this Keyword in Java

In Java, this keyword is used to refer to the current object inside a method or a constructor. It means this is nothing but the reference to the current object.

```
class Main {
   int instVar;

   Main(int instVar){
      this.instVar = instVar;
      System.out.println("this reference = " + this);
   }

   public static void main(String[] args) {
      Main obj = new Main(8);
      System.out.println("object reference = " + obj);
   }
}
```

Here, we can see that the reference of both obj and this is the same. It means this is nothing but the reference to the current object.

Use of this Keyword

There are various situations where this keyword is commonly used.

1. Using this for Ambiguity Variable Names

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name.

Example:

```
class MyClass {
   // instance variable
   int age;
   // parameter
   MyClass(int age){
      age = age;
   }
}
```

In the above program, the instance variable and the parameter have the same name: age. Here, the Java compiler is confused due to name ambiguity. In such a situation, we use this keyword.

Example: Without using this keyword

```
class Main {
   int age;
   Main(int age){
      age = age;
   }

   public static void main(String[] args) {
      Main obj = new Main(8);
      System.out.println("obj.age = " + obj.age); // Output: 0
   }
}
```

In the above example, we have passed 8 as a value to the constructor. However, we are getting 0 as an output. This is because the Java compiler gets confused because of the ambiguity in names between instance the variable and the parameter.

Example: Using this keyword

```
class Main {
   int age;
   Main(int age){
      this.age = age;
   }

   public static void main(String[] args) {
      Main obj = new Main(8);
      System.out.println("obj.age = " + obj.age);  // 8
   }
}
```

2. this with Getters and Setters

Another common use of this keyword is in setters and getters methods of a class.

Example:

```
class Main {
  String name;
  // setter method
  void setName( String name ) {
     this.name = name;
  }
  // getter method
  String getName(){
     return this.name;
  }

  public static void main( String[] args ) {
     Main obj = new Main();
     // calling the setter and the getter method
     obj.setName("Mr. Bean");
     System.out.println("obj.name: "+obj.getName());
  }
}
```

3. Using this in Constructor Overloading

While working with constructor overloading, we might have to invoke one constructor from another constructor. In such a case, we cannot call the constructor explicitly. Instead, we have to use this keyword. Here, we use a different form of this keyword. That is, this().

Example:

```
public class PK {
   public static void main(String[] args) {


      Person p1 = new Person("Mini", 18);

      p1.display();

   }
}

class Person{

   String name;
   int age;

   public Person(){
      System.out.println("The No-arg Constructor is called.");
   }

   // Update the value of the name, age, weight
   Person(String name, int age){
      this();
      this.name = name;
      this.age = age;
   }

   void display(){
      System.out.println("Person's Name : " + name);
      System.out.println("Person's Age : " + age);
   }

}
```

Note: One of the huge advantages of this() is to reduce the amount of duplicate code. Invoking one constructor from another constructor is called explicit constructor invocation.

Inheritance in Java

Inheritance is one of the core principles of Object-Oriented Programming (OOP). In Java, inheritance allows a class to inherit the fields (attributes) and methods (behaviors) of another class. This mechanism promotes code reusability and a hierarchical class structure.

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class. The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Why Do We Need Java Inheritance?

1. **Code Reusability**:
    - Code written in a superclass is common to all its subclasses. This means child classes can directly use the code from the parent class, reducing redundancy.
2. **Method Overriding**:
    - Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is crucial for achieving runtime polymorphism in Java.

3. **Abstraction**:
    - Inheritance helps in abstracting details. Abstract classes and methods allow subclasses to implement details, thereby providing specific functionalities without exposing the complete implementation to the user.

Important Terminologies in Java Inheritance

- **Class**:
    - A class is a blueprint or template for creating objects. It defines the common characteristics and behaviors that the objects will have.
- **Super Class/Parent Class**:
    - This is the class whose properties and methods are inherited by another class. It can also be referred to as a base class.
- **Sub Class/Child Class**:
    - This is the class that inherits properties and methods from another class. It can also add its own fields and methods in addition to those inherited from the superclass.
- **Reusability**:
    - Inheritance supports reusability, meaning that when a new class is created and there is already a class with some of the desired code, the new class can be derived from the existing class. This allows the new class to reuse the fields and methods of the existing class.

How to Use Inheritance in Java?

Inheritance in Java is implemented using the **extends** keyword. The extends keyword indicates that a class is derived from another class, thus inheriting its properties and methods.

Syntax:

```
class DerivedClass extends BaseClass {
  // methods and fields
}
```

Example:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // inherited method from Animal class
        dog.bark(); // method in Dog class
    }
}
```