

java is a machine, embedded devices smart phones, cloudm etc
it shares the features of the language and libraries
principles - java can run anywhere
- it has a collection of predefined classes. you can use them either
- by extending them.

JAVA EE(Advance java)

JAVA ME(for mobies)

features of java

- * simple
- *object oriented
- *Distributed
- *interpreted
- *robust
- *secure
- *Portable
- *multi-threded
- *Garbage-collector

Compile

sagir.java conveted to sagir.class or to byte code

run

sagir.class is coverted to machin understandable form..through JVM

JDK : Java developement kit contains tools needed to develop the java program. Jthese tools can be compiler (javac.exe) or (java.exe)

JRE : java run time environment contains **java virtual machin (JVM)**and java package classes(java library).

Java virtual machine interprets the byte code into machin code depending upon the underlying operating system and hardwhrere combination..it also provides platform independent way of executing code.

Java is case senisitive language li C and C++

java is nearly 100% object oriented language

DATA TYPES AND KEYWORDS IN JAVAo

keywords :

abstract default goto package this asset do if private throw boolean double throws enum else short extends false int short try true void switch finally fianl float new switch continu const super long volatile catch byte break interface private protected public **etc....**

DATATYPE : A type identities a set of values . A data type in programming language is set of data with predefined values.

Data types are divided into two groups:

- Primitive data types** – includes **byte, short, int, long, float, double, boolean** and **char**
- Non-primitive data types - such as String,Arrays andClasses

java is strongly typed language how ??

Java is a strongly typed programming language because every variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.

A **strongly-typed** programming **language** is one in which each type of data (such as integer, character, etc.) is predefined as part of the programming **language** and all constants or variables defined for a given program must be described with one of the data types.

In Java, when a variable is declared, it must be informed to the compiler what data type the variable stores like integer, float, double or string etc.

For Example, consider the simple code snippet :

```
int age = 20;  
String name = "Tridib";  
boolean ans = true;
```

In this snippet, each variable is declared with the data type. Similarly, an array object must be instantiated with the size of the array.

Secondly, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Hence java is a strongly-typed programming language !

Constant in Java

A constant variable is the one whose value is fixed and only one copy of it exists in the program. Once you declare a constant variable and assign value to it, you cannot change its value again throughout the program.

Unlike in C language constants are not supported in Java(directly). But, you can still create a constant by declaring a variable static and final.

- Once you declare a variable static they will be loaded in to the memory at the compile time i.e. only one copy of them is available.
- once you declare a variable final you cannot modify its value again.

Example

Live Demo

```
class Data {  
  
    static final int integerConstant = 20;  
  
}
```

```
public class ConstantsExample {  
  
    public static void main(String args[]) {  
  
        System.out.println("value of integerConstant: "+Data.integerConstant);  
  
    }  
  
}
```

Output

```
value of integerConstant: 20
```

Final variable in Java

Once you declare a variable final you cannot change its value. If you try to do so a compile time error will be generated.

Example

[Live Demo](#)

```
public class FinalExample {  
  
    public static void main(String args[]) {  
  
        final int num = 200;  
  
        num = 2544;  
  
    }  
  
}
```

Output

```
FinalExample.java:4: error: cannot assign a value to final variable num  
    num = 2544;  
    ^  
1 error
```

The main difference between final variable and a constant (static and final) is that if you create a final variable without static keyword, though its value is un-modifiable, a separate copy of the variable is created each time you create a new object. Where a constant is un-modifiable and have only one copy through out the program. For example, consider the following Java program,

Example

Live Demo

```
class Data {  
    final int integerConstant = 20;  
}  
  
public class ConstantExample {  
    public static void main(String args[]) {  
        Data obj1 = new Data();  
        System.out.println("value of integerConstant: "+obj1.integerConstant);  
        Data obj2 = new Data();  
        System.out.println("value of integerConstant: "+obj2.integerConstant);  
    }  
}
```

Output

```
value of integerConstant: 20  
value of integerConstant: 20
```

Here we have created a final variable and trying to print its value using two objects, though value of the variable is same at both instances, since we have used a different object for each they are the copies of the actual variable.

According to the definition of the constant you need to have a single copy of the variable throughout the program (class).

Therefore, to create constant as per definition, you need to declare it both static and final

Type conversion in java

1) widening conversion

```
int y= 3
```

```
float x= y //widening conversion no error
```

2) Narrowing conversion

```
float x =3.4f
```

```
int y = x; //error
```

```
int y = int(x); //no error due to casting
```

```
float k = 3.5 //error
```

```
float k = 3.5f //no error
```

permitted conversions

byte to short ,int, long ,float, double

short to int, long ,float, double

char to int, long ,float, double

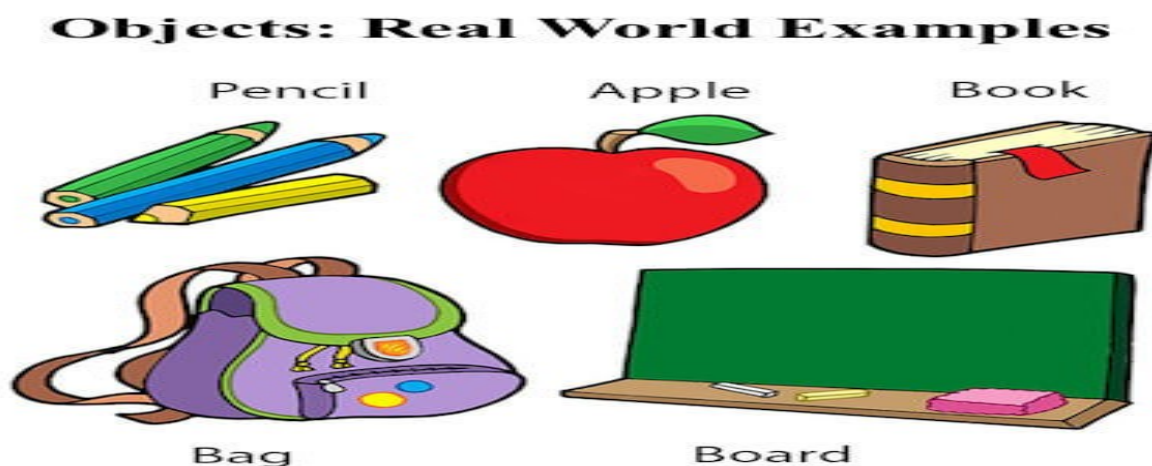
int, to long ,float, double

long to float, double

float to double

CLASSES AND OBJECTS IN JAVA

What is an object in Java



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- State: represents the data (value) of an object.

- Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.

- Identity: An object identity is typically implemented **via a unique ID. The value of the ID** is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

Characteristics of Object



For Example, Pen is an object. Its name is Reynolds; **color is white**, known as its state. **It is used to write**, so **writing is its behavior**.

An object is an instance of a class.

A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

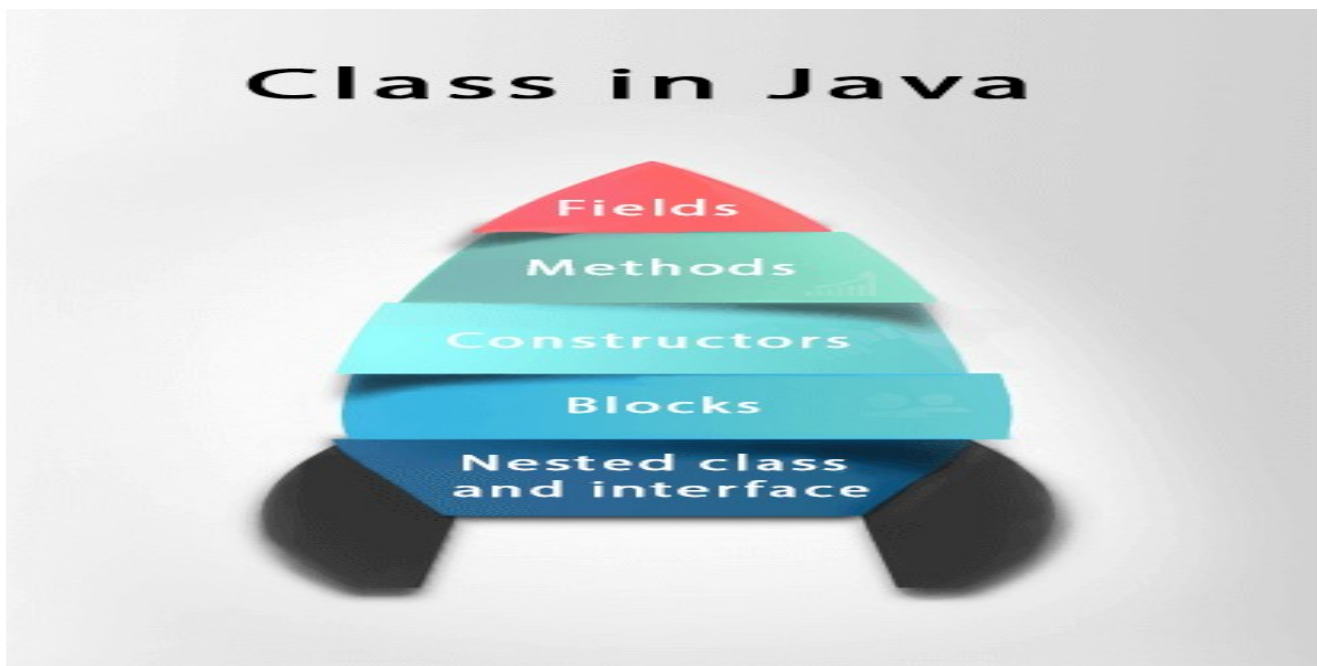
- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface



Syntax to declare a class:

1. **class** <class_name>{
 2. field;
 3. method;
 4. }
-

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
 - Code Optimization
-

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

1. `//Java Program to illustrate how to define a class and fields`
2. `//Defining a Student class.`
3. `class Student{`
4. `//defining fields`
5. `int id;//field or data member or instance variable`
6. `String name;`
7. `//creating main method inside the Student class`
8. `public static void main(String args[]){`
9. `//Creating an object or instance`
10. `Student s1=new Student();//creating an object of Student`
11. `//Printing values of the object`
12. `System.out.println(s1.id);//accessing member through reference variable`
13. `System.out.println(s1.name);`
14. `}`
15. `}`

Test it Now

Output:

0

null

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

1. `//Java Program to demonstrate having the main method in`
2. `//another class`
3. `//Creating Student class.`
4. `class Student{`
5. `int id;`
6. `String name;`
7. `}`
8. `//Creating another class TestStudent1 which contains the main method`
9. `class TestStudent1{`
10. `public static void main(String args[]){`
11. `Student s1=new Student();`
12. `System.out.println(s1.id);`
13. `System.out.println(s1.name);`
14. `}`
15. `}`

Output:

0

null

3 Ways to initialize objec

There are 3 ways to initialize object in Java.

- 1.By reference variable
- 2.By method
- 3.By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1. class Student{
2. int id;
3. String name;
4. }
5. class TestStudent2{
6. public static void main(String args[]){
7. Student s1=new Student();
8. s1.id=101;
9. s1.name="Sonoo";
10. System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12.}
```

Test it Now

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

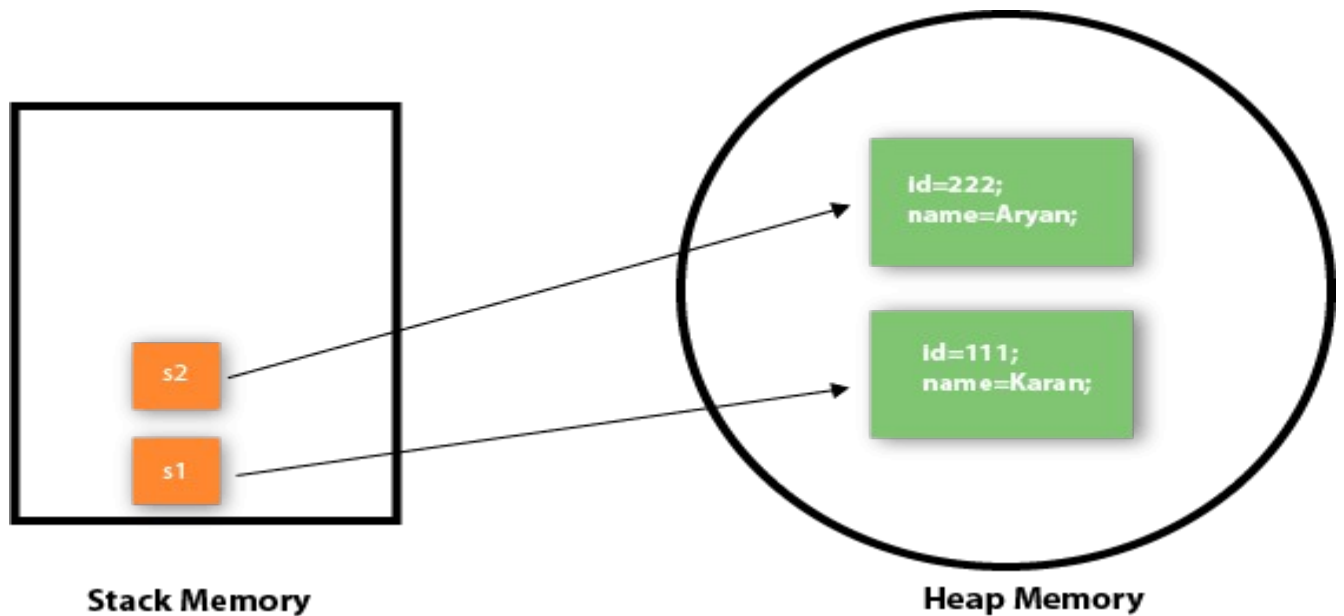
```
1. class Student{
2. int id;
3. String name;
4. }
5. class TestStudent3{
6. public static void main(String args[]){
7. //Creating objects
8. Student s1=new Student();
9. Student s2=new Student();
10. //Initializing objects
11. s1.id=101;
12. s1.name="Sonoo";
13. s2.id=102;
14. s2.name="Amit";
15. //Printing data
16. System.out.println(s1.id+" "+s1.name);
17. System.out.println(s2.id+" "+s2.name);
18. }
19.}
```

Test it Now

Output:

101 Sonoo

102 Amit



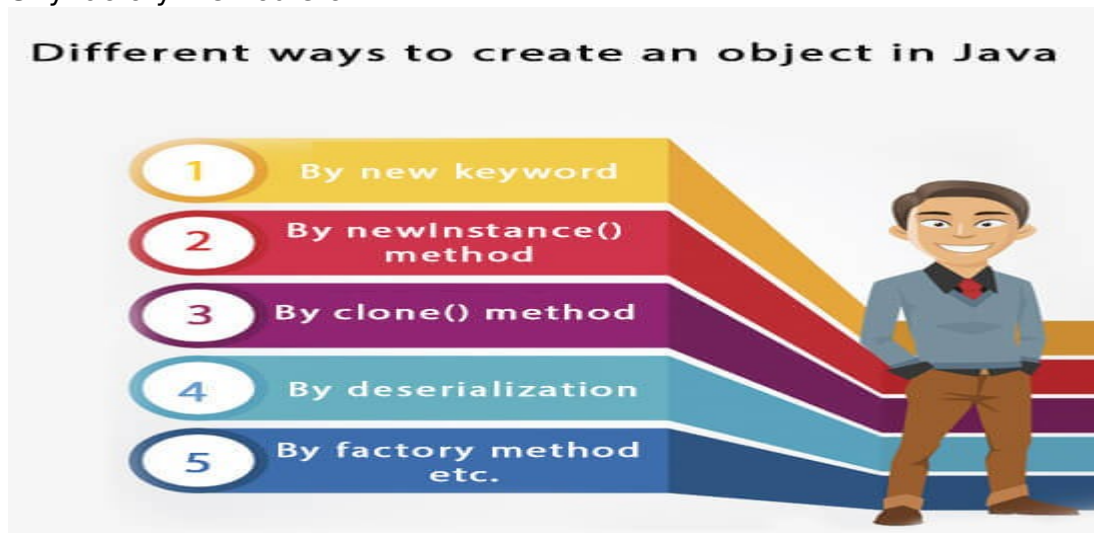
As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization

○By factory method etc.



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

1. **new** Calculation();//anonymous object

Calling method through a reference:

1. Calculation c=**new** Calculation();
2. c.fact(5);

Calling method through an anonymous object

1. **new** Calculation().fact(5);
- 2.

Let's see the full example of an anonymous object in Java.

1. **class** Calculation{
2. **void** fact(**int** n){
3. **int** act=1;
4. **for**(**in** i=1;i<=n;i++)
5. fact=fact*i;
6. }
7. System.out.println("factorial is "+fact);
8. }
9. **public static void** main(String args[]){
10. **new** Calculation().fact(5);//calling method with anonymous object

11.}

12.}

Output:

Factorial is 120

STATIC AND INSTANCE VARIABLE

```
public class VariableExample{

int myVariable; //instance variable

static int data = 30;//static variable


public static void main(String args[]){

    VariableExample obj = new VariableExample();

    System.out.println("Value of instance variable: "+obj.myVariable);

    System.out.println("Value of static variable: "+VariableExample.data);

}

}
```

kjdfks

Following are the notable differences between class (static) and instance variables.

Instance variables	Static (class) variables
Instance variables are declared in a class, but outside a method, constructor or any block.	Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.	Static variables are created when the program starts and destroyed when the program stops.
Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. <i>ObjectReference.VariableName</i> .	Static variables can be accessed by calling with the class name <i>ClassName.VariableName</i> .
Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.	There would only be one copy of each class variable per class, regardless of how many objects are created from it.

```
public class example{
    int x;//static variable
    static int y ;//static member variable
    public void fun1(){//instance member function
static class test{//valid class inside class
        public static string country = "INDIA";
    }
    public static void fun2(){//static member function
}
public class test2{
}
```

A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println()` method, for example, the system actually executes several statements in order to display a message on the console.

Static Method

A static method is also called a class method and is common across the objects of the class and this method can be accessed using class name as well.

Non-Static Method

Any method of a class which is not static is called non-static method or an instance method.

Following are the important differences between static and non-static method.

Sr. No .	Key	Static	Non-Static
1	Access	A static method can access only static members and can not access non-static members.	A non-static method can access both static as well as non-static members.
2	Binding	Static method uses compile time binding or early binding.	Non-static method uses run time binding or dynamic binding.
3	Overriding	A static method cannot be overridden being compile time binding.	A non-static method can be overridden being dynamic binding.
4	Memory allocation	Static method occupies less space and memory allocation happens once.	A non-static method may occupy more space. Memory allocation happens when method is invoked and memory is deallocated once method is executed completely.
5	Keyword	A static method is declared using static keyword.	A normal method is not required to have any special keyword.

Example of static vs non-static method

JavaTester.java

```
public class JavaTester {  
  
    public static void main(String args[]) {  
  
        Tiger.roar();  
  
        Tiger tiger = new Tiger();  
  
        tiger.eat();  
  
    }  
}  
  
class Tiger {  
  
    public void eat(){  
  
        System.out.println("Tiger eats");  
  
    }  
  
    public static void roar(){  
  
        System.out.println("Tiger roars");  
  
    }  
}
```

Output

```
Tiger roars  
Tiger eats
```

WRAPPER CLASS

As the name suggests, **wrapper classes are objects encapsulating primitive Java types.**

Each Java primitive has a corresponding wrapper:

- boolean, byte, short, char, int, long, float, double
- Boolean, Byte, Short, Character, Integer, Long, Float, Double

These are all defined in the `java.lang` package, hence we don't need to import them manually.

2. Wrapper Classes

“What's the purpose of a wrapper class?”. It's one of the most [common Java interview questions](#).

Basically, **generic classes only work with objects and don't support primitives**. As a result, if we want to work with them, we have to convert primitive values into wrapper objects.

For example, the Java Collection Framework works with objects exclusively. Long back when (prior to Java 5, almost 15 years back) there was no autoboxing and we, for example, couldn't simply call `add(5)` on a collection of `Integers`.

At that time, those **primitive values needed to be manually converted to corresponding wrapper classes** and stored in collections.

Today, with autoboxing, we can easily do `ArrayList.add(101)` but internally Java converts the primitive value to an `Integer` before storing it in the `ArrayList` using the `valueOf()` method.

3. Primitive to Wrapper Class Conversion

Now the big question is: how do we convert a primitive value to a corresponding wrapper class e.g. an `int` to `Integer` or a `char` to `Character`?

Well, we can either use constructor or static factory methods to convert a primitive value to an object of a wrapper class.

As of Java 9, however, constructors for many boxed primitives such as [Integer](#) or [Long](#) have been deprecated.

So **it's highly recommended to only use the factory methods on new code.**

Let's see an example of converting an int value to an Integer object in Java:

```
| Integer object = new Integer(1);
```

```
| Integer anotherObject = Integer.valueOf(1);
```

The `valueOf()` method returns an instance representing the specified int value.

It returns cached values which makes it efficient. It always caches values between -128 to 127 but can also cache other values outside this range.

Similarly, we can also convert boolean to Boolean, byte to Byte, char to Character, long to Long, float **convert String to Integer** to Float, and double to Double. Though if we have to then we need to use `parseInt()` method because String isn't a wrapper class.

On the other hand, **to convert from a wrapper object to a primitive value, we can use the corresponding method such as `intValue()`, `doubleValue()` etc:**

```
| int val = object.intValue();
```

A comprehensive reference can be found [here](#).

4. Autoboxing and Unboxing

In the previous section, we showed how to manually convert a primitive value to an object.

After Java 5, **this conversion can be done automatically by using features called autoboxing and unboxing.**

“Boxing” refers to converting a primitive value into a corresponding wrapper object. Because this can happen automatically, it's known as autoboxing.

Similarly, **when a wrapper object is unwrapped into a primitive value then this is known as unboxing.**

What this means in practice is that we can pass a primitive value to a method which expects a wrapper object or assign a primitive to a variable which expects an object:

```
| List<Integer> list = new ArrayList<>();  
| list.add(1); // autoboxing
```

```
| Integer val = 2; // autoboxing
```

In this example, Java will automatically convert the primitive `int` value to the wrapper.

Internally, it uses the `valueOf()` method to facilitate the conversion. For example, the following lines are equivalent:

```
| Integer value = 3;
```

```
| Integer value = Integer.valueOf(3);
```

Though this makes conversion easy and codes more readable, there are some cases where **we shouldn't use autoboxing e.g. inside a loop.**

Similar to autoboxing, unboxing is done automatically when passing an object to a method that expects a primitive or when assigning it to a primitive variable:

```
| Integer object = new Integer(1);  
| int val1 = getSquareValue(object); //unboxing  
| int val2 = object; //unboxing  
  
| public static int getSquareValue(int i) {  
|     return i*i;  
| }
```

Basically, **if we write a method that accepts a primitive value or wrapper object, we can still pass both values to them.** Java will take care of passing the right type e.g. primitive or wrapper depending upon context

Command line argument

Packages

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are –

- java.lang** – bundles the fundamental classes
- java.io** – classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals*–

```
/* File name : Animal.java */
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

Now, let us implement the above interface in the same package *animals* –

```
package animals;
/* File name : MammalInt.java */

public class MammalInt implements Animal {

    public void eat() {
        System.out.println("Mammal eats");
    }

    public void travel() {
        System.out.println("Mammal travels");
    }

    public int noOfLegs() {
        return 0;
    }

    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

```
}
```

Now compile the java files as shown below –

```
$ javac -d . Animal.java  
$ javac -d . MammalInt.java
```

import

The **import** keyword is used to import a package, class or interface.

The import statement can be used to import an entire package or sometimes import certain classes and interfaces inside the package. The import statement is written before the class definition and after the package statement(if there is any). Also, the import statement is optional.

A program that demonstrates this in Java is given as follows:

Example

Live Demo

```
import java.util.LinkedList;  
  
public class Demo {  
  
    public static void main(String[] args) {  
  
        LinkedList<String> l = new LinkedList<String>();  
  
        l.add("Apple");  
  
        l.add("Mango");  
  
        l.add("Cherry");  
  
        l.add("Orange");  
  
        l.add("Pear");  
  
        System.out.println("The LinkedList is: " + l);  
  
    }  
  
}
```

Output

```
The LinkedList is: [Apple, Mango, Cherry, Orange, Pear]
```

Now let us understand the above program.

The LinkedList class is available in the java.util package. The import statement is used to import it. Then LinkedList l is created. LinkedList.add() is used to add a elements to the LinkedList. Then the LinkedList is displayed.

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└─ root
    └─ mypack
        └─ MyPackageClass.java
```

To create a package, use the **package** keyword:

MyPackageClass.java

```
package mypack;

class MyPackageClass {

    public static void main(String[] args) {

        System.out.println("This is my package!");

    }

}
```

[Run Example »](#)

Save the file as **MyPackageClass.java**, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The **-d** keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

Access Modifiers in Java

There are two types of modifiers in Java: access modifiers and non-access modifiers.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

1.Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

2.Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

3.Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private int** data=40;
3. **private void** msg(){System.out.println("Hello java");}
4. }
- 5.
6. **public class** Simple{

```

7. public static void main(String args[]){
8. A obj=new A();
9. System.out.println(obj.data);//Compile Time Error
10.obj.msg();//Compile Time Error
11.}
12.}

```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```

1. class A{
2. private A(){//private constructor}
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6. public static void main(String args[]){
7. A obj=new A();//Compile Time Error
8. }
9. }

```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```

1. //save by A.java
2. package pack;
3. class A{
4. void msg(){System.out.println("Hello");}
5. }
1. //save by B.java
2. package mypack; //not same package
3. import pack.*;
4. class B{

```

```
5. public static void main(String args[]){
6. A obj= new A();//Compile Time Error
7. obj.msg();//Compile Time Error
8. }
9. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1. //save by A.java
2. package pack;
3. public class A{
4. protected void msg(){System.out.println("Hello");}
5. }
1. //save by B.java
2. package mypack;
3. import pack.*;
4.
5. class B extends A{
6. public static void main(String args[]){
7. B obj = new B();
8. obj.msg();
9. }
10.}
```

Output:Hello

4) Public

The `public` access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.     public void msg(){System.out.println("Hello");}
6. }
1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7.     public static void main(String args[]){
8.         A obj = new A();
9.         obj.msg();
10.    }
11.}
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2.     protected void msg(){System.out.println("Hello java");}
3. }
4.
5. public class Simple extends A{
6.     void msg(){System.out.println("Hello java");} //C.T.Error
7.     public static void main(String args[]){
8.         Simple obj=new Simple();
9.         obj.msg();
10.    }
11.}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

Following is an example that demonstrates how to achieve Encapsulation in Java –

```
/* File name : EncapTest.java */
public class EncapTest {
    private String name;
    private String idNum;
    private int age;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    public String getIdNum() {
        return idNum;
    }

    public void setAge( int newAge) {
        age = newAge;
    }
}
```

```

    public void setName(String newName) {
        name = newName;
    }

    public void setIdNum( String newId) {
        idNum = newId;
    }
}

```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program –

```

/* File name : RunEncap.java */
public class RunEncap {

    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge());
    }
}

```

This will produce the following result –

Output

```
Name : James Age : 20
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example

Create a constructor:

```
// Create a Main class

public class Main {

    int x; // Create a class attribute


    // Create a class constructor for the Main class

    public Main() {

        x = 5; // Set the initial value for the class attribute x

    }
```

```
public static void main(String[] args) {  
  
    Main myObj = new Main(); // Create an object of class Main (This  
    will call the constructor)  
  
    System.out.println(myObj.x); // Print the value of x  
  
}  
  
}
```

// Outputs 5

[Try it Yourself »](#)

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like `void`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an `int y` parameter to the constructor. Inside the constructor we set `x` to `y` (`x=y`). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of `x` to 5:

Example

```
public class Main {  
  
    int x;  
  
    public Main(int y) {  
  
        x = y;  
  
    }  
  
    public static void main(String[] args) {  
  
        Main myObj = new Main(5);  
  
        System.out.println(myObj.x);  
  
    }  
  
}
```

```
output = 5
```

[Try it Yourself »](#)

You can have as many parameters as you want:

Example

```
public class Main {  
  
    int modelYear;  
  
    String modelName;  
  
  
    public Main(int year, String name) {  
  
        modelYear = year;    modelName = name;  
  
    }  
    public static void main(String[] args) {  
  
        Main myCar = new Main(1969, "Mustang");  
  
        System.out.println(myCar.modelYear + " " + myCar.modelName);  
    }  
}
```

Outputs 1969 Mustang

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

In the example below, the **Car** class (subclass) inherits the attributes and methods from the **Vehicle** class (superclass):

Example

```
class Vehicle {  
  
    protected String brand = "Ford";           // Vehicle attribute  
  
    public void honk() {                        // Vehicle method  
  
        System.out.println("Tuut, tuut!");  
  
    }  
  
}  
  
class Car extends Vehicle {  
  
    private String modelName = "Mustang";      // Car attribute
```

```
public static void main(String[] args) {  
  
    // Create a myCar object  
  
    Car myCar = new Car();  
  
    // Call the honk() method (from the Vehicle class) on the myCar  
object  
  
    myCar.honk();  
  
    // Display the value of the brand attribute (from the Vehicle  
class) and the value of the modelName from the Car class  
  
    System.out.println(myCar.brand + " " + myCar.modelName);  
  
}  
  
}
```

Try it Yourself »

Did you notice the `protected` modifier in `Vehicle`?

We set the `brand` attribute in `Vehicle` to a `protected` access modifier. If it was set to `private`, the `Car` class would not be able to access it.

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, [Polymorphism](#), which uses inherited methods to perform different tasks.

The final Keyword

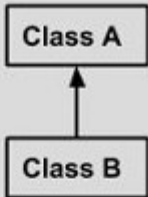
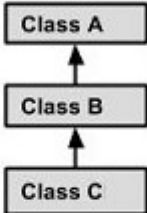
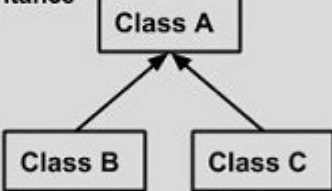
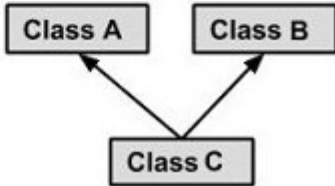
If you don't want other classes to inherit from a class, use the `final` keyword:

If you try to access a `final` class, Java will generate an error:

```
final class Vehicle { ... }  
  
class Car extends Vehicle { ... }
```

Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } public class C extends B { } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A { } public class B extends A { } public class C extends A { } </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A { } public class B { } public class C extends A,B { } </pre> <p>// Java does not support mutiple Inheritance</p>

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

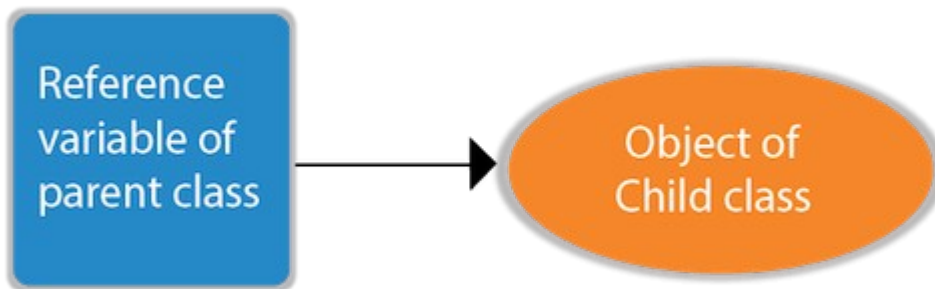
Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



1. **class** A{}
2. **class** B **extends** A{}
1. A a=**new** B();//upcasting
For upcasting, we can use the reference variable of class type or an interface type. For Example:

1. **interface** I{}
 2. **class** A{}
 3. **class** B **extends** A **implements** I{}
- Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

1. **class** Bike{
2. **void** run(){System.out.println("running");}
3. }
4. **class** Splendor **extends** Bike{
5. **void** run(){System.out.println("running safely with 60km");}
- 6.
7. **public static void** main(String args[]){

```
8.  Bike b = new Splendor();//upcasting
9.  b.run();
10. }
11.}
```

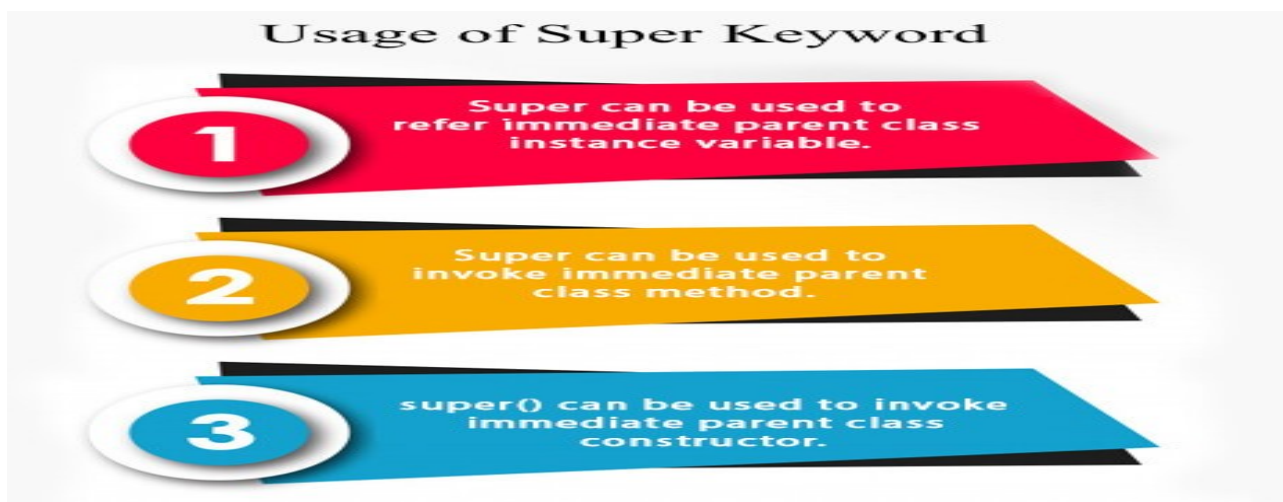
Super Keyword in Java

The `super` keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by `super` reference variable.

Usage of Java `super` Keyword

1. `super` can be used to refer immediate parent class instance variable.
2. `super` can be used to invoke immediate parent class method.
3. `super()` can be used to invoke immediate parent class constructor.



1) `super` is used to refer immediate parent class instance variable.

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
```



```

8. System.out.println(super.color);//prints color of Animal class
9. }
10.}
11.class TestSuper1{
12.public static void main(String args[]){
13.Dog d=new Dog();
14.d.printColor();
15.}}

```

Test it Now

Output:

black

white

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```

1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10.}
11.}
12.class TestSuper2{
13.public static void main(String args[]){
14.Dog d=new Dog();
15.d.work();
16.}}

```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. }
9. }
10. class TestSuper3{
11. public static void main(String args[]){
12. Dog d=new Dog();
13. }}
```

Test it Now

Output:

```
animal is created
```

STRING

Java StringBuffer class

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	is used to return the character at the specified position.
public int	length()	is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String	substring(int	is used to return the substring from the specified

	beginIndex, int endIndex)	beginIndex and endIndex.
--	------------------------------	--------------------------

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
1. class StringBufferExample{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1. class StringBufferExample2{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class StringBufferExample3{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavallo
6. }
7. }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1. class StringBufferExample4{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```

5) StringBuffer reverse() method

The reverse() method of StringBuiler class reverses the current string.

```
1. class StringBufferExample5{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by $(oldcapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class StringBufferExample6{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. }
10.}
```

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the

capacity by $(oldcapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class StringBufferExample7{
2. public static void main(String args[]){
3. StringBuffer sb=new StringBuffer();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. sb.ensureCapacity(10);//now no change
10.System.out.println(sb.capacity());//now 34
11.sb.ensureCapacity(50);//now (34*2)+2
12.System.out.println(sb.capacity());//now 70
13.}
14.}
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

Constructor	Description
StringBuilder()	creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	creates a string Builder with the specified string.
StringBuilder(int length)	creates an empty string Builder with the specified capacity as length.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char),

	append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	is used to reverse the string.
public int capacity()	is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string

```
1. class StringBuilderExample{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
```

2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
1. class StringBuilderExample2{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }
```

3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
1. class StringBuilderExample3{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavallo
6. }
7. }
```

4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
1. class StringBuilderExample4{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }
```


5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1. class StringBuilderExample5{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }
```

6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by $(oldcapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class StringBuilderExample6{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. }
10.}
```

7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by $(oldcapacity * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
1. class StringBuilderExample7{
2. public static void main(String args[]){
3. StringBuilder sb=new StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9. sb.ensureCapacity(10);//now no change
```

```
10.System.out.println(sb.capacity());//now 34
11.sb.ensureCapacity(50);//now (34*2)+2
12.System.out.println(sb.capacity());//now 70
13.}
14.}
```

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is a mechanism to achieve **abstraction**. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple **inheritance in Java**.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.



How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

1. **interface** <interface_name>{
- 2.
3. // declare constant fields
4. // declare methods that abstract
5. // by default.
6. }

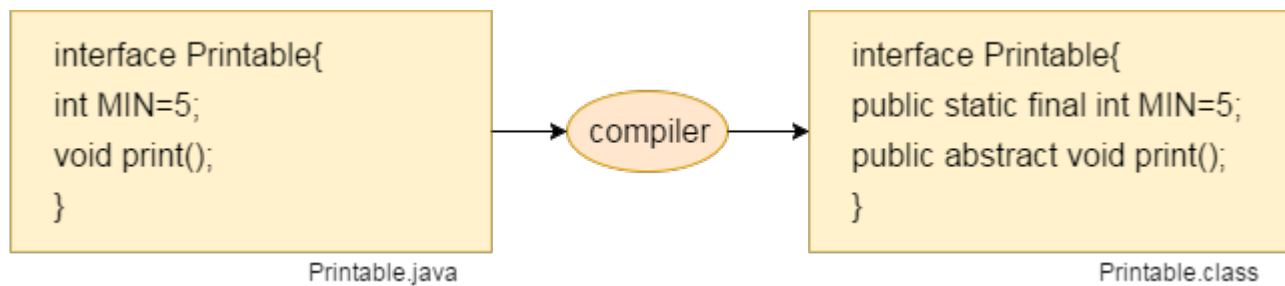
Java 8 Interface Improvement

Since **Java 8**, interface can have default and static methods which is discussed later.

Internal addition by the compiler

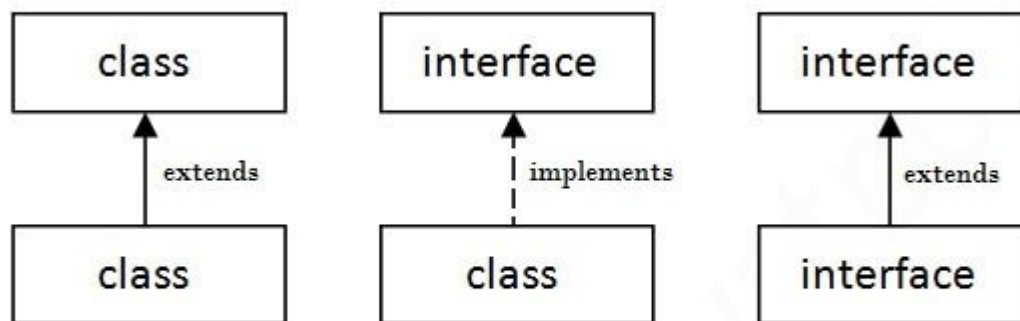
The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a class implements an interface.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. }
4. **class** A6 **implements** printable{
5. **public void** print(){System.out.println("Hello");}
- 6.
7. **public static void** main(String args[]){
8. A6 obj = **new** A6();
9. obj.print();
10. }
11. }

Test it Now

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
1. //Interface declaration: by first user
2. interface Drawable{
3. void draw();
4. }
5. //Implementation: by second user
6. class Rectangle implements Drawable{
7. public void draw(){System.out.println("drawing rectangle");}
8. }
9. class Circle implements Drawable{
10. public void draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. class TestInterface1{
14. public static void main(String args[]){
15. Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDr
    awable()
16. d.draw();
17. }}
```

Test it Now

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
1. interface Bank{
2. float rateOfInterest();
3. }
4. class SBI implements Bank{
```

```

5. public float rateOfInterest(){return 9.15f;}
6. }
7. class PNB implements Bank{
8. public float rateOfInterest(){return 9.7f;}
9. }
10. class TestInterface2{
11. public static void main(String[] args){
12. Bank b=new SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}

```

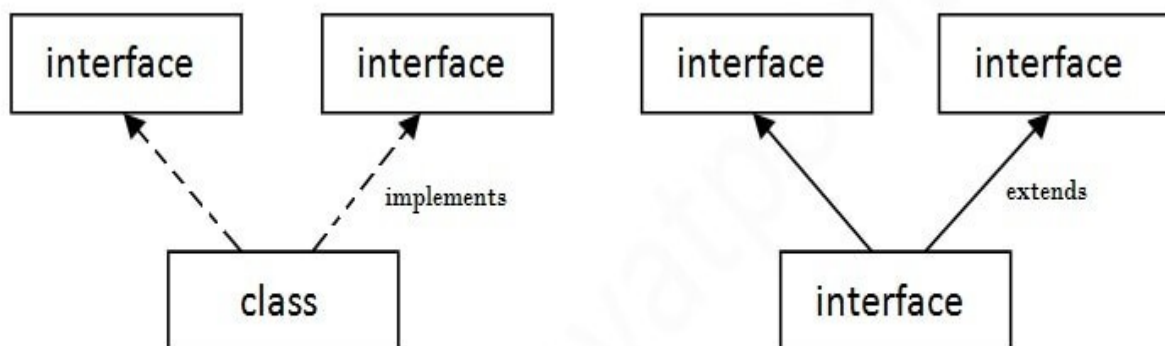
Test it Now

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```

1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void show();
6. }
7. class A7 implements Printable,Showable{
8. public void print(){System.out.println("Hello");}

```

```

9. public void show(){System.out.println("Welcome");}
10.
11.public static void main(String args[]){
12.A7 obj = new A7();
13.obj.print();
14.obj.show();
15. }
16.}

```

Test it Now

Output:Hello

Welcome

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of **class** because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```

1. interface Printable{
2. void print();
3. }
4. interface Showable{
5. void print();
6. }
7.
8. class TestInterface3 implements Printable, Showable{
9. public void print(){System.out.println("Hello");}
10.public static void main(String args[]){
11.TestInterface3 obj = new TestInterface3();
12.obj.print();
13. }
14.}

```

Test it Now

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
1. interface Printable{
2. void print();
3. }
4. interface Showable extends Printable{
5. void show();
6. }
7. class TestInterface4 implements Showable{
8. public void print(){System.out.println("Hello");}
9. public void show(){System.out.println("Welcome");}
10.
11.public static void main(String args[]){
12.TestInterface4 obj = new TestInterface4();
13.obj.print();
14.obj.show();
15. }
16.}
```

Test it Now

Output:

Hello

Welcome

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
1. interface Drawable{
2. void draw();
3. default void msg(){System.out.println("default method");}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
8. class TestInterfaceDefault{
9. public static void main(String args[]){
10.Drawable d=new Rectangle();
```



```
11.d.draw();
12.d.msg();
13.}}
```

Test it Now

Output:

```
drawing rectangle
```

```
default method
```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
1. interface Drawable{
2. void draw();
3. static int cube(int x){return x*x*x;}
4. }
5. class Rectangle implements Drawable{
6. public void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class TestInterfaceStatic{
10.public static void main(String args[]){
11.Drawable d=new Rectangle();
12.d.draw();
13.System.out.println(Drawable.cube(3));
14.}}
```

Test it Now

Output:

```
drawing rectangle
```

```
27
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, **Serializable**, Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

1. //How **Serializable** interface is written?
 2. **public interface** Serializable{
 3. }
-

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the [nested classes](#) chapter. For example:

```
interface printable{
```

1. **void** print();
2. **interface** MessagePrintable{
3. **void** msg();
4. }
5. }

Note :

Declaring abstract method static

If you declare a method in a class abstract to use it, you must override this method in the subclass. But, overriding is not possible with static methods. Therefore, an abstract method cannot be static.

If you still, try to declare an abstract method static a compile time error is generated saying “illegal combination of modifiers – abstract and static”.

Example

In the following Java program, we are trying to declare an abstract method static.

Live Demo

```
abstract class MyClass {  
  
    public static abstract void display();  
  
}  
  
public class AbstractClassExample extends MyClass{  
  
    public void display() {  
  
        System.out.println("This is the subclass implementation of the display  
method ");  
  
    }  
  
    public static void main(String args[]) {
```

```

        new AbstractClassExample().display();

    }

}

```

Compile time error

On compiling, the above program generates the following error.

```

AbstractClassExample.java:2: error: illegal combination of modifiers: abstract
and static
    public static abstract void display();
    ^
1 error

```

Declaring abstract method final

Similarly, you cannot override final methods in Java. But, in-case of abstract, you must override an abstract method to use it. Therefore, you cannot use abstract and final together before a method.

If, you still try to declare an abstract method final a compile time error is generated saying “illegal combination of modifiers: abstract and final”.

Example

In the following Java program, we are trying to declare an abstract method final.

```

public abstract class AbstractClassExample {
    public final abstract void display();
}

```

Compile time error

On compiling, the above program generates the following error.

```

AbstractClassExample.java:2: error: illegal combination of modifiers: abstract
and final
    public final abstract void display();
    ^
1 error

```

in abstract class u can declare abstract method as well as define non abstract method but in interface only u can declare abstract method but can not define non abstract without using the keyword default .

Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();

        numbers.add(5);

        numbers.add(9);

        numbers.add(8);

        numbers.add(1);

        numbers.forEach( (n) -> { System.out.println(n); } );

    }

}
```

How do we create a Functional interface?

An interface with exactly one abstract method is called Functional Interface. @FunctionalInterface annotation is added so that we can mark an interface as functional interface.

The major benefit of java 8 functional interfaces is that we can use lambda expressions to instantiate them and avoid using bulky anonymous class implementation.

```
@FunctionalInterface
interface Foo {
    void test();
}
```