NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 3: 13th Feb 2024

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.

Class-based Programming

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

But what does this really mean?

So, to start, we need to understand what objects are.

They're really the key to understanding this Object Oriented terminology.

Real World Object Exercise

So 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

Now, Real world objects have two major components:

- state
- and behavior

State (computer)

So 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)

We 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:

1. Eating
2. Drinking
3. Fighting
4. Carrying food, those types of things

State and Behavior

So modelling real world objects as software objects, is a fundamental part of Object Oriented Programming.

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

And Objects expose their behavior with methods, which we've talked about before.

So, where does a class fit in?

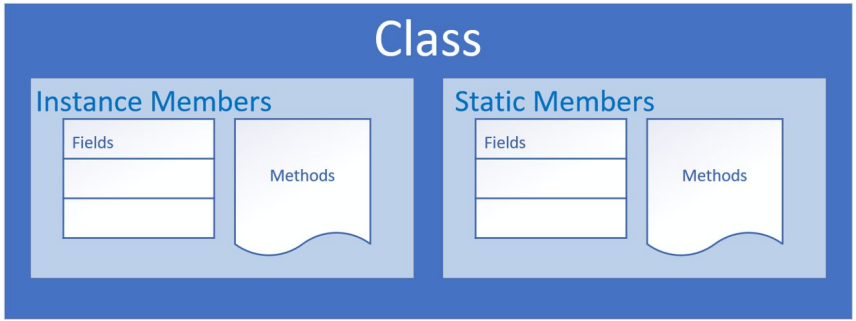
Well, think of a class as a template, or a blueprint for creating objects.

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.

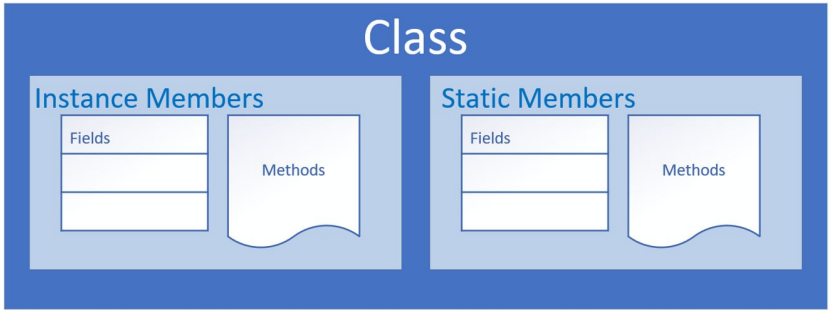
These are called class members.



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.

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.
protected	`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.

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

So, 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

Create a new class for a bank account.

Create fields for account characteristics like:

account number

account balance

customer name

email

phone number

Classes Challenge Exercise

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

Classes Challenge Exercise

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.

Classes Challenge Exercise

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

In addition, you'll have two behavioral methods, for depositing funds, and 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.

Constructor

A constructor is used in the creation of an object, that's an instance of a class.

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.

```
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.
    }
}
```

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 formal 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.

Constructor chaining with this()

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

You can call a constructor only from another constructor.

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.

Constructor Challenge Exercise

So for this challenge, you'll want to:

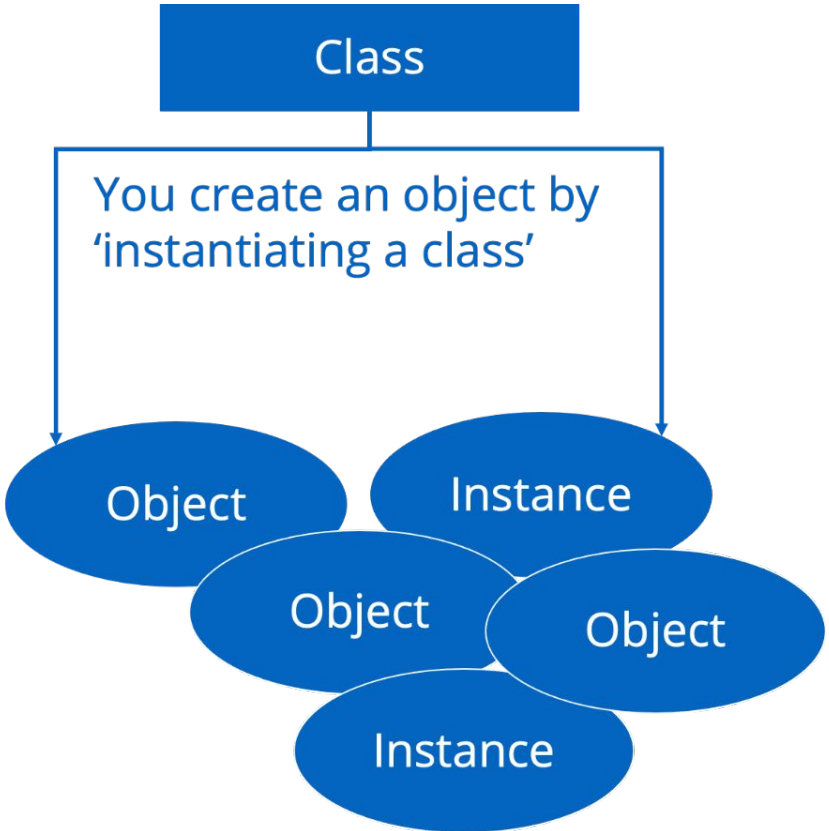
1. Create a new class, called Customer, with three fields:
    a. Name
    b. credit limit
    c. email address.
2. Create the getter methods only, for each field. You don't need to create the setters.

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



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 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.

Each house we build has an address (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.

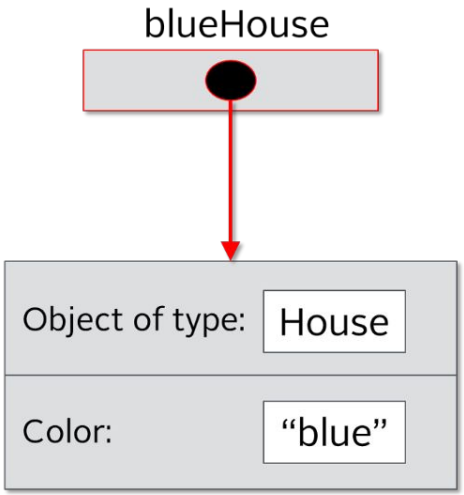
We can pass references as parameters to constructors and methods.

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
    }
}
```

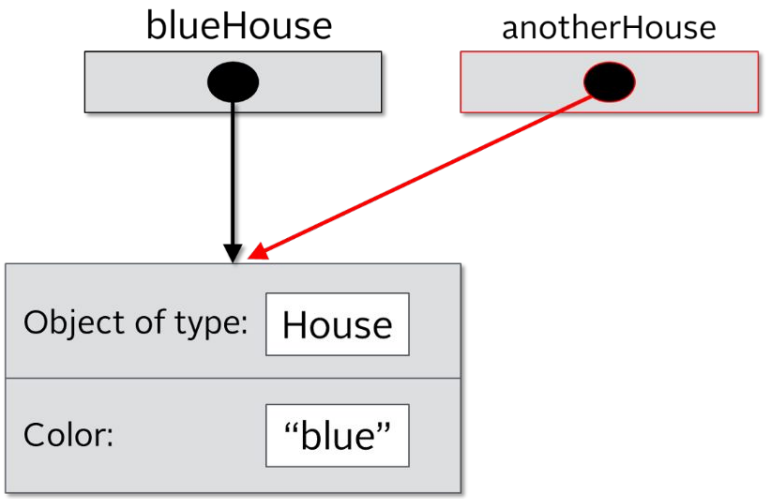
Reference vs Object vs Instance vs Class




```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.

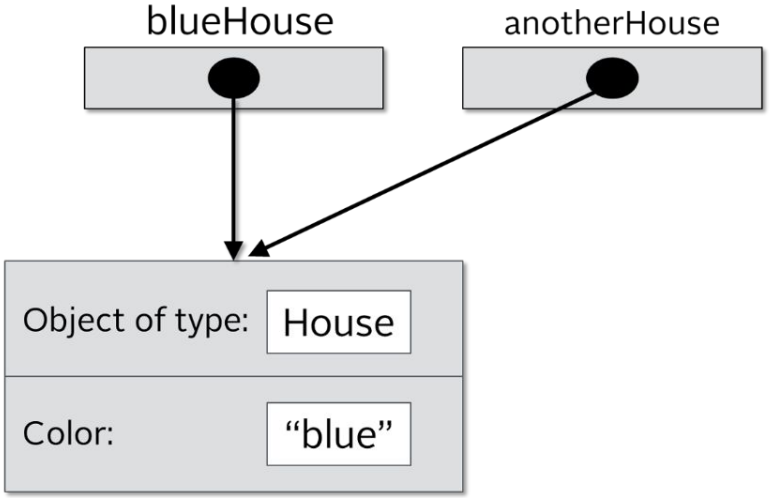
Reference vs Object vs Instance vs Class




```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).

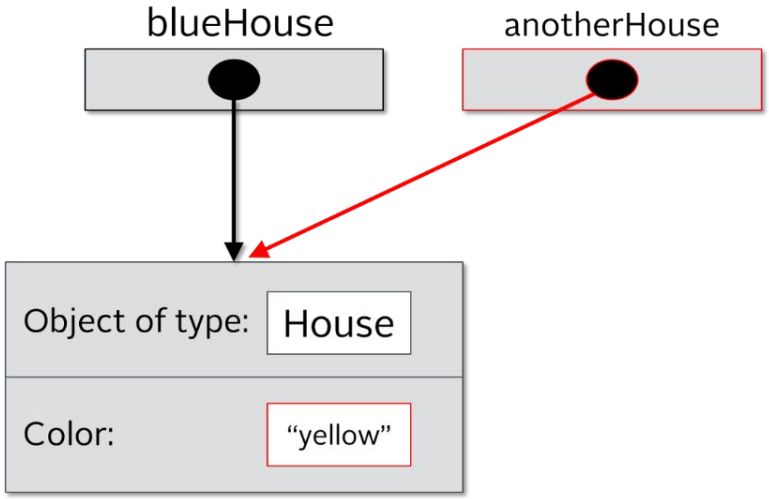
Reference vs Object vs Instance vs Class




```
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**.

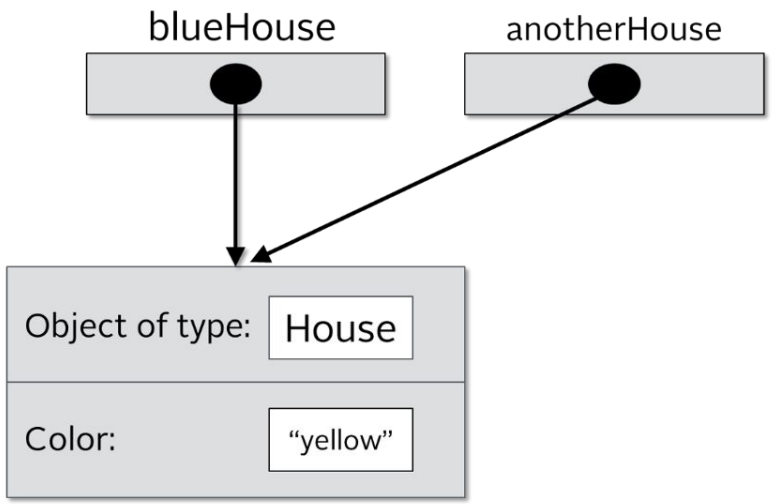
Reference vs Object vs Instance vs Class




```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.

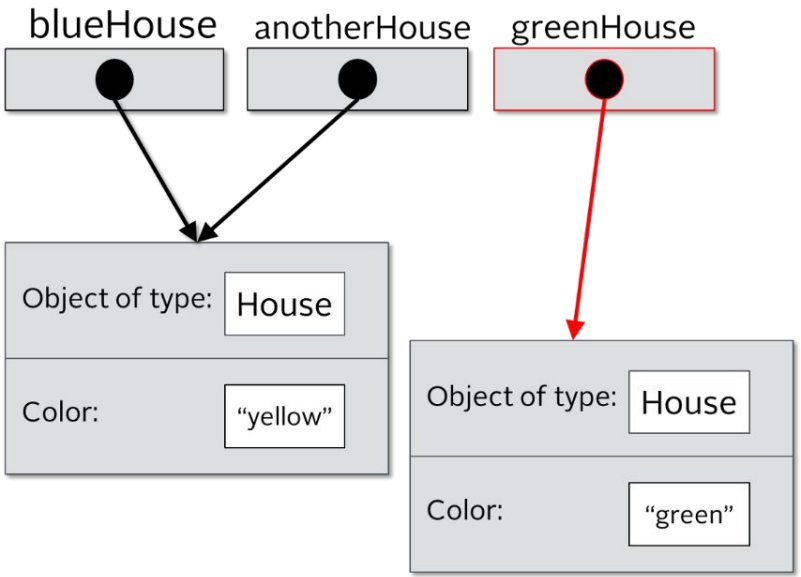
Reference vs Object vs Instance vs Class




```
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.

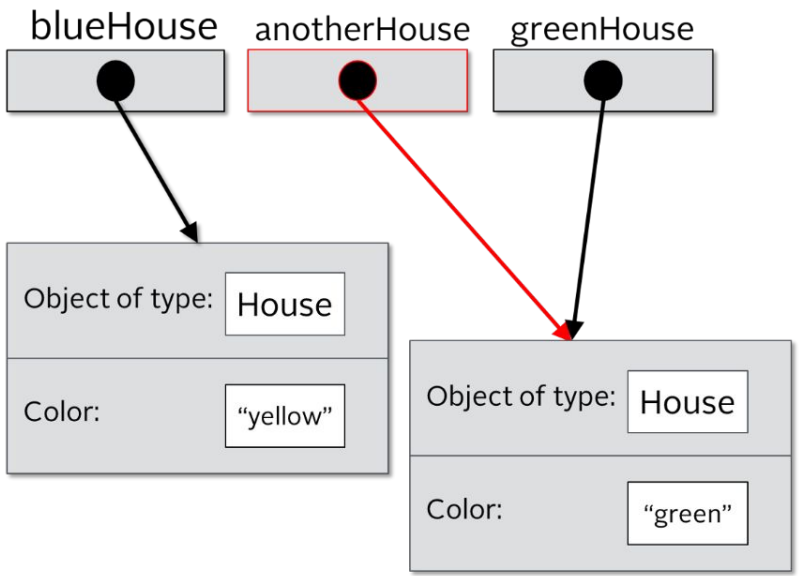
Reference vs Object vs Instance vs Class




```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.

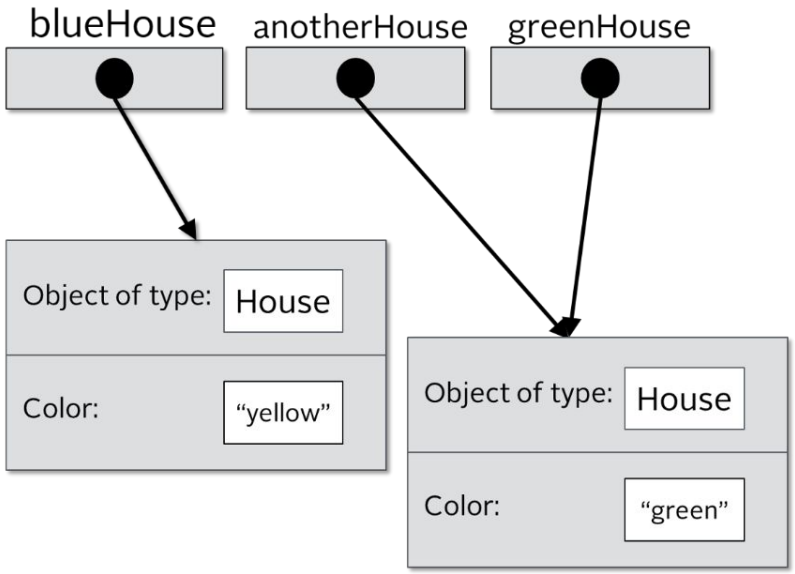
Reference vs Object vs Instance vs Class




```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, we create a new House, and make it red.

But we aren't 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, our code has no way to access it.

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

We didn't create a reference to it.

The reference vs The object

We create a reference to the house object we created.

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

Our reference, myHouse, lets us have access to that beige house, as long as our variable, myHouse, stays in scope.

The reference vs The object

We're creating a red house again, but this is a different object altogether, from the red house we 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 statement is creating another house object in memory, which has no relationship to the one we 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
```

So 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 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.

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