Keyword:this

- In java, this is a **reference variable** that refers to the current object.

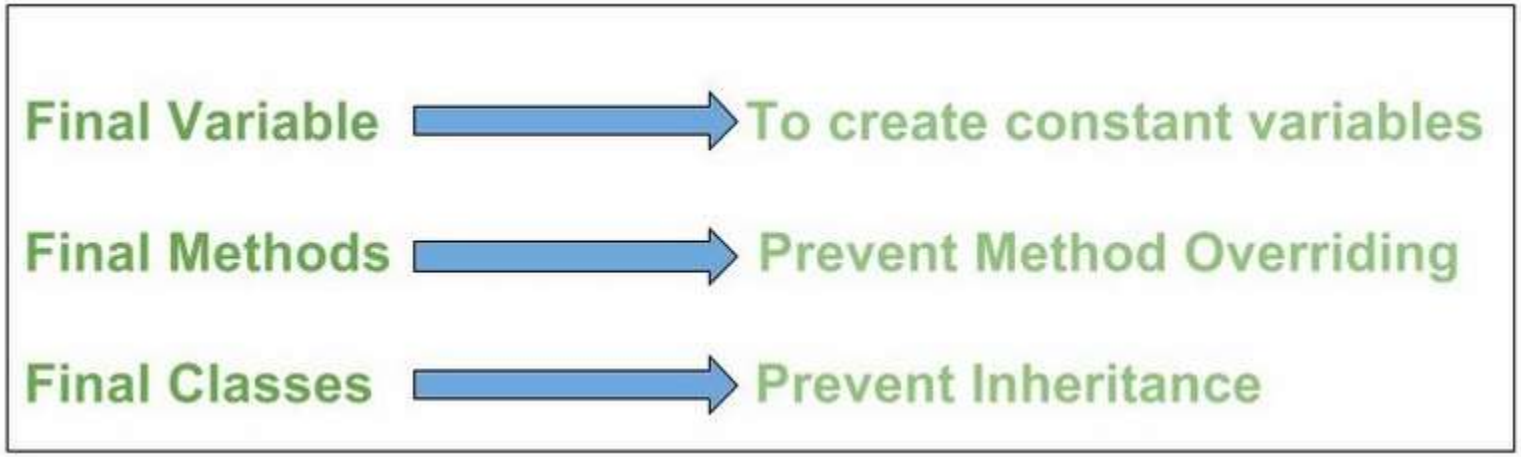
```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

- **we are using this keyword to distinguish local variable and instance variable.**

Keyword: final

- The final keyword in Java can be used with variables, methods, and classes.
- **Final Variables**: When we mark a variable as final, we can’t reassign its value.
- **Final Class**: When a class is final, we can’t extend it.
- **Final Method**: We can’t override a final method.


Final Variable
To create constant variables
Final Methods
Prevent Method Overriding
Final Classes
Prevent Inheritance

- **Example1:**

```
class Bike{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;  //Error
 }
 public static void main(String args[]){
 Bike obj=new  Bike();
 obj.run();
 }
}
```

- **Example2:**

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100k
    mph");}  //Error

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

- **Example3:**

```
final class Bike{}

class Honda1 extends Bike //Error
{
  void run(){System.out.println("running safely with 100k
    mph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Super Keyword in Java

- The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword “super” came into the picture with the concept of Inheritance.

- **Example1:**

```
/* Base class vehicle */
class Vehicle
{
   int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
   int maxSpeed = 180;

   void display()
   {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}

/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

- Output:

Maximum Speed: 120

Method Overriding

- In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.

- Method overriding is one of the way by which java achieve Run Time Polymorphism.
- The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

- **Example1:**

```
class Parent {
   void show()
   {
      System.out.println("Parent's show()");
   }
}
class Child extends Parent {
   // This method overrides show() of Parent
   @Override
   void show()
   {
      System.out.println("Child's show()");
   }
}
class Main {
   public static void main(String[] args)
   {
      // If a Parent type reference refers
      // to a Parent object, then Parent's
      // show is called
      Parent obj1 = new Parent();
      obj1.show();

      // If a Parent type reference refers
      // to a Child object Child's show()
      // is called. This is called RUN TIME
      // POLYMORPHISM.
      Parent obj2 = new Child();
      obj2.show();
   }
}
```

- Output:

Parent's show()

Child's show()

Rules for method overriding:

- **1. Overriding and Access-Modifiers :** The access modifier for an overriding method can allow more, but not less, access than the overridden method.
- For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

- 2. **Final methods can not be overridden :** If we don’t want a method to be overridden, we declare it as final.
- **3. Static methods can not be overridden :**
- When you defines a static method with same signature as a static method in base class, it is known as method hiding.

- **Example:**

```
class Parent {
    // Static method in base class
    // which will be hidden in subclass
    static void m1()
    {
        System.out.println("From parent "
                           + "static m1()");
    }

    // Non-static method which will
    // be overridden in derived class
    void m2()
    {
        System.out.println("From parent "
                           + "non-static(instance) m2()");
    }
}

class Child extends Parent {
    // This method hides m1() in Parent
    static void m1()
    {
        System.out.println("From child static m1()");
    }

    // This method overrides m2() in Parent
    @Override
    public void m2()
    {
        System.out.println("From child "
                           + "non-static(instance) m2()");
    }
}
```

```
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this
        // should call to class Child static
        // overridden method. Since static
        // method can not be overridden, it
        // calls Parent's m1()
        obj1.m1();

        // Here overriding works
        // and Child's m2() is called
        obj1.m2();
    }
}
```

- **Output:**

```
From parent static m1()
From child non-static(instance) m2()
```

- 4. **Private methods can not be overridden :** Private methods cannot be overridden as they are bonded during compile time.
- 5. **The overriding method must have same return type (or subtype) :**
- From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.

- **Example:**

```
class A {}
class B extends A {}

class Base
{
   A fun()
   {
      System.out.println("Base fun()");
      return new A();
   }
}

class Derived extends Base
{
   B fun()
   {
      System.out.println("Derived fun()");
      return new B();
   }
}

public class Main
{
   public static void main(String args[])
   {
     Base base = new Base();
     base.fun();

     Derived derived = new Derived();
     derived.fun();
   }
}
```

- Output:

Base fun()

Derived fun()

- 6. **Invoking overridden method from sub-class :** We can call parent class method in overriding method using super keyword.

- Example:

```
class Parent {
   void show()
   {
      System.out.println("Parent's show()");
   }
}

// Inherited class
class Child extends Parent {
   // This method overrides show() of Parent
   @Override
   void show()
   {
      super.show();
      System.out.println("Child's show()");
   }
}

// Driver class
class Main {
   public static void main(String[] args)
   {
      Parent obj = new Child();
      obj.show();
   }
}
```

- **Output:**

Parent's show()

Child's show()