- ## Concept of Polymorphism:

- **Java Polymorphism**

1) Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
2) Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
3) For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

Example

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

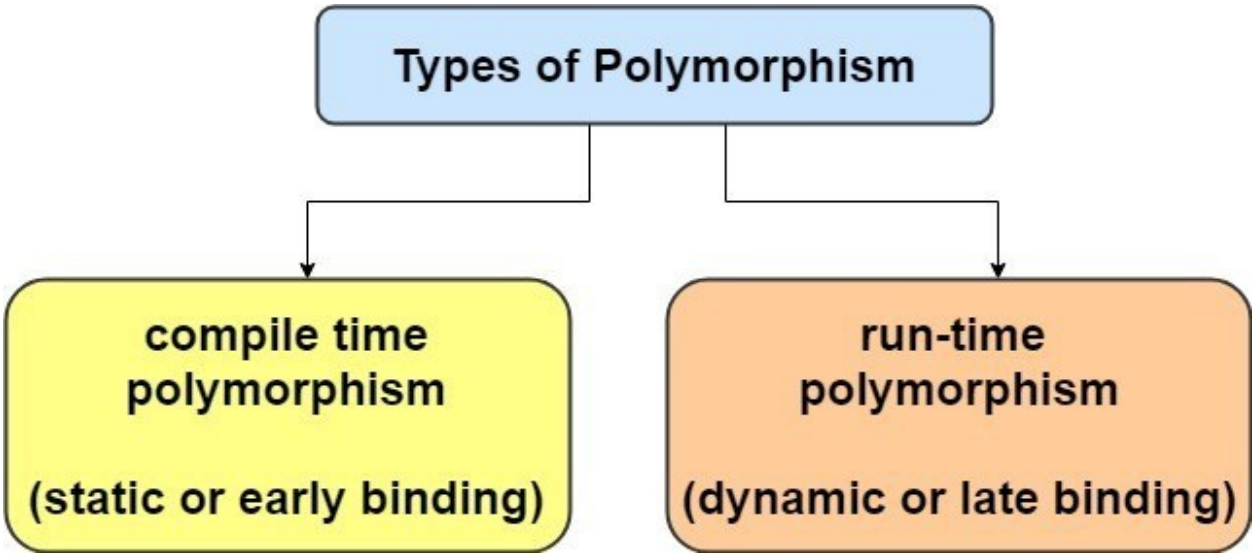
Output:

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

- ## Use Overridden methods to effect Polymorphism: Polymorphism in Java – Method Overloading | Method Overriding

1) Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So Polymorphism means the ability to take many forms. Is a concept by which we can perform a ***single task*** *in* ***different ways***. It is one of the most striking features of Object Oriented Programming in Java.
2) In terms of Java Programming Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

3) There are two types of polymorphism in java: **compile time polymorphism (static or early binding)** and **runtime polymorphism (dynamic or late binding)**.



1. Compile Time Polymorphism – Method Overloading

2. Run Time Polymorphism – Method Overriding

1) Compile Time Polymorphism

- Method Overloading in Java

1. Method overloading in Java is a feature which makes it possible to use the **same method** name to perform **different tasks**. In this tutorial post we will understand the concept of **Java Method Overloading**. In the previous post, we understood what are ***methods in java*** so if you have missed that post you can check it out.
2. If a class has **multiple** methods having **same name** but **different** in **parameters**, it is known as **Method Overloading**. Overloading allows different methods to have same name, but **different signatures** where signature can differ by **number of input parameters** or **type of input parameters** or **order of input parameters**. Overloading is related to **compile time (or static) polymorphism.**

Let's, take an example situation to understand –

Suppose you have a program to perform addition of numbers and you create a method to add 2 numbers as – add(int x, int y).

Now you have a new requirement to add 3 numbers so you again create a new method as – add3(int x, int y, int z). But now again you have one more new requirement to add 4 numbers. You see the hassle here right? Every time you get this requirement you have to find a new method name. But the underlying logical operation is the same right? Cause in the end we are anyways going to perform addition. So why use different method names. So that is where method overloading comes to the rescue.

Without Method Overloading

```
int add2(int x, int y)
{
   return(x+y);
}
int add3(int x, int y,int z)
{
   return(x+y+z);
}
int add4(int w, int x,int y, int z)
{
   return(w+x+y+z);
}
```

With Method Overloading

```
int add(int x, int y)
{
   return(x+y);
}
int add(int x, int y,int z)
{
   return(x+y+z);
}
int add(int w, int x,int y, int z)
{
   return(w+x+y+z);
}
```

Method Overloading in Java

Advantage of method overloading

Method overloading increases the readability of the program.

Different ways to overload the method

There are 3 ways to overload the method in java

- By changing **number of arguments**
- By changing the **data type**
- By changing the **order** of arguments (if different data types are involved)

Example of Method Overloading in Java

Java Method Overloading example program code

```
1  public class MethodOverloading {
2
3      //1) 2 arguments int & double
4      double add(int x, double y)
5      {
6          return(x+y);
7      }
8      //2) 2 arguments but different order - double & int
9      double add(double x, int y)
10     {
11         return(x+y);
12     }
13     //3) 3 arguments
14     double add(double x, int y, float z)
15     {
16         return(x+y+z);
17     }
```

```
18
19    public static void main(String[] args) {
20        // TODO code application logic here
21        int a=5;
22        double b = 7.5;
23        float c = 4.5f;
24
25        double result;
26
27        MethodOverloading obj = new MethodOverloading();
28        result = obj.add(b, a); // 1st add function
29        System.out.println("Addtion is: "+result);
30        result = obj.add(a, b); // 2nd add function
31        System.out.println("Addtion is: "+result);
32        result = obj.add(b,a,c); // 3rd add function
33        System.out.println("Addtion is: "+result);
34
35    }
36
37}
```

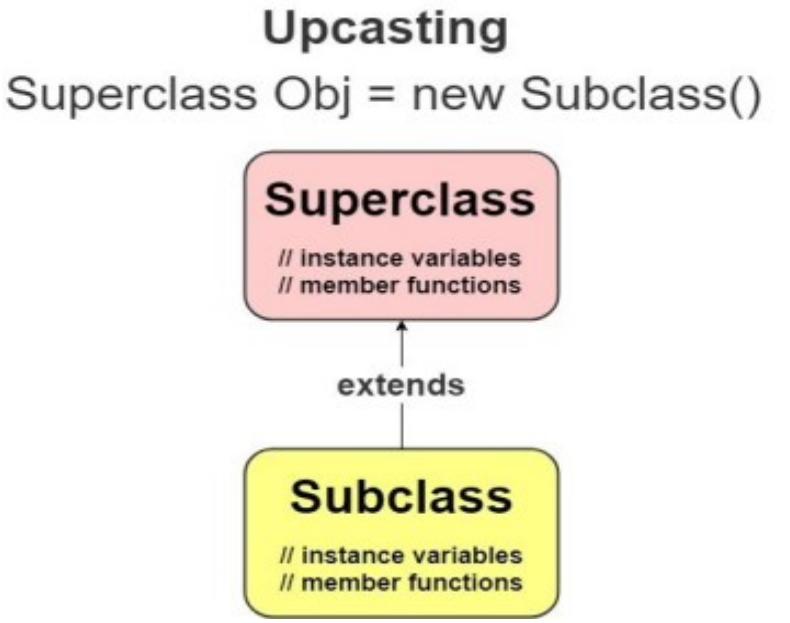
Output

```
Addtion is: 12.5
Addtion is: 12.5
Addtion is: 17.0
```

2) Run Time Polymorphism

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a **superclass**. Thus this happens only when **Inheritance** is implemented. The method to be called is based on the object being referred to by the reference variable.

- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.
- A superclass reference variable can refer to a subclass object. This is also known as **upcasting**. Java uses this fact to resolve calls to overridden methods at run time.



Let's see a program example to understand the working of run-time polymorphism -

```
1 // A Simple Java program to demonstrate
2 // method overriding in java
3 // Base Class
4 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() { System.out.println("Child's show()"); }
}

// Driver class
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();
```

```
24
25
26
27
28     }
29}
30
31
32
33
34
```

Output

Parent's show()
Child's show()

In the above example, you can see that depending on the type of Object (Parent or Child) the respective method show() is being called. Also using the parent object, we have created a reference to the child class (Parent obj2 = new Child()). Thus now the parent object is referring to the child class and hence we are able to access the child class's show() method.

Some Rules to follow in Method Overriding –

- **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.

- Final methods can not be overridden.
- Static methods can not be overridden(Method Overriding vs. Method Hiding).
- Private methods can not be overridden.
- The overriding method must have same return type (or subtype)
- **Invoking overridden method from sub-class :** We can call parent class method in overriding method using **super** keyword.
- **Overriding and constructor :** We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
- **Overriding and abstract method :** Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise compile-time error will be thrown.

- **Distinguish between abstract and concrete classes:**

Abstract Class: An abstract class is a type of class in Java that is declared by the abstract keyword. An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword.
An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. It may or may not contain abstract method. An abstract method is declared by abstract keyword, such methods cannot have a body. If a class contains abstract method, then it also needs to be abstract.
Concrete Class: A concrete class in Java is a type of subclass, which implements all the abstract method of its super abstract class which it extends to. It also has implementations of all methods of interfaces it implements.

Abstract Class vs. Concrete Class

1. **Modifier:** An abstract class is declared using abstract modifier. Concrete class should not be declared using abstract keyword, on doing so, it will also become abstract class.
2. **Instantiation:** An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword. An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. A concrete class can be instantiated directly, using a new keyword.

Example: Invalid direct instantiation of an abstract class.

```
abstract class DemoAbstractClass {
   abstract void display();
}

public class JavaApplication {
   public static void main(String[] args)
   {
      DemoAbstractClass AC = new DemoAbstractClass();
      System.out.println("Hello");
   }
}
```

Compile Error:

```
prog.java:9: error: DemoAbstractClass is abstract; cannot be instantiated
      DemoAbstractClass AC = new DemoAbstractClass();
                             ^
```

Example: Valid instantiation by defining all abstract method of an abstract class.

```
abstract class DemoAbstractClass {
   abstract void display();
}

public class JavaApplication {
   public static void main(String[] args)
   {
      DemoAbstractClass AC = new DemoAbstractClass() {
         void display()
         {
            System.out.println("Hi.");
         }
      };
      AC.display();
```

```
        System.out.println("How are you?");
    }
}
```

Output:

```
Hi.

How are you?
```

Example: Direct instantiation of concrete using new keyword.

```
abstract class DemoAbstractClass {
    abstract void display();
}

class ConcreteClass extends DemoAbstractClass {
    void display()
    {
        System.out.println("Hi.");
    }
}

public class JavaApplication {
    public static void main(String[] args)
    {
        ConcreteClass C = new ConcreteClass();
        C.display();
        System.out.println("How are you?");
    }
}
```

Output:

```
Hi.

How are you?
```

3. **Abstract methods:** An abstract class may or may not, have an abstract method. A concrete class cannot have an abstract method, because class containing an abstract method must also be abstract.
4. **Final:** An abstract class cannot be **final**, because all its abstract methods must defined in the subclass. A concrete class can be declared as **final**.
5. **Interface:** Interface implementation is not possible with abstract class, however, it is possible with concrete class.

ABSTRACT CLASS	CONCRETE CLASS
An abstract class is declared using	A concrete class is note declared using

ABSTRACT CLASS	CONCRETE CLASS
abstract modifier.	abstract modifier.
An abstract class cannot be directly instantiated using the new keyword.	A concrete class can be directly instantiated using the new keyword.
An abstract class may or may not contain abstract methods.	A concrete class cannot contain an abstract method.
An abstract class cannot be declared as final.	A concrete class can be declared as final.
Interface implementation is not possible	Interface implementation is possible.

Some important points:

- A concrete class is a subclass of an abstract class, which implements all its abstract method.
- Abstract methods cannot have body.
- Abstract class can have static fields and static method, like other classes.
- An abstract class cannot be declared as final.
- Only abstract class can have abstract methods.
- A private, final, static method cannot be abstract, as it cannot be overridden in a subclass.
- Abstract class cannot have abstract constructors.
- Abstract class cannot have abstract static methods.
- If a class extends an abstract class, then it should define all the abstract methods (override) of the base abstract class. If not, the subclass (the class extending abstract class) must also be defined as abstract class.

- **To declare abstract methods to create abstract classes:**

Abstract Class in Java with example

A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). A normal class (non-abstract class) cannot have abstract methods. In this guide we will learn what is an abstract class, why we use it and what are the rules that we must remember while working with it in Java.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it. Why? We will discuss that later in this guide:

Why we need an abstract class?

Let's say we have a class Animal that has a method sound() and the subclasses(inheritance) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Abstract class Example

```
//abstract parent class
abstract class Animal{
  //abstract method
  public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

  public void sound(){
        System.out.println("Woof");
  }
  public static void main(String args[]){
```

```
        Animal obj = new Dog();
        obj.sound();
    }
}
```

Output:

```
Woof
```

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
  //This is abstract method
  abstract void myMethod();

  //This is concrete method with body
  void anotherMethod(){
    //Does something
  }
}
```

Rules

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Do you know? Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user.Interfaces on the other hand are used for 100% abstraction

Why can't we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then what would happen? There would be no actual implementation of the method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

```
abstract class AbstractDemo{
  public void myMethod(){
    System.out.println("Hello");
  }
  abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

  public void anotherMethod() {
      System.out.print("Abstract method");
  }
  public static void main(String args[])
  {
    //error: You can't create object of it
    AbstractDemo obj = new AbstractDemo();
    obj.anotherMethod();
  }
}
```

Output:

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo
```

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

Abstract class vs. Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, Animal is an abstract class and Cat, Dog & Lion are concrete classes.

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. You can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.
   For now let's just see some basics and example of abstract method.
   1) Abstract method has no body.
   2) Always end the declaration with a **semicolon**(;).
   3) It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
   4) A class has to be declared abstract to have abstract methods.

   Note: The class which is extending abstract class must override all the abstract methods.

Example of Abstract class and method

```
abstract class MyClass{
  public void disp(){
    System.out.println("Concrete method of parent class");
  }
  abstract public void disp2();
}

class Demo extends MyClass{
  /* Must Override this method while extending
   * MyClas
   */
  public void disp2()
  {
     System.out.println("overriding abstract method");
  }
  public static void main(String args[]){
     Demo obj = new Demo();
     obj.disp2();
  }
}
```

Output:

```
overriding abstract method
```

Abstract method in Java with examples

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well.

This is how an abstract method looks in java:

```
public abstract int myMethod(int n1, int n2);
```
As you see this has no body.

Rules of Abstract Method

1. Abstract methods don't have body, they just have method signature as shown above.
2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Example 1: abstract method in an abstract class

```
//abstract class
abstract class Sum{
   /* These two are abstract methods, the child class
    * must implement these methods
    */
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   //Regular method
   public void disp(){
            System.out.println("Method of class Sum");
   }
}
//Regular class extends abstract class
class Demo extends Sum{
```

```
  /* If I don't provide the implementation of these two methods, the
   * program will throw compilation error.
   */
  public int sumOfTwo(int num1, int num2){
          return num1+num2;
  }
  public int sumOfThree(int num1, int num2, int num3){
          return num1+num2+num3;
  }
  public static void main(String args[]){
          Sum obj = new Demo();
          System.out.println(obj.sumOfTwo(3, 7));
          System.out.println(obj.sumOfThree(4, 3, 19));
          obj.disp();
  }
}
```

Output:

```
10
26
Method of class Sum
```

Example 2: abstract method in interface

All the methods of an interface are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

```
//Interface
interface Multiply{
  //abstract methods
  public abstract int multiplyTwo(int n1, int n2);

  /* We need not to mention public and abstract in interface
   * as all the methods in interface are
   * public and abstract by default so the compiler will
   * treat this as
   * public abstract multiplyThree(int n1, int n2, int n3);
   */
  int multiplyThree(int n1, int n2, int n3);

  /* Regular (or concrete) methods are not allowed in an interface
   * so if I uncomment this method, you will get compilation error
   * public void disp(){
   *    System.out.println("I will give error if u uncomment me");
```

```
   * }
   */
}

class Demo implements Multiply{
  public int multiplyTwo(int num1, int num2){
    return num1*num2;
  }
  public int multiplyThree(int num1, int num2, int num3){
    return num1*num2*num3;
  }
  public static void main(String args[]){
    Multiply obj = new Demo();
    System.out.println(obj.multiplyTwo(3, 7));
    System.out.println(obj.multiplyThree(1, 9, 0));
  }
}
```

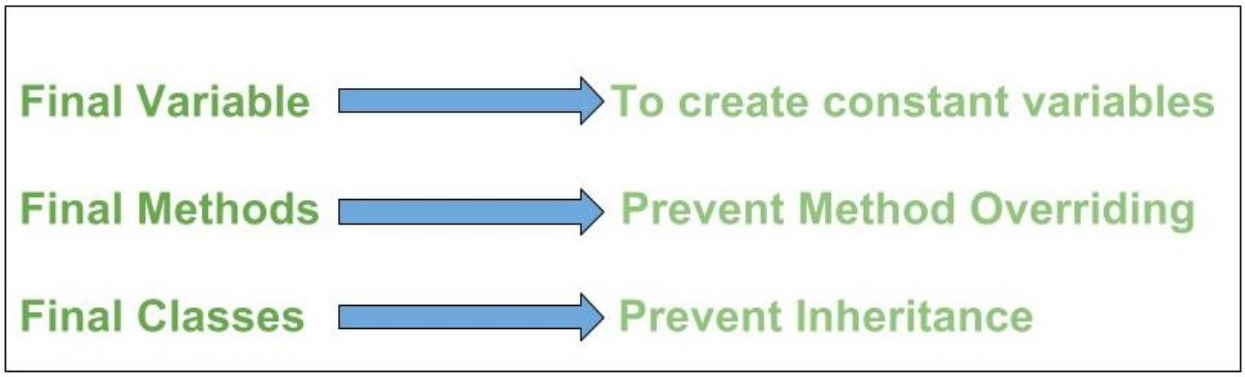
Output:

```
21
0
```

- ## Final Methods and Final Classes:

final keyword in java

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**. Following are different contexts where final is used.



Final variables

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is

a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a  blank final static  variable
static final double PI;
```

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared. This approach is the most common. A final variable is called **blank final variable**, if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside static block.

Let us see above different ways of initializing a final variable through an example.

```
//Java program to demonstrate different
// ways of initializing a final variable

class Gfg
{
    // a final variable
    // direct initialize
    final int THRESHOLD = 5;
```

```
// a blank final variable
final int CAPACITY;

// another blank final variable
final int  MINIMUM;

// a final static variable PI
// direct initialize
static final double PI = 3.141592653589793;

// a  blank final static  variable
static final double EULERCONSTANT;

// instance initializer block for
// initializing CAPACITY
{
   CAPACITY = 25;
}

// static initializer block for
// initializing EULERCONSTANT
static{
   EULERCONSTANT = 2.3;
}

// constructor for initializing MINIMUM
// Note that if there are more than one
```

```
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;
    }


}
```

When to use a final variable :
The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Reference final variable :
When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like

```
final StringBuffer sb;
```

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity*. To understand what is mean by internal state of the object, see below example :

```
// Java program to demonstrate
// reference final variable


class Gfg
{
    public static void main(String[] args)
    {
        // a final reference variable sb
        final StringBuilder sb = new StringBuilder("IOPE");


        System.out.println(sb);
```

```
        // changing internal state of object
        // reference by final reference variable sb
        sb.append("ForIOPE");

        System.out.println(sb);
    }
}
```

Output:

```
IOPE
IOPEForIOPE
```

The *non-transitivity* property also applies to arrays, because arrays are objects in java. Arrays with final keyword are also called final arrays.

Note :

1. As discussed above, a final variable cannot be reassign, doing it will throw compile -time error.

```
        // Java program to demonstrate re-assigning
        // final variable will throw compile-time error

        class Gfg
        {
            static final int CAPACITY = 4;

            public static void main(String args[])
            {
                // re-assigning final variable
                // will throw compile-time error
                CAPACITY = 5;
            }
        }
```

Output

```
Compiler Error: cannot assign a value to final variable CAPACITY
```

2. When a final variable is created inside a method/constructor/block, it is called local final
variable, and it must initialize once where it is created. See below program for local final variable

```
// Java program to demonstrate
// local final variable


// The following program compiles and runs fine


class Gfg
{
   public static void main(String args[])
   {
      // local final variable
      final int i;
      i = 20;
      System.out.println(i);
   }
}
```

Output:

```
20
```

3. Note the difference between C++ *const* variables and Java *final* variables. const variables in C++ must be assigned a value when declared. For final variables in Java, it is not necessary as we see in above examples. A final variable can be assigned value later, but only once.
4. *final* with for each loop : final with for-each statement is a legal statement.

```
// Java program to demonstrate final
```

```
// with for-each statement


class Gfg
{
   public static void main(String[] args)
   {
      int arr[] = {1, 2, 3};


      // final with for-each statement
      // legal statement
      for (final int i : arr)
         System.out.print(i + " ");
   }
}
```

Output:

```
1 2 3
```

Explanation : Since the i variable goes out of scope with each iteration of the loop, it is actually re-declaration each iteration, allowing the same token (i.e. i) to be used to represent multiple variables.

Final classes

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class :

1. One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float etc. are final classes. We can not extend them.

```
final class A
{
    // methods and fields
}
// The following class is illegal.
class B extends A
```

```
8.  {
9.      // COMPILE-ERROR! Can't subclass A
10. }
```

11. The other use of final with classes is to create an immutable class like the predefined String class. You can not make a class immutable without making it final.

Final methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. The Object class does this—a number of its methods are final. We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

```
class A
{
   final void m1()
   {
      System.out.println("This is a final method.");
   }
}

class B extends A
{
   void m1()
   {
      // COMPILE-ERROR! Can't override.
      System.out.println("Illegal!");
   }
}
```

- ## Creating and Using Interfaces:

What is an interface in Java?

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. We will cover this in detail, later in this guide:

What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Syntax:
Interfaces are declared by specifying a keyword “interface”. E.g.:

```
interface MyInterface
{
  /* All the methods are public abstract by default
   * As you see they have no body
   */
  public void method1();
  public void method2();
}
```

Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

Do you know? class implements interface but an interface extends another interface.

```
interface MyInterface
{
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2();
   */
  public void method1();
  public void method2();
}
class Demo implements MyInterface
{
  /* This class must have to implement both the abstract methods
   * else you will get compilation error
   */
  public void method1()
  {
        System.out.println("implementation of method1");
  }
  public void method2()
  {
        System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
        MyInterface obj = new Demo();
        obj.method1();
  }
}
```

Output:

```
implementation of method1
```

Interface and Inheritance

As discussed above, an interface can not implement another interface. It has to extend the other interface. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

```
interface Inf1{
  public void method1();
}
interface Inf2 extends Inf1 {
  public void method2();
}
```

```
public class Demo implements Inf2{
  /* Even though this class is only implementing the
   * interface Inf2, it has to implement all the methods
   * of Inf1 as well because the interface Inf2 extends Inf1
   */
   public void method1(){
            System.out.println("method1");
   }
   public void method2(){
            System.out.println("method2");
   }
   public static void main(String args[]){
            Inf2 obj = new Demo();
            obj.method2();
   }
}
```

In this program, the class Demo only implements interface Inf2, however it has to provide the implementation of all the methods of interface Inf1 as well, because interface Inf2 extends Inf1.

Tag or Marker interface in Java

An empty interface is known as tag or marker interface. For example Serializable, EventListener, Remote (java.rmi.Remote) are tag interfaces. These interfaces do not have any field and methods in it.

Nested interfaces

An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface. For example Entry interface in collections framework is declared inside Map interface, that's why we don' use it directly, rather we use it like this: Map.Entry.

Key points: Here are the key points to remember about interfaces:
1) We can't instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.

3) implements keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be

mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

```
interface Try
{
  int a=10;
  public int a=10;
  public static final int a=10;
  final int a=10;
  static int a=0;
}
```

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try
{
    int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

```
class Sample implements Try
{
 public static void main(String args[])
 {
   x=20; //compile time error
 }
```

```
}
```

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
  public void aaa();
}
interface B
{
  public void aaa();
}
class Central implements A,B
{
  public void aaa()
  {
     //Any Code here
  }
  public static void main(String args[])
  {
     //Statements
   }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
  public void aaa();
}
interface B
{
  public int aaa();
}

class Central implements A,B
{

  public void aaa() // error
  {
```

```
  }
  public int aaa() // error
  {
  }
  public static void main(String args[])
  {

  }
}
```

15) Variable names conflicts can be resolved by interface name.

```
interface A
{
   int x=10;
}
interface B
{
   int x=100;
}
class Hello implements A,B
{
   public static void Main(String args[])
   {
      /* reference to x is ambiguous both variables are x
       * so we are using interface name to resolve the
       * variable
       */
      System.out.println(x);
      System.out.println(A.x);
      System.out.println(B.x);
   }
}
```

Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface

Programs :

```
// A simple interface
interface Player
```

```
    {
        final int id = 20;
        int move();
    }
```

To implement an interface we use keyword: implement

```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 20;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("INFORMATION TECHNOLOGY");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}
```

Output:

```
INFORMATION TECHNOLOGY
20
```

A real world example:

Let's consider the example of vehicles like bicycle, car, and bike……., they have common

functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc. implement all these functionalities in their own class in their own way.

```
import java.io.*;

interface Vehicle {

   // all are the abstract methods.
   void changeGear(int a);
   void speedUp(int a);
   void applyBrakes(int a);
}

class Bicycle implements Vehicle{

   int speed;
   int gear;

    // to change gear
   @Override
   public void changeGear(int newGear){

      gear = newGear;
   }

   // to increase speed
   @Override
   public void speedUp(int increment){

      speed = speed + increment;
   }

   // to decrease speed
   @Override
   public void applyBrakes(int decrement){

      speed = speed - decrement;
   }

   public void printStates() {
       System.out.println("speed: " + speed
           + " gear: " + gear);
   }
}
```

```
class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
            + " gear: " + gear);
    }

}
class GFG {

    public static void main (String[] args) {

        // creating an instance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();
```

```
        // creating instance of bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

Output:

```
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1
```

New features added in interfaces in JDK 8

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.
   Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
// An example to show that interfaces can
// have methods from JDK 1.8 onwards
interface in1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
    }
}

// A class that implements interface.
class testClass implements in1
{
    // Driver Code
```

```
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
    }
}
```

Output :

```
Hello
```

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```
// An example to show that interfaces can
// have methods from JDK 1.8 onwards
interface in1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}

// A class that implements interface.
class testClass implements in1
{
    // Driver Code
    public static void main (String[] args)
    {
        in1.display();
    }
}
```

Output :

```
hello
```