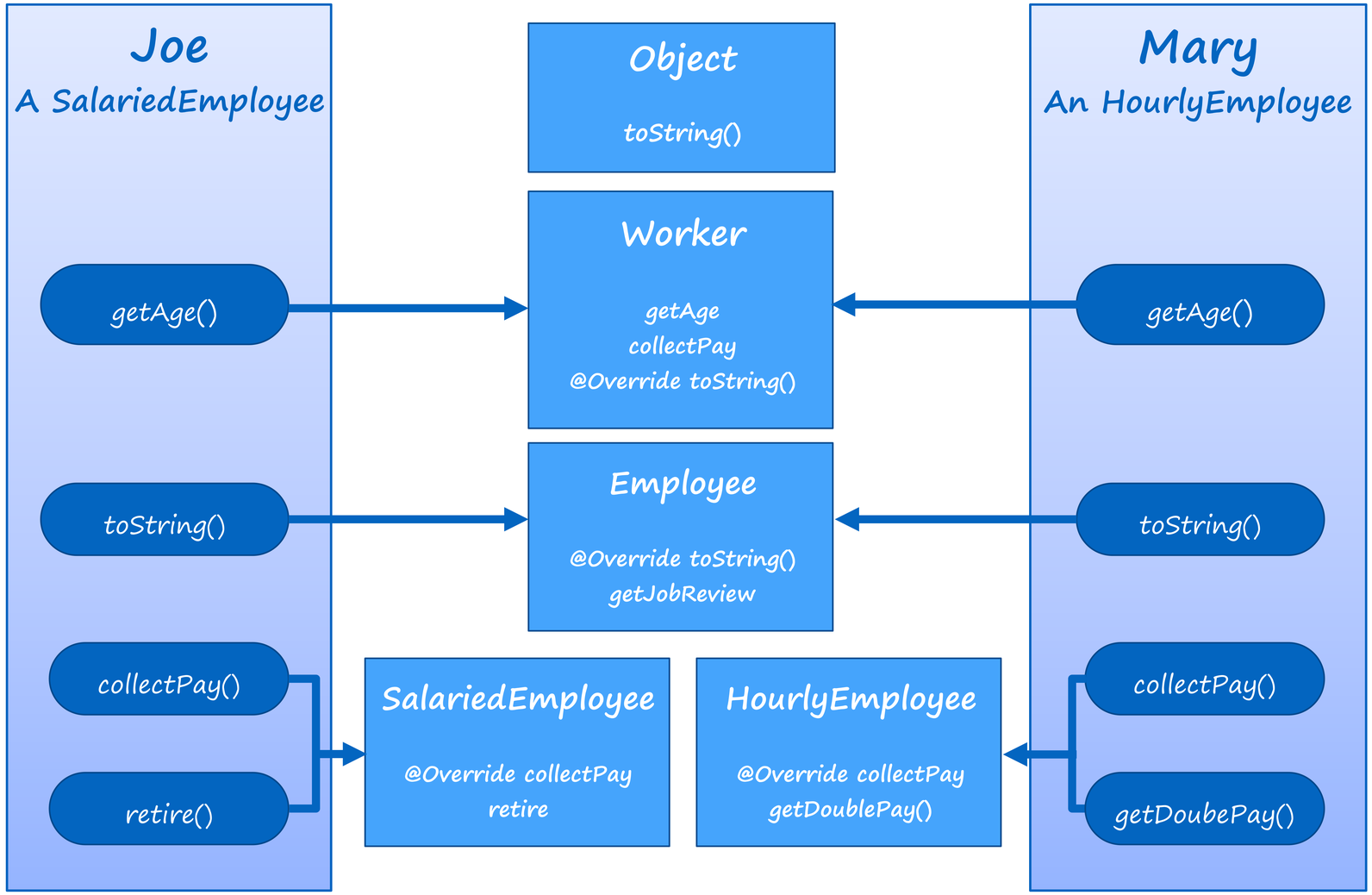
When Joe or Mary call getAge(), the method's implementation is on Worker and is not overridden by any other class, so the getAge method on Worker is executed.

Making the call



When Joe or Mary 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

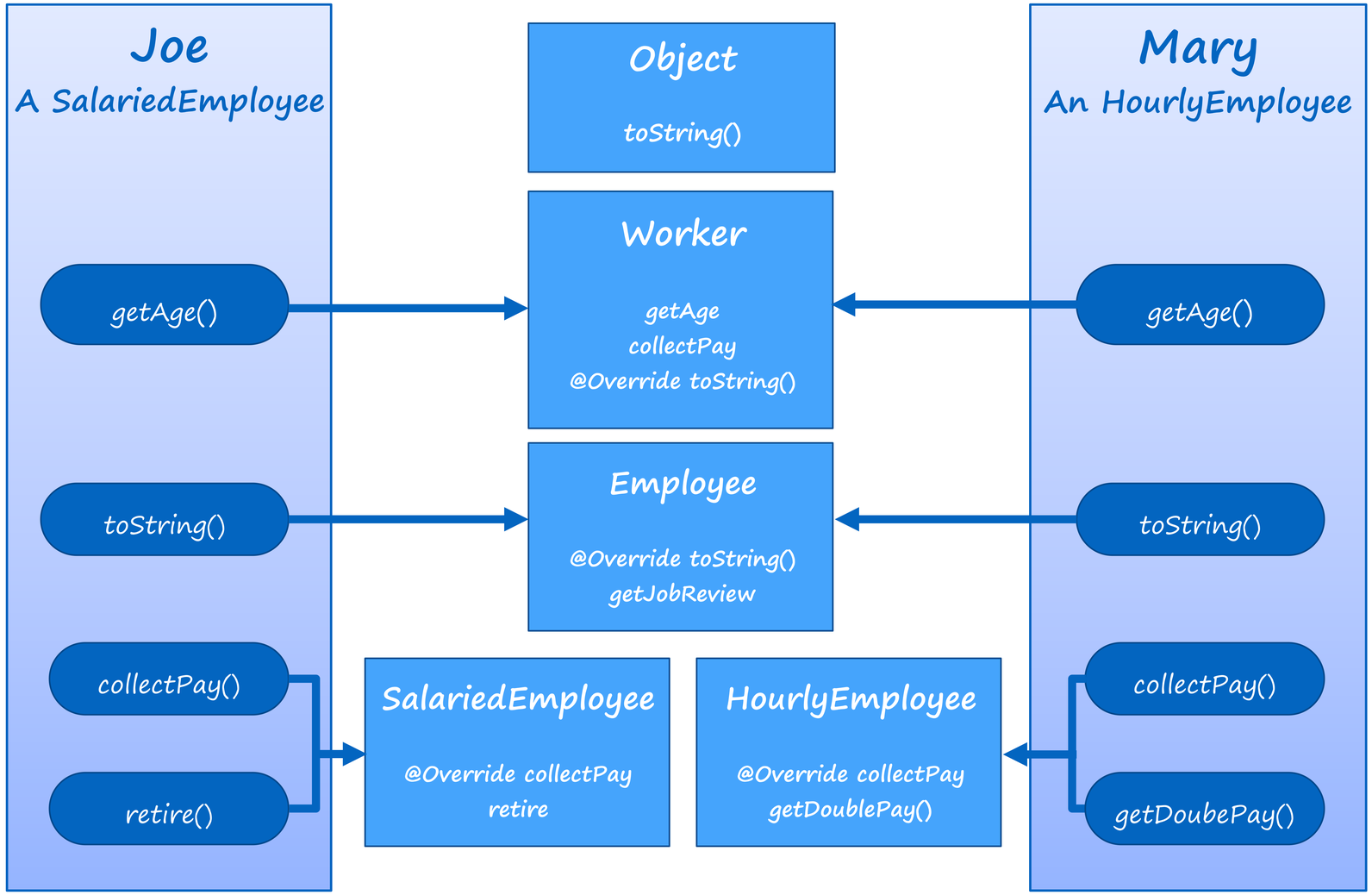


Looking at the **collectPay** method, this method was overridden by both SalariedEmployee, and HourlyEmployee.

Joe will execute the method on SalariedEmployee.

Mary will execute the one on HourlyEmployee.

Making the call



SalariedEmployee has a method, **retire**, that's not overridden, meaning it's only in 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 in this class and not in any super class.

this vs super

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

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

{LP} LearnProgramming .academy

this vs super

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

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

The keyword **this** is used to call the current class members (both 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 these two keywords anywhere in a class except for static elements such as a static method. Any attempt to do so will lead to compile time errors.

Keyword this

```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 within constructors and setters and is optionally used within getters.

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

In the getter, I don't have any parameters so there's no conflict. Therefore, the use of **this** is optional there.

Keyword super

```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, I have a method called **printMethod** that calls **super.printMethod**.

this() vs super() call

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

These are known as calls since they look like regular method calls 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.

{LP} LearnProgramming .academy

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, I have three constructors.

All three constructors initialize variables.

There's repeated code in each constructor.

I'm 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

```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;
    }
}
```

In this example, I 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 I 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, I have a class **Shape**, with x and y instance variables, and 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, I have both the **super()** and **this()** calls.

{LP} LearnProgramming .academy

Method Overriding vs Overloading

Let's review the **main differences between method overriding and method overloading.**

{LP} LearnProgramming .academy

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.

{LP} LearnProgramming .academy

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 parameters**).

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

OVERRIDING

same name
same parameters

```java
class Dog {

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

class GermanShepherd extends Dog {

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

OVERLOADING

same name
different parameters

```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");
        }
    }
}
```


{LP} LearnProgramming .academy

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.

Covariant Return Type

The return type of an overridden method can be the same type as the parent method's declaration.

But it can also be a subclass.

The term, covariant return type, is more appropriate.

{LP} LearnProgramming .academy

Covariant Return Type

I briefly mentioned in a previous video that there's a clone method on the class Object that all classes inherit from.

A simplified look at this declaration, for our purposes, is shown below.

```
protected Object clone() throws CloneNotSupportedException
```

And if you overrode this method by using IntelliJ's code generation tools, it would generate this code in your class:

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

Covariant Return Type

But in general, when you're cloning an instance, you're going to want to return an Object that's the same type as the Object you are cloning.

Remember, I said all classes ultimately have Object as a base class, so every class can be said to be a covariant of Object.

Covariant Return Type

The clone method (generated by IntelliJ) override (from Object)

```
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

The clone method overridden in a Person class.

This is a valid override of Object's clone method.

```
class Person {

    private String name;
    private String birthDate;

    public Person(String name, String birthDate) {
        this.name = name;
        this.birthDate = birthDate;
    }

    @Override
    public Person clone() {
        return new Person(name, birthDate);
    }
}
```


{LP} LearnProgramming .academy

What's a Text Block?

A Text Block is just a special format for **multi-line** String literals.

It's simply a **String** with a new representation in the source code.

It became part of the official language as of **JDK 15**.

Some Common Escape Sequences

An escape sequence starts with a backslash. Java has several, but the most common ones are shown on this slide.

These include the tab, a newline, a double quote character, or a backslash character if you need it in your text.

Escape Sequence	Description
\t	Insert a tab character.
\n	Insert a new line character.
\"	Insert a double quote character.
\\	Insert a backslash character.

Format Specifiers

At their most complex, format specifiers take the form shown here.

%[argument_index$][flags][width][.precision]conversion

They start with a percent sign, and end with a conversion symbol, and have lots of options in between.

I've looked at one conversion type, d, which is the symbol used for a decimal integer value.

The String

The String has **over 60 methods** available.

The String is a sequence of characters, meaning its characters are ordered and indexed.

The index starts at 0 and not 1.

The table below shows the indices above each character for the String "Hello

Index	0	1	2	3	4	5	6	7	8	9	10
Character	H	e	l	l	o		W	o	r	l	d

We can say the character "H" is at index 0, and "W" is at index 6.

The length of this String is 11, but its last index is 10.

The purpose of String methods

We can split String's methods up into three basic categories:

- String Inspection Methods.
- Methods for Comparing String values.
- String Manipulation Methods.

Let's start with a look at some of the String Inspection Methods.

{LP} LearnProgramming .academy

String Inspection Methods

method	description
length	Returns the number of characters in the String.
charAt	Returns the character at the index that's passed.
indexOf lastIndexOf	Returns an integer, representing the index in the sequence where the String or character passed, can be located in the String.
isEmpty	Returns true if length is zero.
isBlank	Returns true if length is zero OR the string only contains whitespace characters, added in JDK 11.

String Comparison Methods

method	description
contentEquals	Returns a boolean if the String's value is equal to the value of the argument passed. This method allows for arguments other than String, for any type that is a character sequence.
equals	Return a boolean if the String's value is equal to the value of the argument passed.
equalsIgnoreCase	Return a boolean if the String's value is equal (ignoring case), to the value of the argument passed.

String Comparison Methods

method	description
contains	Returns a boolean if the String contains the argument passed.
endsWith startsWith	These return a boolean, and are much like the contains method, but more specific to the placement of the argument in the String.
regionMatches	Returns a boolean, if defined sub-regions are matched.

String Comparison Methods

The first set of methods don't actually change the underlying meaning of the text value but performs some kind of clean up.

method	description
indent	This method was added in JDK 15, and adds or removes spaces from the beginning of lines in multi-line text.
strip stripLeading stripTrailing trim	The difference between the strip method and trim method is that the strip() supports a larger set of white space characters. It and the corresponding stripLeading and stripTrailing methods were added in JDK 11.
toLowerCase toUpperCase	Returns a new String, either in a lower case or in upper case

The second set of string manipulation methods transforms the String value and returns a String with a different meaning than the original String.

method	description
concat	Similar to the plus operator for strings, it concatenates text to the String and returns a new String as the result.
join	Allows multiple strings to be concatenated together in a single method, specifying a delimiter.
repeat	Returns the String repeated by the number of times specified in the argument.
replace replaceAll replaceFirst	These methods replace characters or strings in the string, returning a new String with replacements made.
substring subSequence	These return a part of the String, its range defined by the start and end index specified.

String vs StringBuilder

Java provides a mutable class that lets us change its text value.

This is the StringBuilder Class.

{LP} LearnProgramming .academy

Creating Instances

Instantiating String Objects

```
String hello = "Hello";
String helloWorld = "Hello" + " World";
String badHello = new String("Hello"); // Valid code, but redundant
```

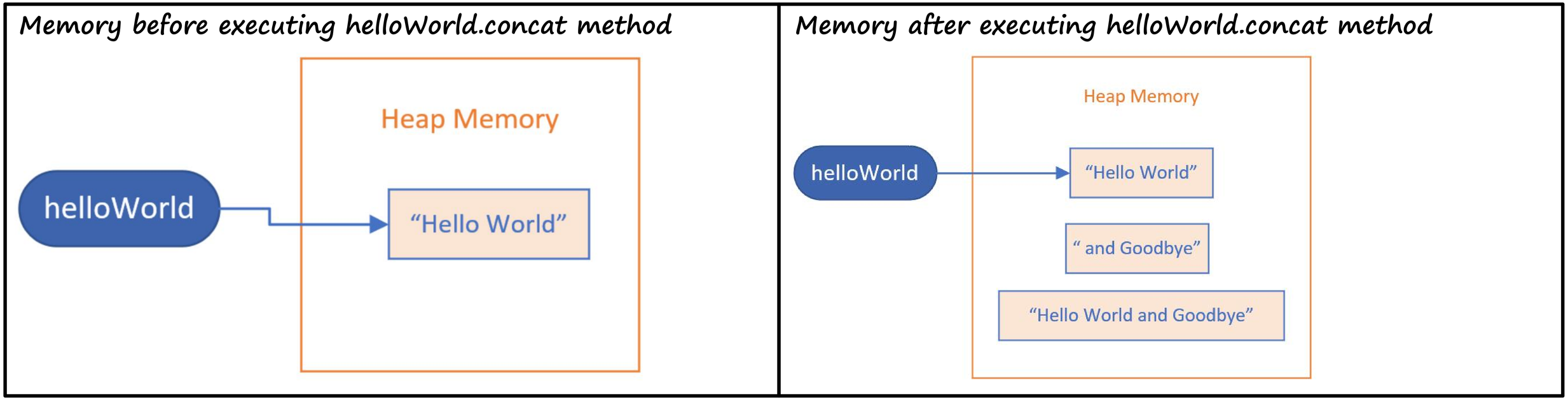
Instantiating StringBuilder Objects

```
StringBuilder helloBuilder = new StringBuilder("Hello");
StringBuilder emptyBuilder = new StringBuilder();
StringBuilder emptyBuilder5 = new StringBuilder(5);
StringBuilder stringBuilder = new StringBuilder(helloBuilder);
```

There are four ways to create a new StringBuilder object using the new keyword:

- Pass a String.
- Pass no arguments at all.
- Pass an integer value.
- Pass some other type of character sequence (like StringBuilder).

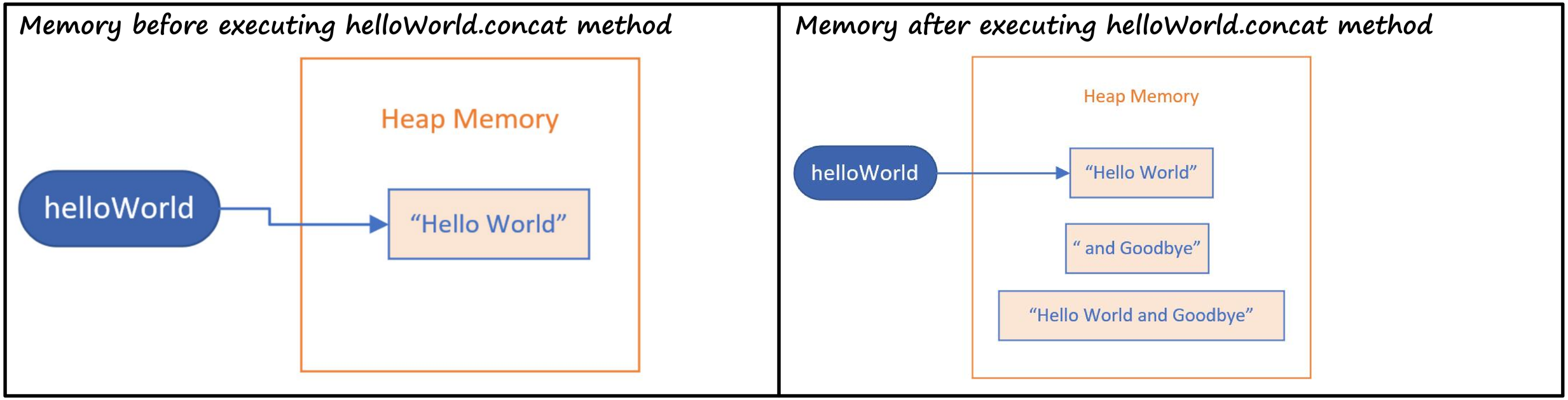
String



When I passed the String literal, "and Goodbye", to the concat method, this created an Object in memory for that literal, "and Goodbye".

It also created the result of the concat method, the object, the String, that has the value, "Hello World and Goodbye".

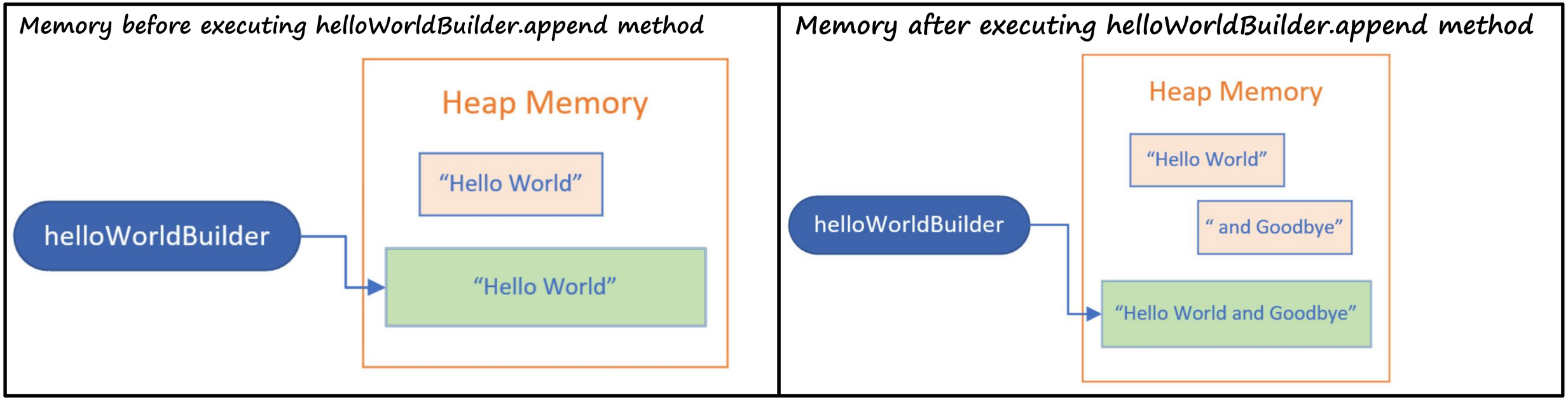
String



These methods don't change the internals of the existing String object.

The String referenced by the helloWorld variable never changed, instead a new String was created by the method call.

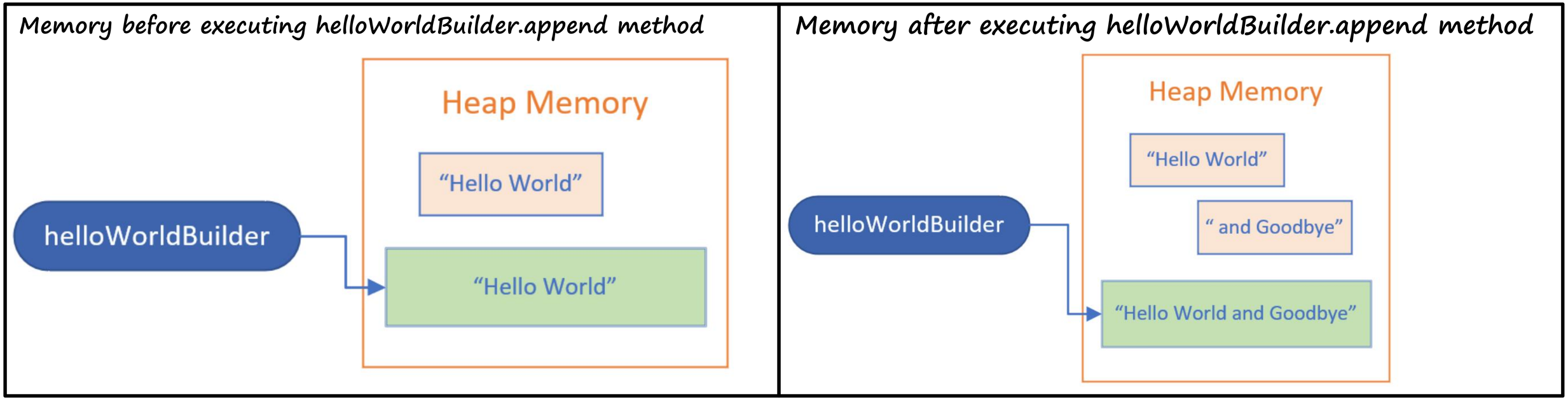
StringBuilder



On this slide Strings and StringBuilders in different colors, with the StringBuilder object in green.

After the call to the append method, I still only have one StringBuilder object.

StringBuilder



The variable helloWorldBuilder is still referencing the same object, but the value of that object changed.

This is important because it means the character sequence in the StringBuilder changed.

String methods vs. StringBuilder methods

String methods create a new object in memory and return a reference to this new object.

StringBuilder methods return a StringBuilder reference, but it's really a self-reference.

{LP} LearnProgramming .academy

Some methods unique to the StringBuilder class

A StringBuilder class has many similar methods to Strings.

But it also has methods to remove and insert characters or Strings. In addition, you can truncate the string builder's size.

method	description
delete deleteCharAt	You can delete a substring using indices to specify a range, or delete a single character at an index.
insert	You can insert text at a specified position.
reverse	You can reverse the order of the characters in the sequence.
setLength	setLength can be used to truncate the sequence, or include null sequences to 'fill out' the sequence to that length.