JAVA

constructor overloading:- same as method overloading
constructor chaining:- when one constructor calls another overloaded constructor
we can call a constructor from another constructor

need to use *this()* to execute another constructor, passing arguments if required
and *this()* must be the first executable statement if its used from another constructor

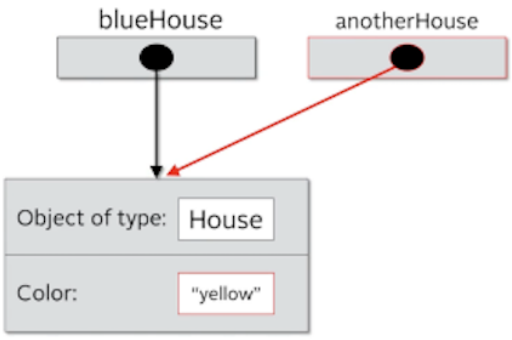
```
1 usage
Account(){
    this( firstName: "sam",  lastName: "jackson",  accountNumber: 111111111,  balance: 2000, age: 18, address: "123 address street");
    System.out.println("Empty Constructor called");
}
2 usages
Account(String firstName, String lastName, int accountNumber, int balance,
        int age, String address) {

    if(firstName == null || lastName == null || address == null) return;
    if(age < 18) return;

    this.firstName = firstName;
    this.lastName = lastName;
    this.accountNumber = accountNumber;
    this.balance = balance;
    this.age = age;
    this.address = address;
}
```

generally its not recommended to use getters and setter in constructor

references vs instances vs objects vs class

object is the instance of a class and reference is the address of an object

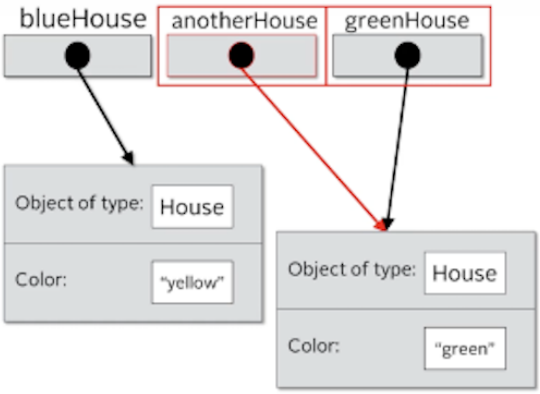
Reference vs Object vs Instance vs Class




```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()); // yed
        System.out.println(anotherHouse.getColor());// red

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

        System.out.println(blueHouse.getColor()); //yel
        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 anothe ouse the same color now. Why? Remember we have two **references** that point to the same **object** in memory. Once we char of one, **both references** still point to the same **object**. In our real world example, there is still just one physical house a address, even though we have written the same address on two pieces of paper.

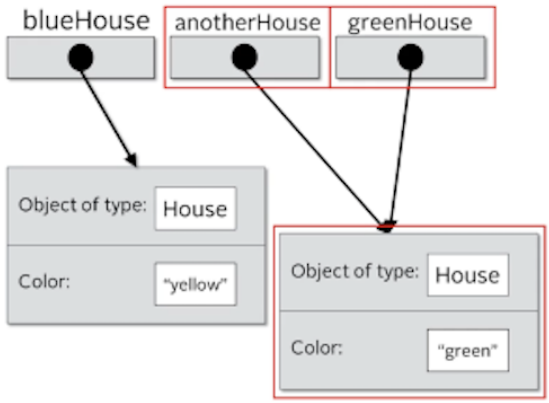
Reference vs Object vs Instance vs Class




```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 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

```java
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 first line is said to be eligible for garbage collection immediately after its execution.

Its no longer accessible.

Static vs Instance variables & methods

- declared with static keyword
- also known as static member variable
- every instance of a class shares the same static variable
- so if changes are made to that variable all the other instance will see the effect of that change
- Its best to use the class name to access the static variable rather than reference name:

  Class.StaticMember rather than Instance.StaticMember

- can be used for storing counters, generating unique ids, storing const values like pi, creating and controlling access to a shared resource like logs and db instance.

```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 confusin
    }
}

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 represent a specific state of a class.

Static Methods :

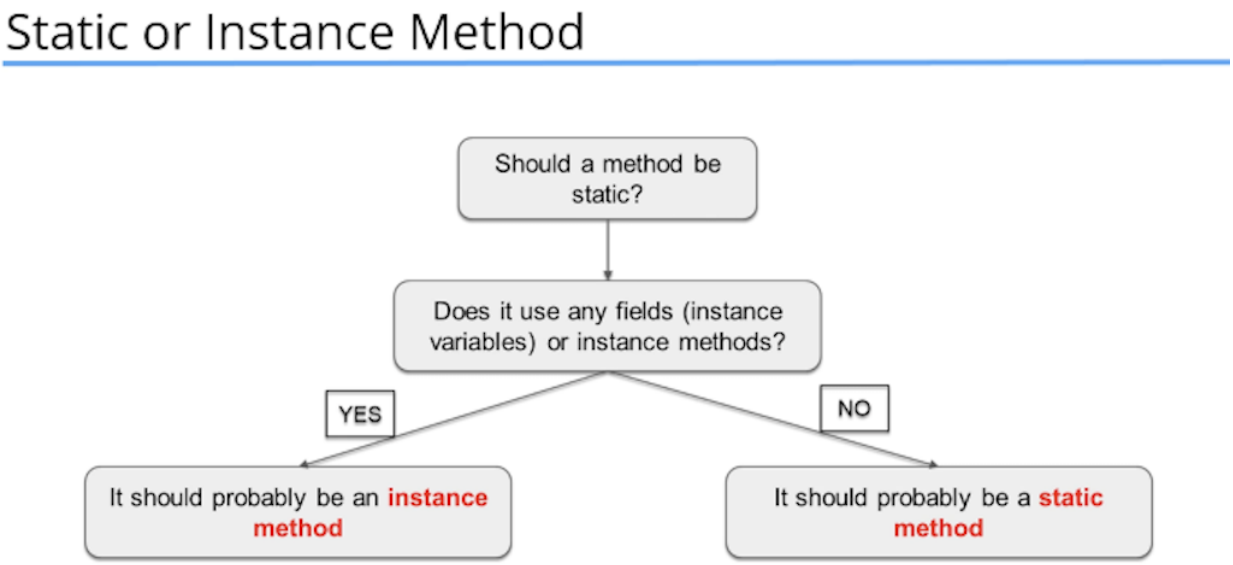
- Inside the static method we can't use *'this'* keyword
- Static methods cant access instance methods and instance variables directly
- usually used for ops that don't require any data from an instance of the class
- if a method don't use instance variables then declare it as a static method
- ex - Main is a static method and its called by JVM when it starts the java app.

```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();

- Instance methods can access instance methods and variables directly as well as they can also access static methods and variables directly.



-

POJO (Plain Old Java Object)

- Its a class that generally only has instance fields.
- Its used to house data, and pass data, between functional classes
- It usually has few, if any methods other than getters and setters
- Many database frameworks use POJO's to read data from, or to write to, databases, files or streams.
- A POJO also might be called a bean or 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.
- Its a description of an object, that can be modeled as just data.
- There are many generation tools, that will turn a data model into generated POJO's or JavaBeans.
- ex- code generation in Intellij, which allowed us to generate getters, setters, and constructors in a uniform way

- ```java
    this.id = id;
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    this.classList = classList;
  }

  @Override
  public String toString() {
    return "Student{" +
            "id='" + id + '\'' +
            ", name='" + name + '\'' +
            ", dateOfBirth='" + dateOfBirth + '\'' +
            ", classList='" + classList + '\'' +
            '}';
  }
}
```
- toString() is a special method in Java, we can implement this method in any class in Java and this method lets us print the current state of an object.

**Annotation - ex- @Override** (same thing as decorators in js and python?)

- Annotations are a type of metadata.
- Metadata is a way of formally describing additional information about our code
- Annotations are more structured, and have more meaning that comments, because they can be used by the compiler, or other types of pre-processing functions, to get information about the code.
- Metadata doesn't effect how the code runs, so this will still run, with or without the annotation.

**Overriden Method**

- An overriden method, is not the same as an overloaded method.
- An overriden method is a special method in java, that classes can implement, if they use a specified method signature.
- Every object, when passed to System.out.println(), will have the toString() method implicitly executed, if youve created such a method on your class.

**POJO vs The Record**

-
```