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.

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

```
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

```
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.

Constructors Bad Example

```
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

```
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

```
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

```
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

```
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.