**JAVA REVISION**

Static:- core advantage of the static method is that there is no need to create an object to invoke.

Variable: local, instance, static

Get memory at runtime when object is created

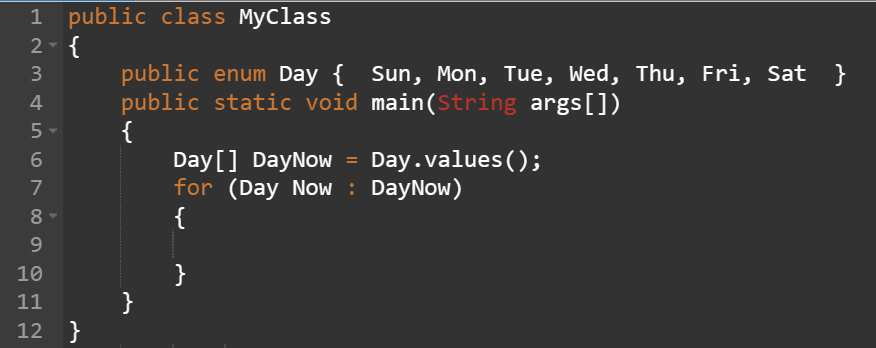
Shortening: (int)float\_var\_name

**Switch**

* Case value must be of switch expression type only
* Case value must be constant or literal not variables
* Switch expression type: byte, int, short, long, enum, string
* Can have default label

**Enum**

* Java enum is a class that represent the group of constants. (immutable such as final variables).
* We use the keyword enum and put the constants in curly braces separated by comma.



**Loop**

* for each loop:

for(data\_type variable : array\_name)

{

//code to be executed

}

* labelled for loop:

labelname:

for(initialization; condition; increment/decrement)

{

//code to be executed

break labelname

}

* infinite loop

for( ; ; ) while(true){} do{}while(true)

**OOPs**

* Object: an entity that has a state, identity, and behavior
* Class: blueprint from which we can create an individual object
* Inheritance: one object acquires all properties and behaviors of parent object, used to achieve runtime polymorphism.
* Polymorphism: If one task is performed in different ways, it is known as polymorphism.

In Java, we use method overloading and method overriding to achieve polymorphism.

* Abstraction: Hiding internal details and showing functionality is known as abstraction.

In Java, we use abstract class and interface to achieve abstraction.

* Encapsulation: Binding (or wrapping) code and data together into a single unit are known as encapsulation. A java class is the example of encapsulation.

***new***: allocate memory at runtime, get memory in heap

**Naming Convention**

Class: Noun

Interface: Adjective

Method: Verb

* If we have multiple classes in a java file, it good idea to save file name with class name which has main() method
* 3 ways to initialize object:

- by reference obj.name=”jhsbc”

- by method

- by constructor

* Ways to create object

- by new keyword

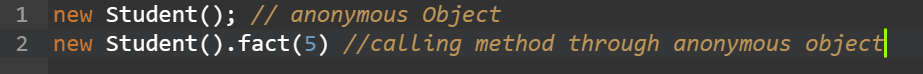
- by newInstance() method

- by clone() method

- by deserialization

- by factory method

* Anonymous object: object which has no reference





Two objects

**Constructor**

* At the time of calling constructor, memory for object is allocated in memory
* Everytime an object is created using new keyword, atleast one constructor is called
* Rules:

- name must be the same as its class name

- must have no explicit return type

- cannot be abstract, static, final, and synchronized

- can use access modifier while declaring

* The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
* The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example- Student s1=new Student(“ram”,12);

* Constructor can also be overloaded like Java methods.

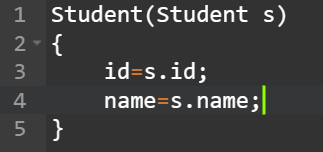
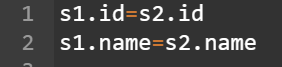
(Constructor overloading in Java is a technique of having more than one constructor with different parameter lists)

* There is no copy constructor in java. However there are many ways to copy the values of one object into another in Java. They are:

- By constructor

- By assigning the values of one object into another

- By clone() method of Object class

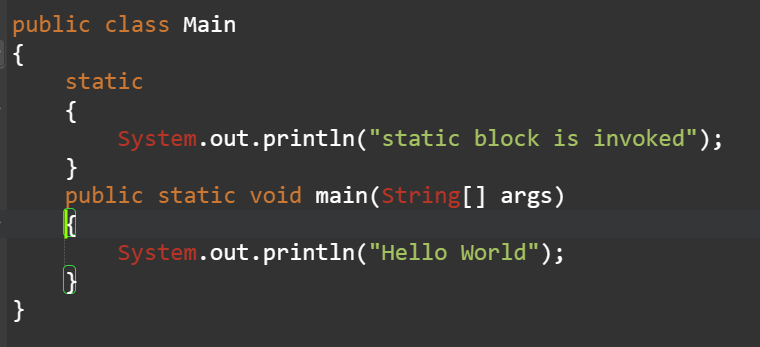
* Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the ***java.lang.reflect*** package.
* Use: object creation, starting a thread, calling a method, etc

**Static**

* Can apply with variables, methods, blocks, nested class
* Belongs to class than an instance of class
* Static variable gets memory only once in class area at time of class loading.
* Makes program memory efficient
* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class (ex- Student.staticMethod();)
* A static method can only access static data member and can change the value of it.
* A non-static method can access static data member and static function and can change the value of it as well.
* The static method cannot use non static data member or call non-static method directly.
* this and super cannot be used in static context.
* Java main() method is static because object is required to call method. If it were non-static method, JVM creates an object first and then call main() that will lead problem of extra memory location.
* Static block:

- Is used to initialize the static data member.

- It is executed before the main method at the time of classloading.



* Since JDK 1.7, it is not possible to execute java class without main() method

**Inheritance**

* Syntax: class A extends class B
* Inheritance represent IS-A relationship
* Types:

1. single inheritance
2. multilevel inheritance
3. hierarchical inheritance
4. hybrid inheritance

* To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

* When you extend a class, you inherit all of the methods and fields from the superclass, except for the constructors. This is because constructors are responsible for initializing the object, and the way that an object is initialized is specific to the class.
* Use:
  + For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
  + For Code Reusability.

**Aggregation**

* If a class have an entity reference, it is known as Aggregation.
* Aggregation represents HAS-A relationship.

class Employee{

int id;

String name;

Address address;//Address is a class

...

}

**Method overloading**

* If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
* There are two ways to overload the method in java
  + By changing number of arguments
  + By changing the data type
* In java method overloading is not possible by changing return type of method only because of ambiguity.
* We can overload main() method as well. We can have any number of main() methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.
* Method Overloading and Type Promotion



* + One type is promoted to another implicitly if no matching datatype is found.
  + If there are matching type arguments in the method, type promotion is not performed.
  + If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

**Method overriding**

* If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
* Use:
  + Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
  + Method overriding is used for runtime polymorphism
* Rules :
  + The method must have the same name as in the parent class
  + The method must have the same parameter as in the parent class.
  + There must be an IS-A relationship (inheritance).
* **A static method cannot be overridden** because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.
* We **cannot override main()** method because main() method is static method.
* We **cannot override constructor**.

**Super**

* 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:
  + super can be used to refer immediate parent class instance variable.
  + super can be used to invoke immediate parent class method.
  + super() can be used to invoke immediate parent class constructor.
* super() is added in each class constructor automatically by compiler if there is no super() or this().

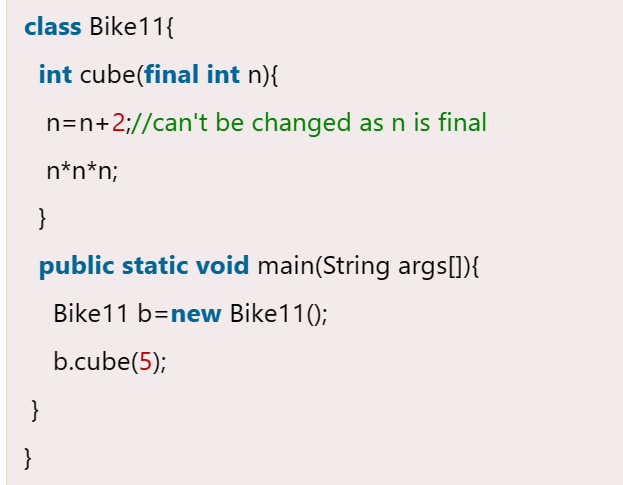
**Instance initializer block**

|  |
| --- |
| * **Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. |
| * The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

* Rules:
  + The instance initializer block is created when instance of the class is created.
  + The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
  + The instance initializer block comes in the order in which they appear.

**Final keyword**

* Final keyword is used in java to restrict the user
* Final can be: variable, method, class
* If you make any variable as final, you cannot change the value of final variable(It will be constant).
* If you make any method as final, you cannot override it.
* final method is inherited but you **cannot override** it
* If you make **any class as final, you cannot extend it.**
* A final variable that is not initialized at the time of declaration is known as blank final variable. It can be initialized only in constructor.
* A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.
* **We cannot declare constructor as final because constructor is never inherited.**
* If you declare any parameter as final, you cannot change the value of it.



**Polymorphism**

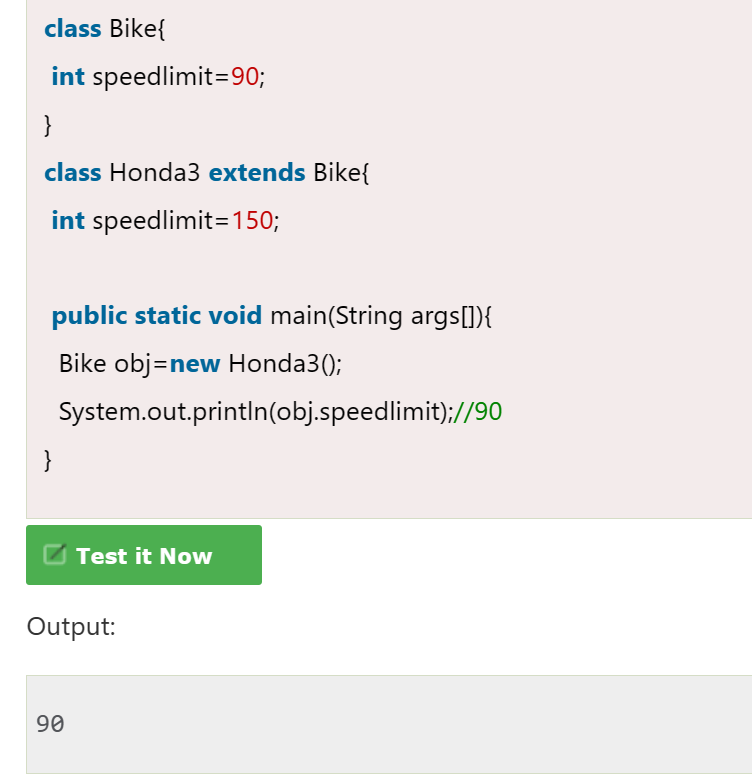
* There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
* If you overload a static method in Java, it is the example of **compile time polymorphism**.
* **Runtime polymorphism** 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.
* If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.



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.

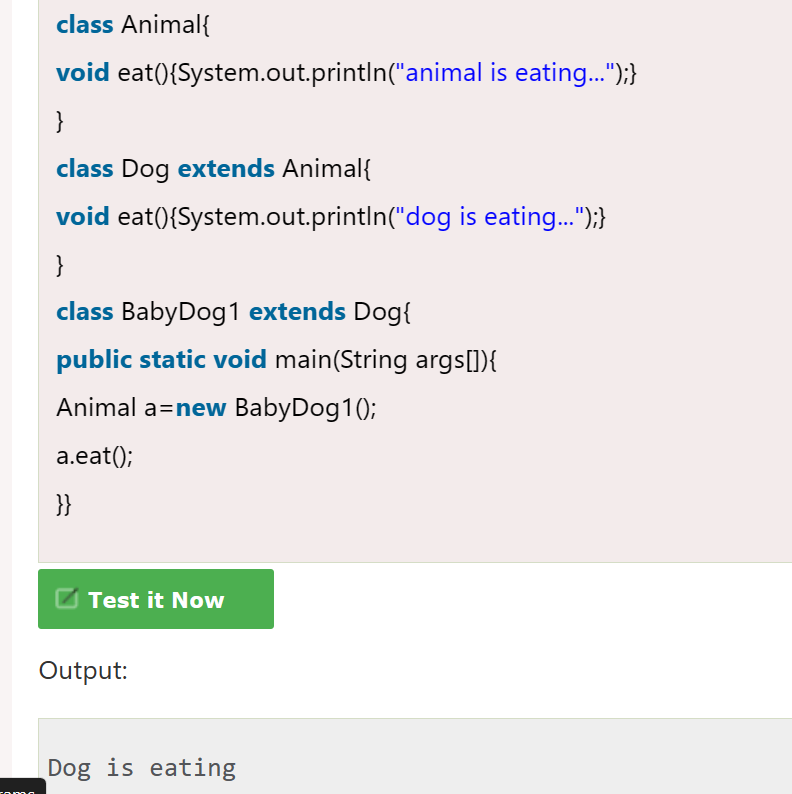
Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

* Runtime polymorphism cannot be achieved by data members.



A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given above, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

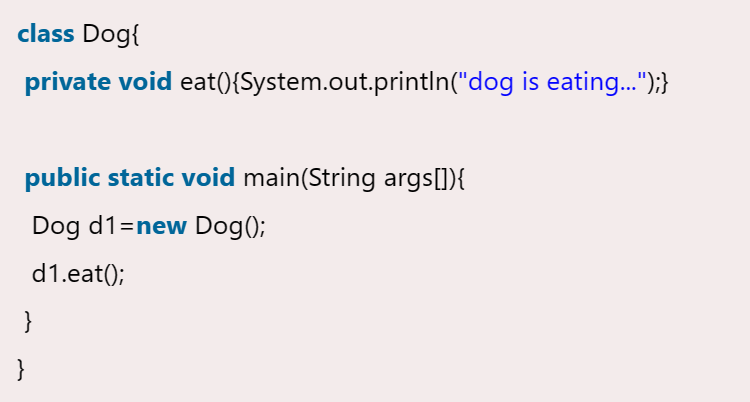


Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

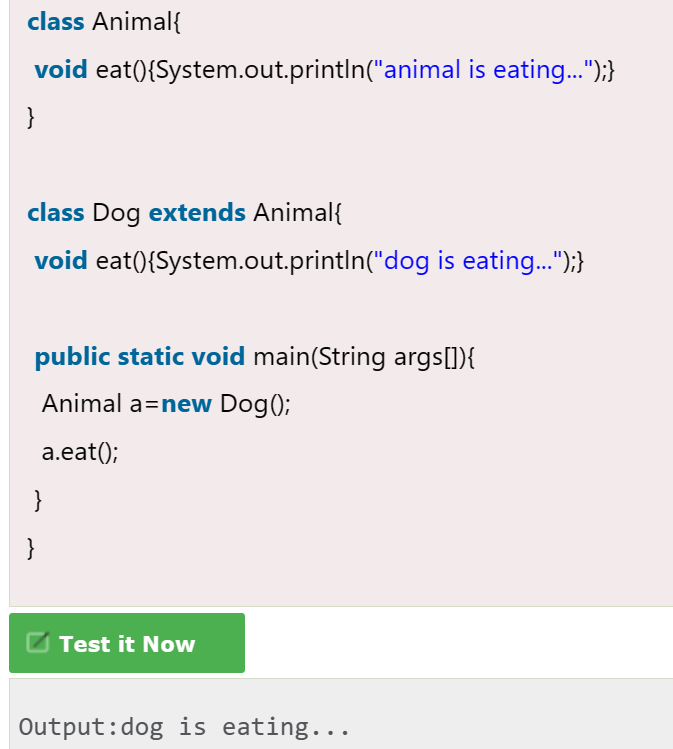
**Static and dynamic binding**

* Connecting a method call to the method body is known as binding.
* There are two types of binding:
  + Static binding (also known as early binding)
  + Dynamic binding (also known as late binding)
* **Static binding**: When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.



* **Dynamic binding**: When type of the object is determined at run-time, it is known as dynamic binding.



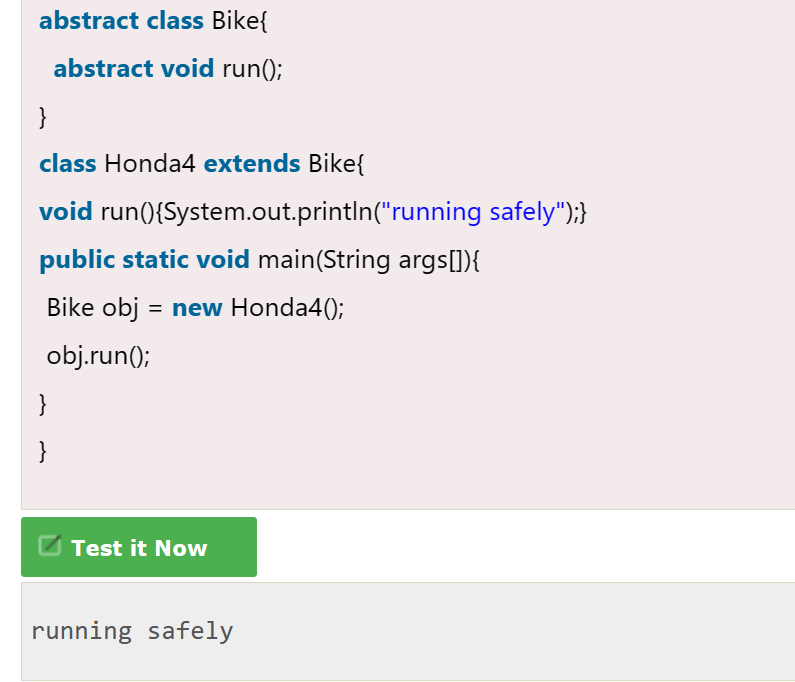
In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

**Abstraction**

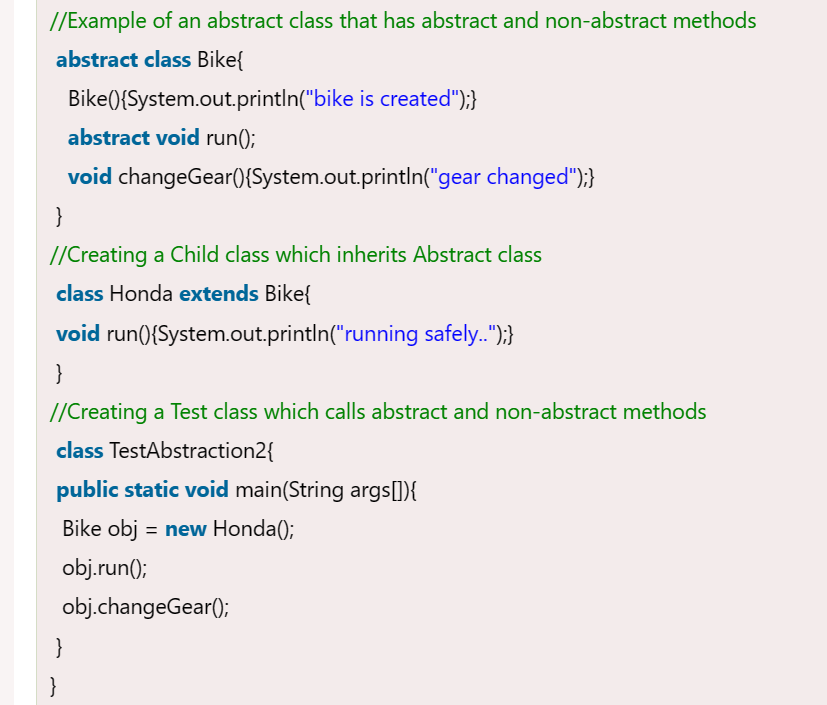
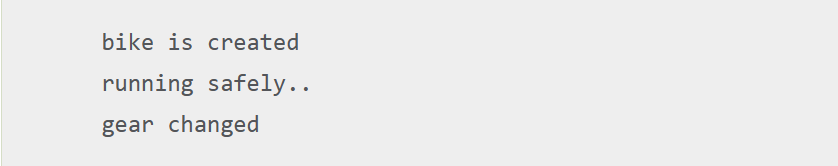
* **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
* **There are two ways to achieve abstraction in java**
  + **Abstract class (0 to 100%)**
  + **Interface (100%)**

**Abstract class**

* A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).
* Important points:
  + An abstract class must be declared with an abstract keyword.
  + It can have abstract and non-abstract methods.
  + It needs to be extended and its method implemented.
  + It cannot be instantiated.
  + It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
  + It can have final methods which will force the subclass not to change the body of the method.
  + If there is a abstract method in class, that class must be abstract
* In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.



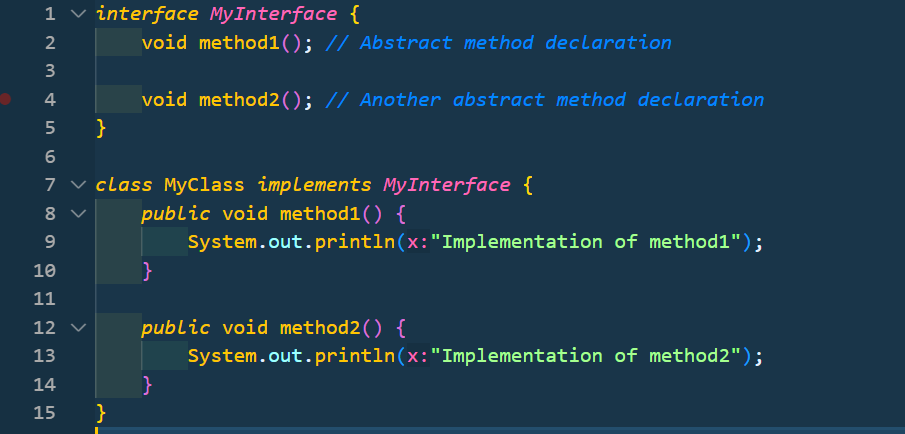
* An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

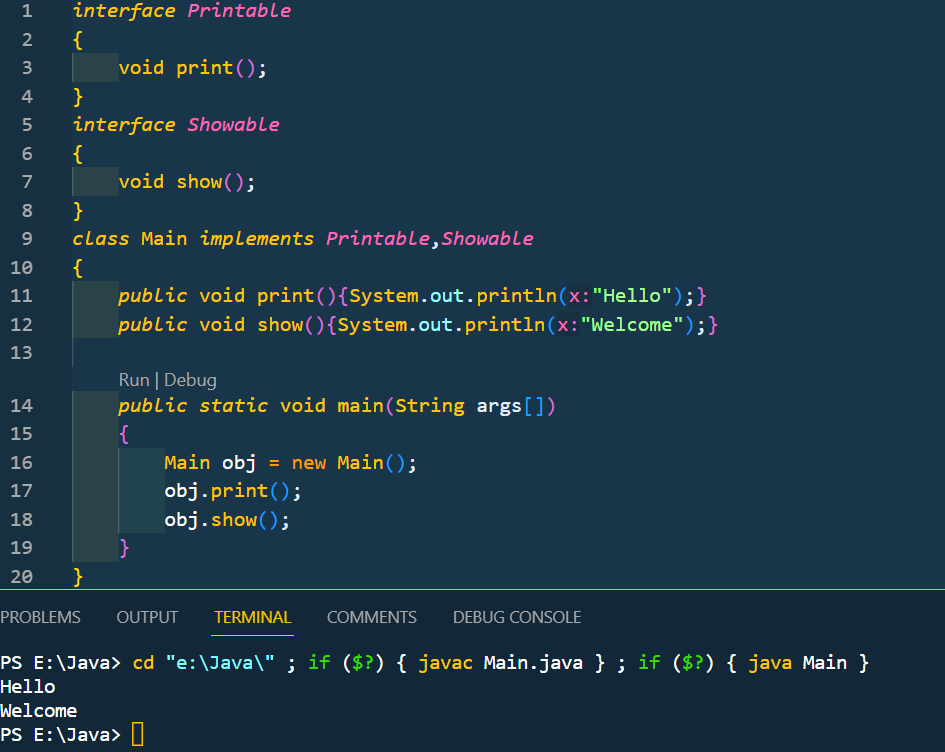
* If you are extending abstract class that have abstract method, you must either provide implementation of method or make this class abstract.

**Interface**

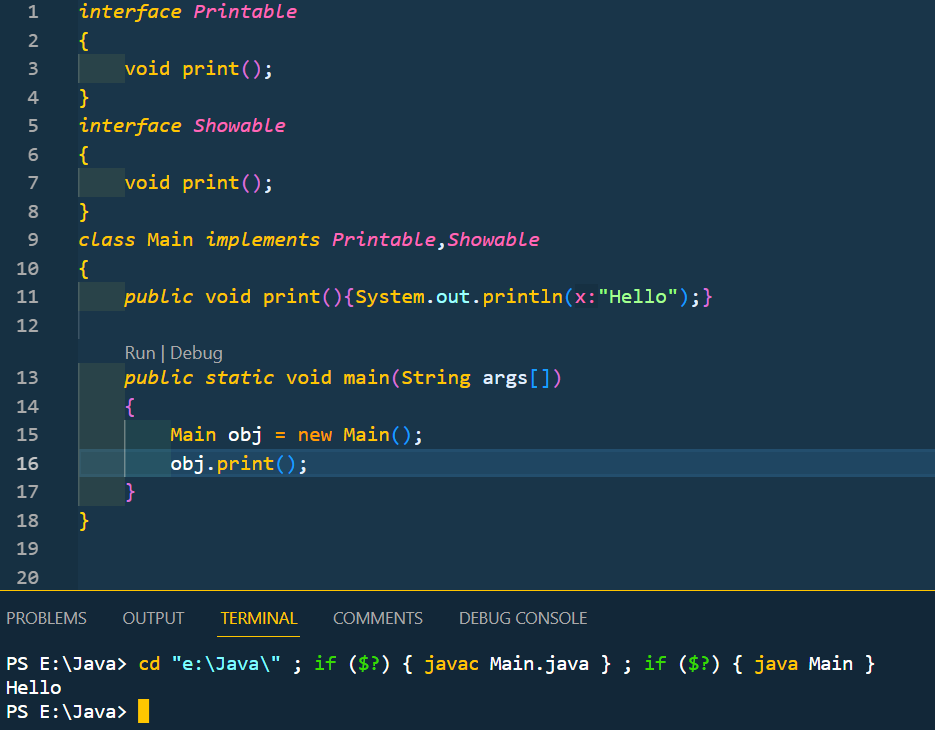
* An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
* There can be only abstract methods in the Java interface, not method body.
* All of the **methods in an interface are implicitly abstract**, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.) This is because interfaces are used to define the behaviour of a class, and the implementation of that behaviour is left up to the class that implements the interface.
* Interface can have abstract methods and variables but it cannot have a method body.
* It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).
* 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.
* There are mainly three reasons to use interface:
  + It is used to achieve abstraction.
  + By interface, we can support the functionality of multiple inheritance.
  + It can be used to achieve loose coupling.
* All of the fields in an interface are implicitly **public, static and final**. This is because the fields in an interface are meant to be constants that are shared by all of the classes that implement the interface.
* A class that implements an interface must implement all the methods declared in the interface.
* The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.
* A class extends another class, an interface extends another interface, but a **class implements an interface**.



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

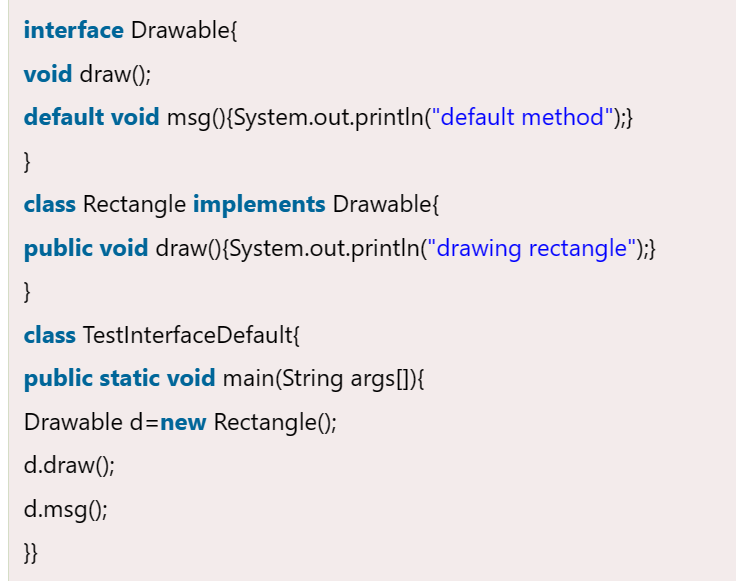


* Multiple inheritance is not supported in case of class due to ambiguity. However it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

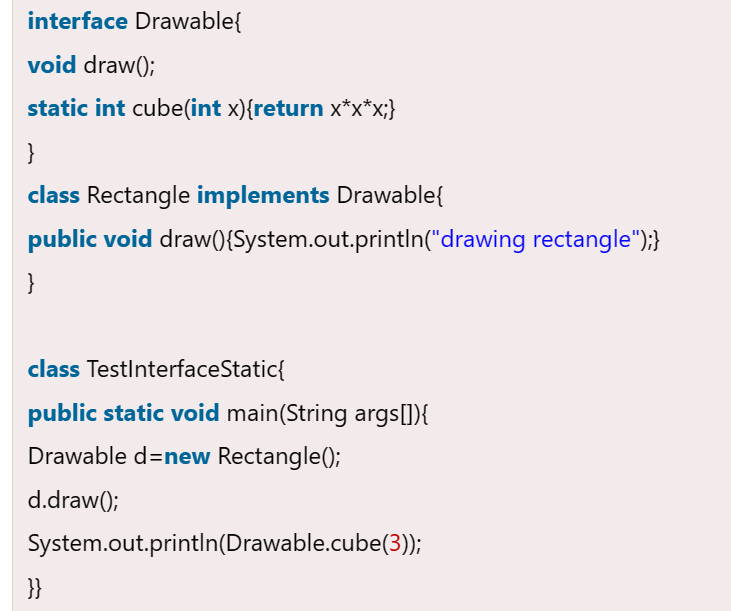


As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

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

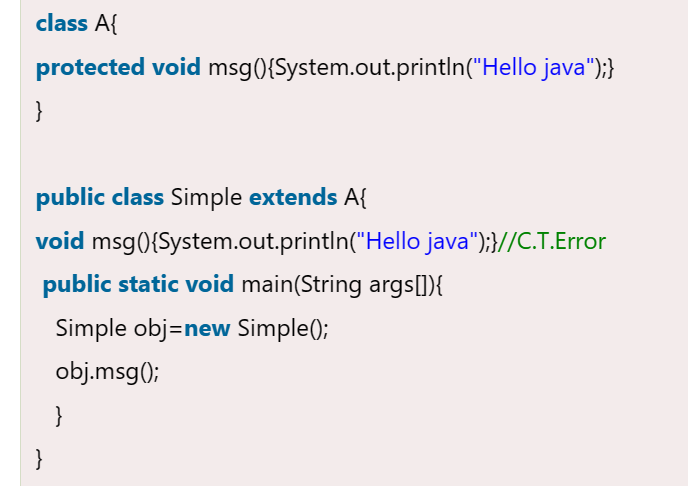


* Since Java 8, we can have static method in interface. Let's see an example:



**Access modifier**

* The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
* Access modifiers in java: private, default, protected, public
* There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc.
* A class cannot be private and protected except nested class
* If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.



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

**Encapsulation**

* We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
* The **Java Bean** class is the example of a fully encapsulated class.
* Advantages:
  + t is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
  + The encapsulate class is **easy to test**. So, it is better for unit testing.
  + It provides you the **control over the data**.

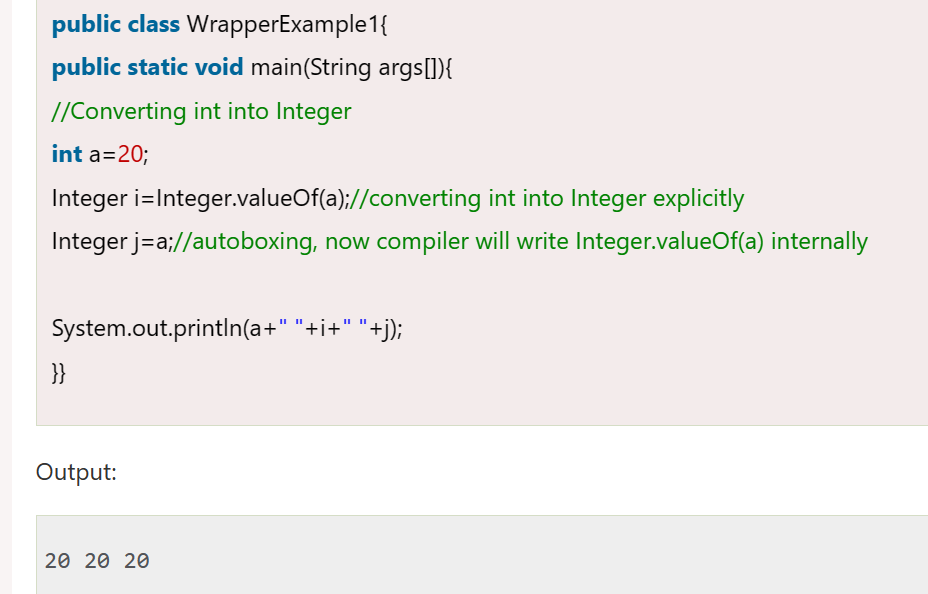
**Object class in java**

* In Java, the Object class is a built-in class provided by the Java platform. It serves as the root class for all other classes in Java, which means that every class in Java is implicitly a subclass of the Object class.
* The Object class provides a set of common methods and behaviors that are inherited by all other classes.

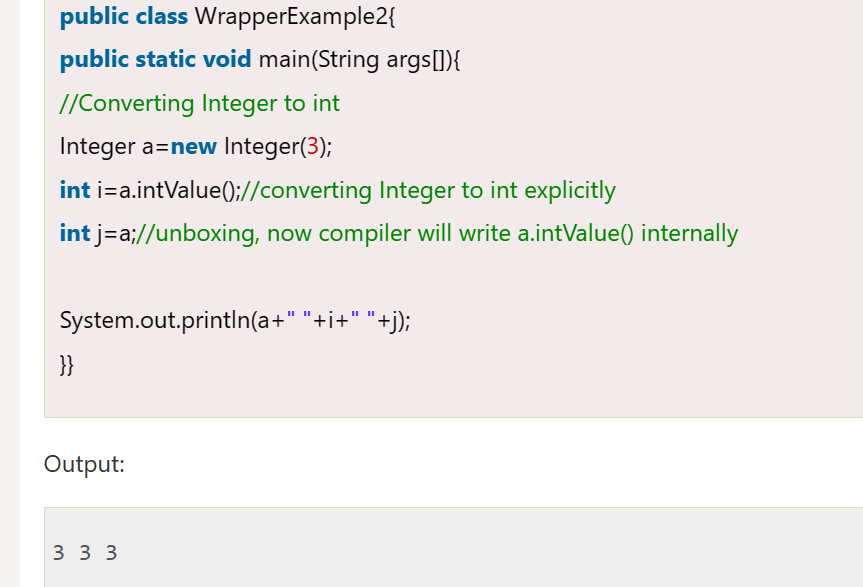
**Wrapper classes**

* The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.
* Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.
* Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.



* The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.



* In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed.

**strictfp keyword**

* Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.
* The strictfp keyword can be applied on methods, classes and interfaces.

1. strictfp class A{}//strictfp applied on class
2. strictfp interface M{}//strictfp applied on interface
3. class A{

strictfp void m(){}//strictfp applied on method

}

* The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

**Java array**

* In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces.
* Syntax to declare array:
  + dataType[] arr;
  + dataType []arr;
  + dataType arr[];
* Instantiation of array:
  + arrayRefVar=new dataType[size];
* examples:
  + int arr[] = new int[5];
  + we can declare, instantiate and initialize java array together:

int arr[] = {1,2,3,4,5};

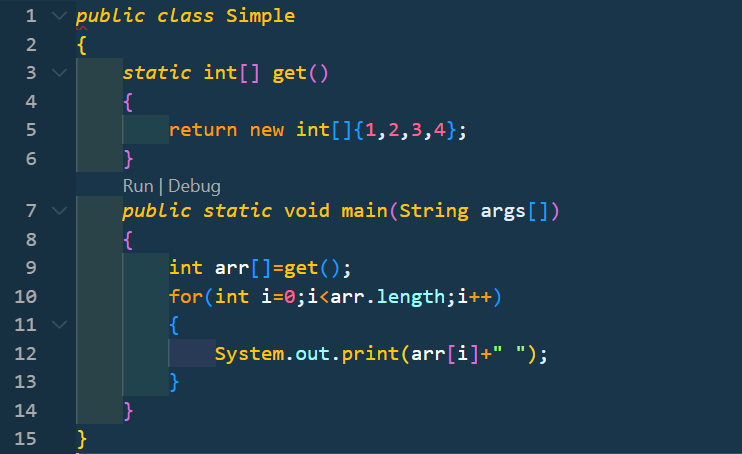
* + passing array to method:

methodName(arr);

* + receiving array in method:

void methodName(int arr[])

* + returning and receiving array from method:



* Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method:

printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method

* Multidimensional array:
  + dataType[][] arr;
  + dataType [][]arr;

Syntax of multidimensional array

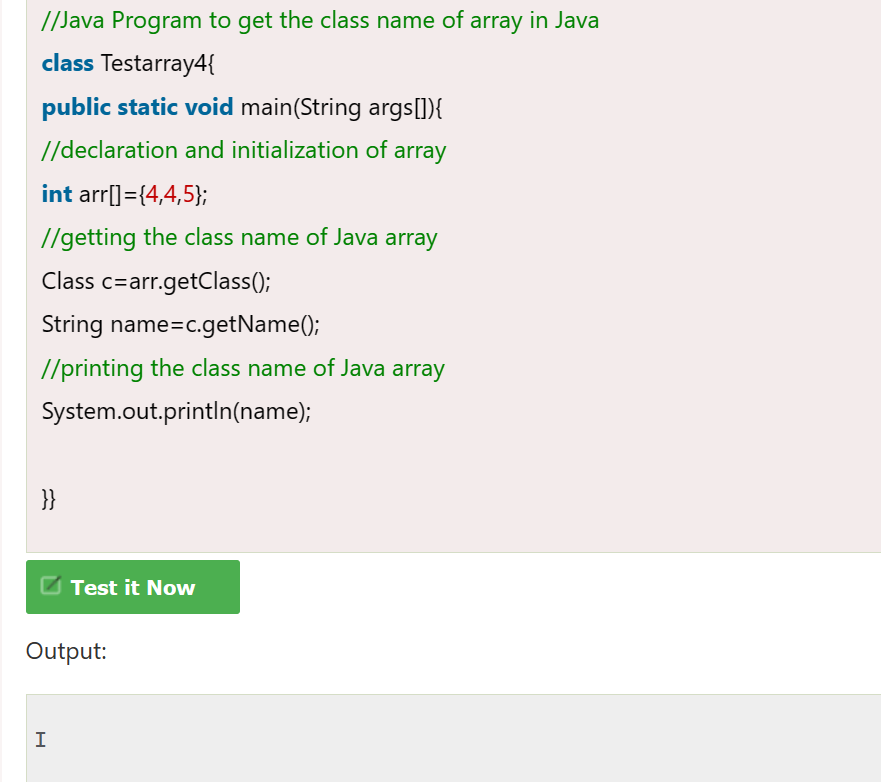
* + dataType arr[][];
  + dataType []arr[]

Instantiate multidimensional array

* + int arr[][]=new int[4][5];
* Jagged array: If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.



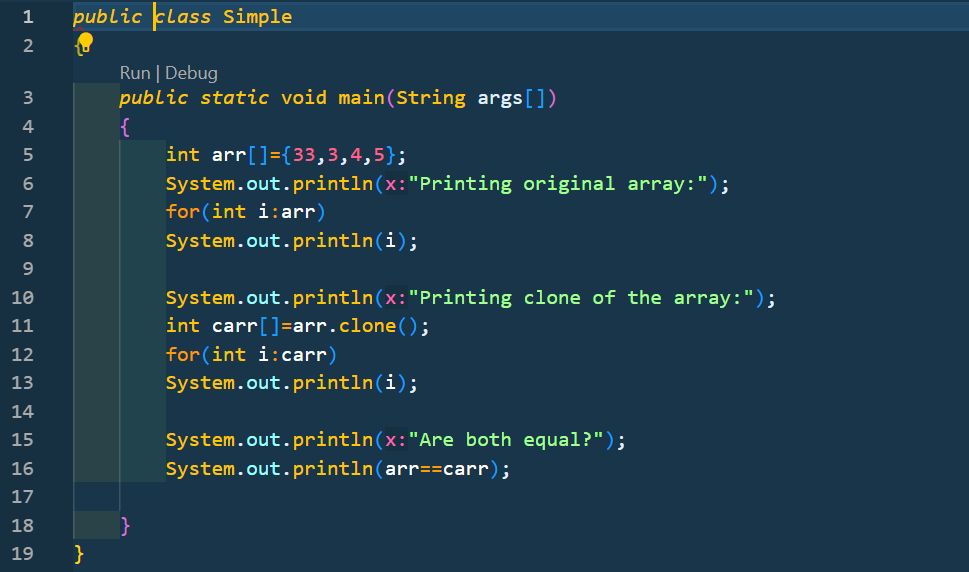
* In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.



* We can copy an array to another by the arraycopy() method of System class.
* **public** **static** **void** arraycopy(Object src, **int** srcPos,Object dest, **int** destPos,**int** length)

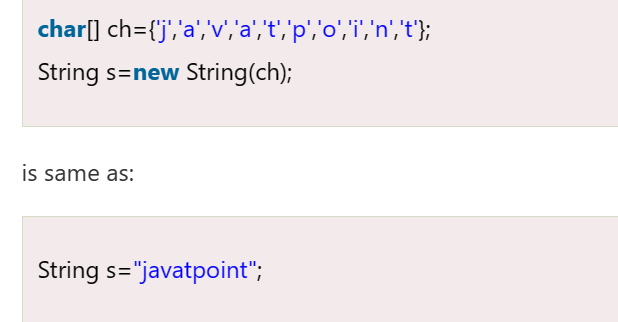


* Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.



**Java string**

* In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:



* **Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.
* The java.lang.String class implements Serializable, Comparable and CharSequence [interfaces](https://www.javatpoint.com/interface-in-java).
* The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in Java by using these three classes.
* The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.
* Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.
* There are two ways to create String object:

1. By string literal

String s="welcome";

* + Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
  + String objects are stored in special memory area known as string constant pool
  + String constant pool is a separate memory block inside the heap where the string object are stored by JVM.
  + To make Java more memory efficient, it uses the concept of string literal (because no new objects are created if it exists already in the string constant pool).

1. By new keyword

String s=**new** String("Welcome");//creates two objects and one r eference variable

* + In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).
* String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones.
* There are three ways to compare String in Java:

1. By Using equals() Method: It compares values of string for equality. equals() and equalsIgnoreCase()
2. By Using == Operator: The == operator compares references not values.
3. By compareTo() Method: compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

* There are two ways to concatenate strings in Java:

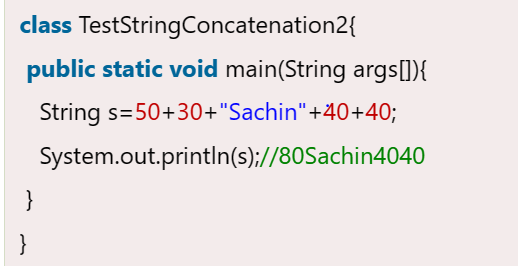
1. By + (String concatenation) operator
   * In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's append method.

String s="Sachin"+" Tendulkar";

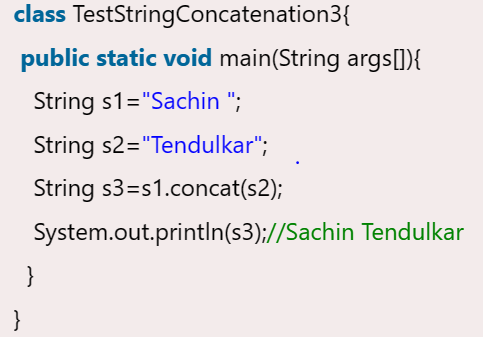
The **Java compiler transforms** above code to this:

String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

* + The String concatenation operator can concatenate not only String but primitive values also. For Example:

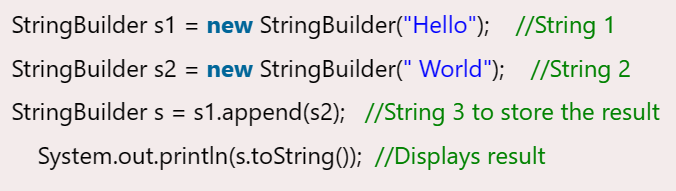


1. By concat() method



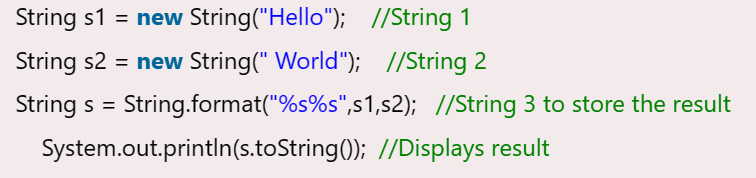
* String concatenation using StringBuilder class

StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastet way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.



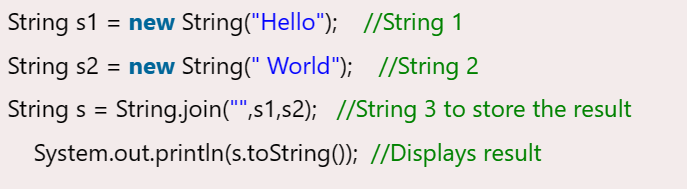
### String concatenation using format() method

String.format() method allows to concatenate multiple strings using format specifier like %s followed by the string values or objects.



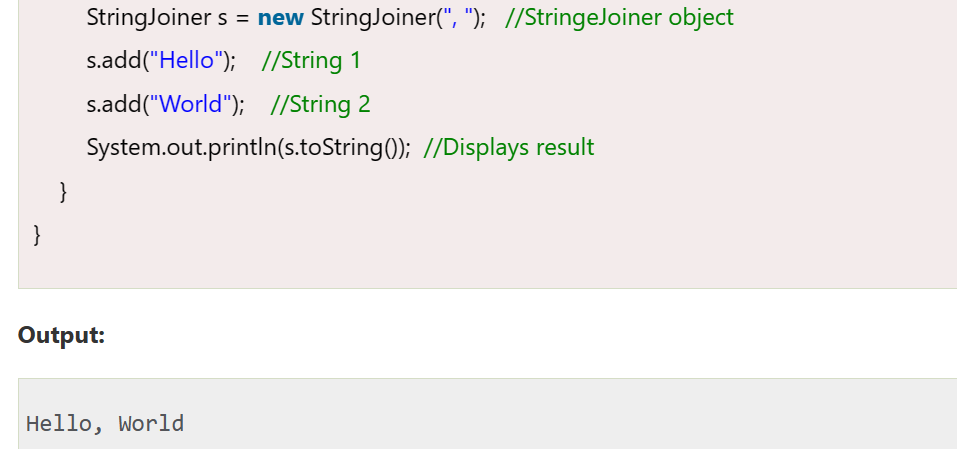
### String concatenation using String.join() method (Java Version 8+)

The String.join() method is available in Java version 8 and all the above versions. String.join() method accepts arguments first a separator and an array of String objects.



### String concatenation using StringJoiner class (Java Version 8+)

StringJoiner class has all the functionalities of String.join() method. In advance its constructor can also accept optional arguments, prefix and suffix.

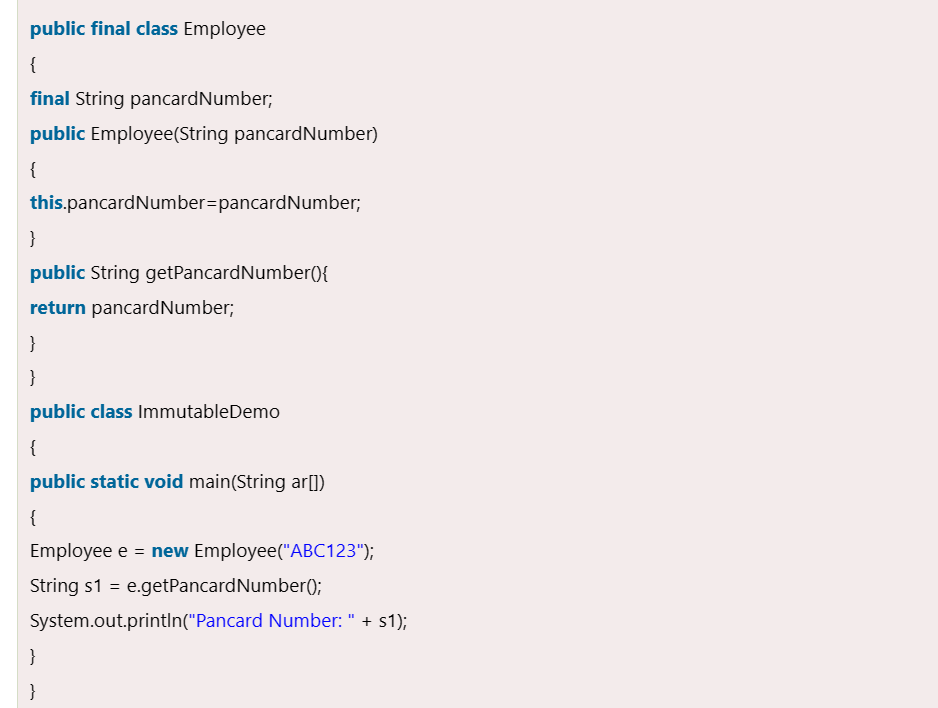


### String concatenation using Collectors.joining() method (Java (Java Version 8+)

The Collectors class in Java 8 offers joining() method that concatenates the input elements in a similar order as they occur.

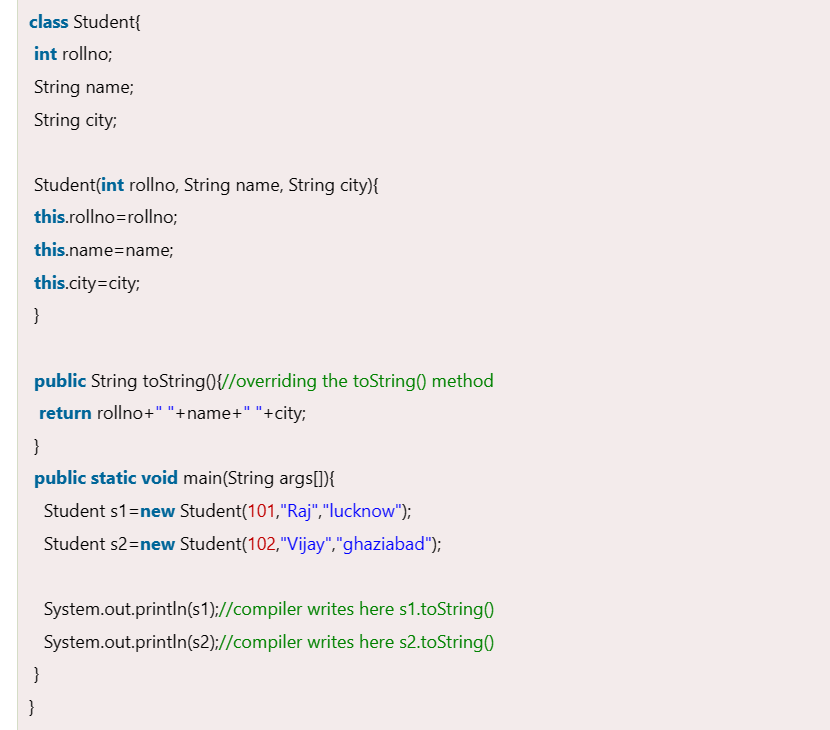


* Java **StringBuffer** class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.
* Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously.
* Example: StringBuffer sb=**new** StringBuffer("Hello ");
* 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.
* Example: StringBuilder sb=**new** StringBuilder("Hello ");
* It is not thread safe.
* There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable.
* We can also create immutable class by creating final class that have final data members as the example given below:



The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.
* If you want to represent any object as a string, **toString() method** comes into existence.
* The toString() method returns the String representation of the object.
* If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.
* By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.



**I/O operation**

* **InputStreamReader** class: To accept the data from keyboard, i.e. System.in, we need to connect it to an input stream as some input stream is needed to read data.
  + Connects the keyboard to an input stream object. Here we can use InputStreamReader that can read data from the keyboard.

InputStreamReader isr=new InputStreamReader(System.in)

In this statement, we are creating InputStreamReader object and connecting the keyboard (System.in) to it.

* + Connect InputStreamReader to BufferedReader, which is another input type of stream. We are using BufferedReader as it has got methods to read data properly, coming from the stream.

BufferedReader br = new BufferedReader(isr);

Here, we are creating BufferedReader object (br) and connecting the InputStreamReader object(isr) to it

import java.io.\*;

import java.lang.\*;

class Program1 {

public static void main(String args[]) throws IOException {

String str;

InputStreamReader isr= new InputStreamReader(System.in);

BufferedReader br = new BufferedReader(isr);

System.out.println("Enter the value for A");

char ch = (char) br.read();

} }

String str = br.readLine();

Now, the number is in str, i.e. in form of a String. This should be converted into an int by using parseInt() method of Integer class as:

int n = Integer.parseInt(br.readLine()) // String to int conversion

Simirlarly

float f = Float.parseFloat(br.readLine()) // String to float conversion

double d = Double.parseDouble(br.readLine())//String to double conversion

long L = Long.parseLong(br.readLine())//String to Long conversion

short s = Short.parseShort(br.readLine())//String to short conversion

byte b = Short.parseByte(br.readLine())//String to Byte conversion

* **Scanner** class: There are various ways to read input from the keyboard, the **java.util.Scanner** class is one of them. The Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

To read input from keyboard, we can use Scanner class as:

Scanner sc = new Scanner(System.in)

There is a list of commonly used Scanner class methods:

• public String next(): to read a string

• public String nextLine():to read a string till the end of the line..

• public byte nextByte(): to read byte value.

• public short nextShort():to read short value.

• public int nextInt():to read int value.

• public long nextLong():to read long value.

• public float nextFloat():to read float value.

• public double nextDouble():to read double value.

• public String next().charAt(0): to read a single character

* **Console** class (I/O) It has been becoming a preferred way for reading user’s input from the Keyboard. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Example of Console class that reads name of user:

import java.io.\*;

class A{

public static void main(String args[]){

Console c=System.console();

System.out.println("Enter your name");

String n=c.readLine();

System.out.println("Welcome "+n);

} }

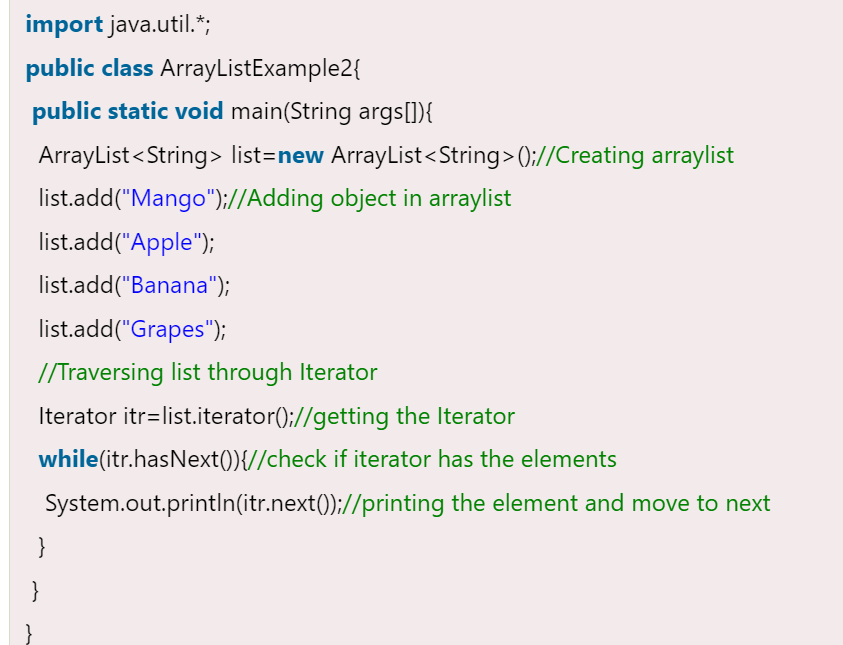
**JAVA COLLECTIONS**

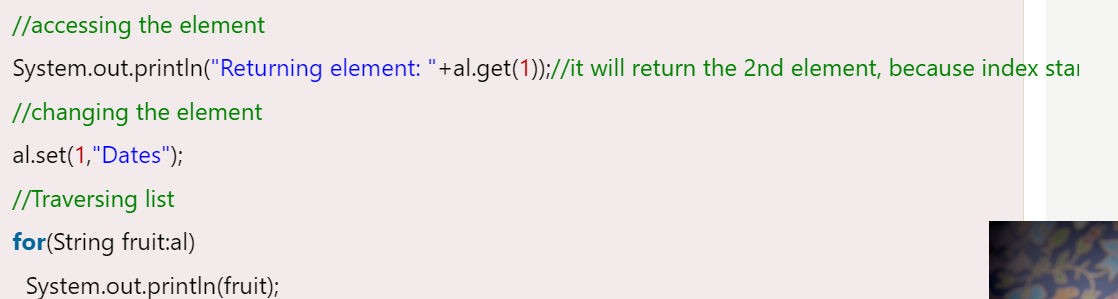
**ArrayList**

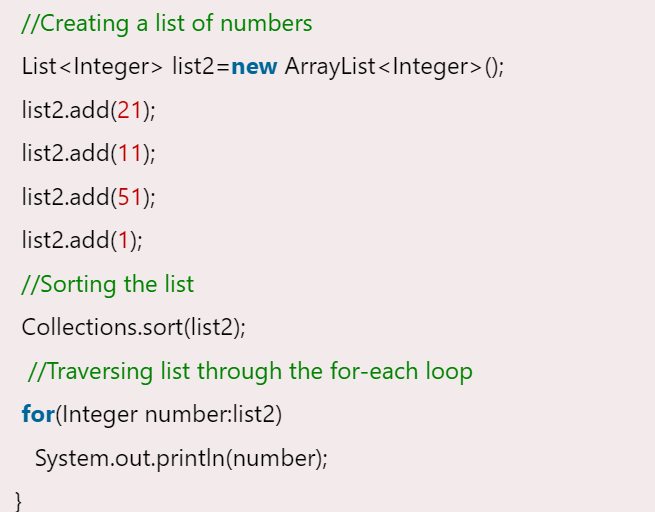
* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
* Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.
* We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

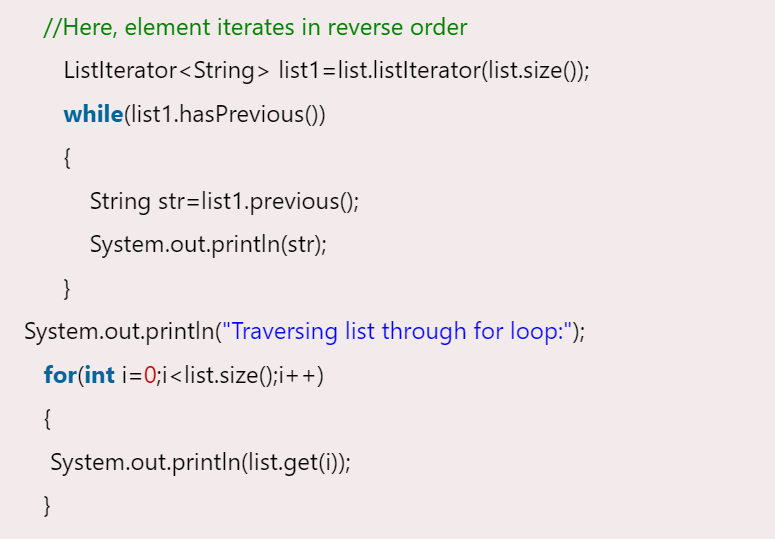
ArrayList<**int**> al = ArrayList<**int**>(); // does not work

ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine









### User-defined class objects in Java ArrayList

**class** Student{

**int** rollno;

  String name;

**int** age;

  Student(**int** rollno,String name,**int** age){

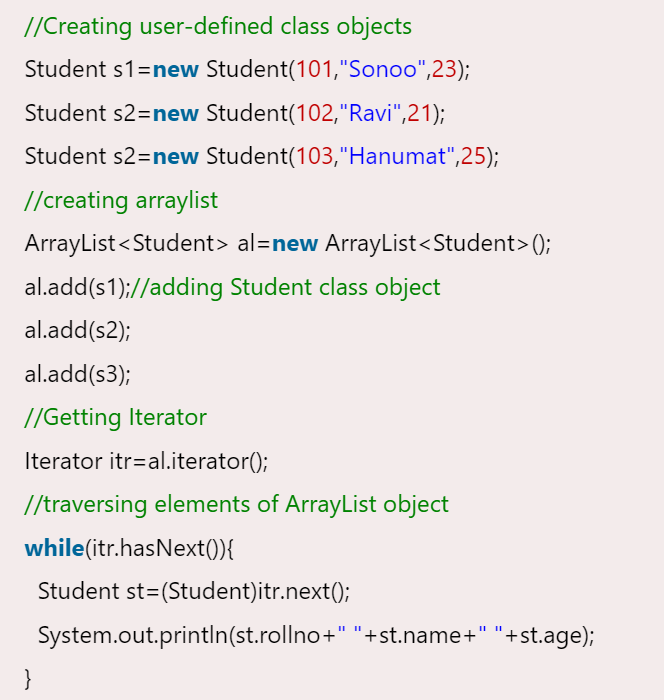
**this**.rollno=rollno;

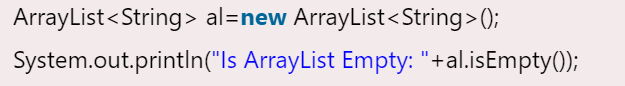
**this**.name=name;

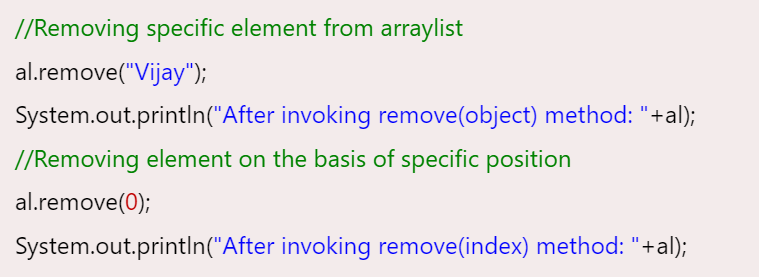
**this**.age=age;

  }

}

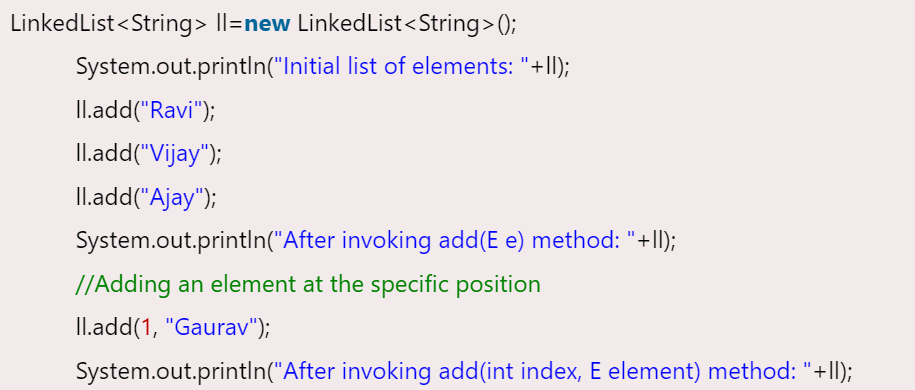


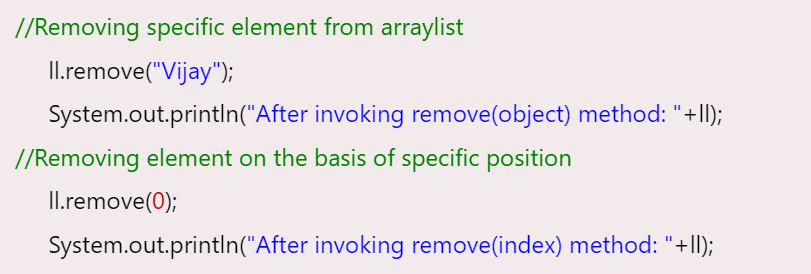


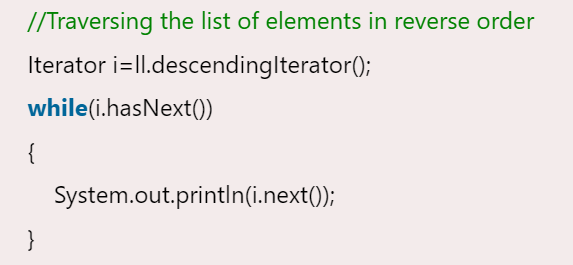


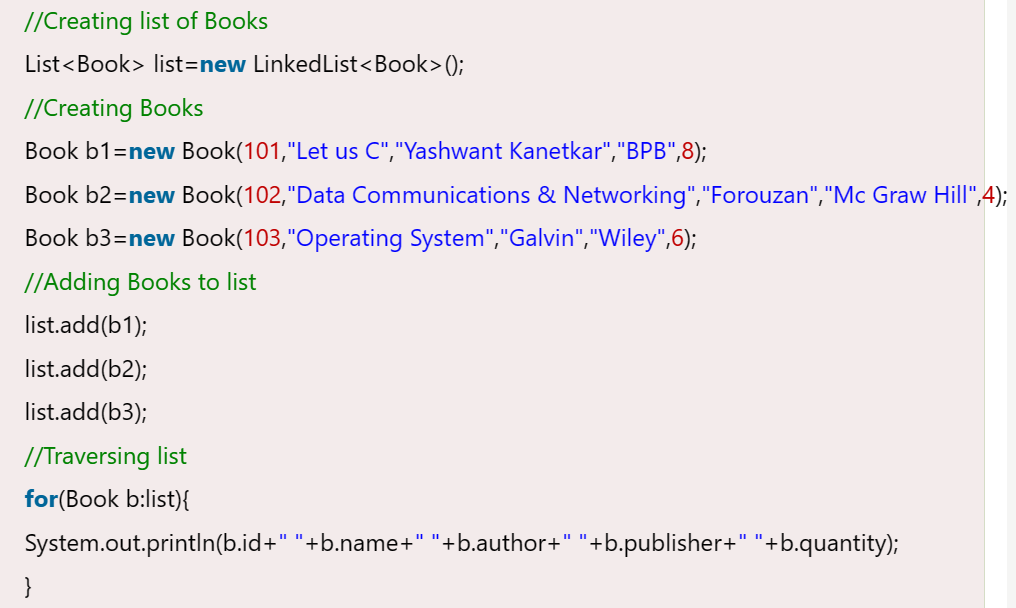
**LinkedList**

* Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.
* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.







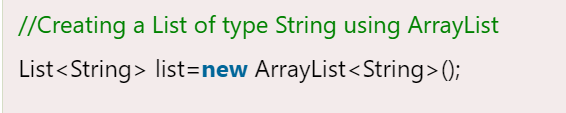


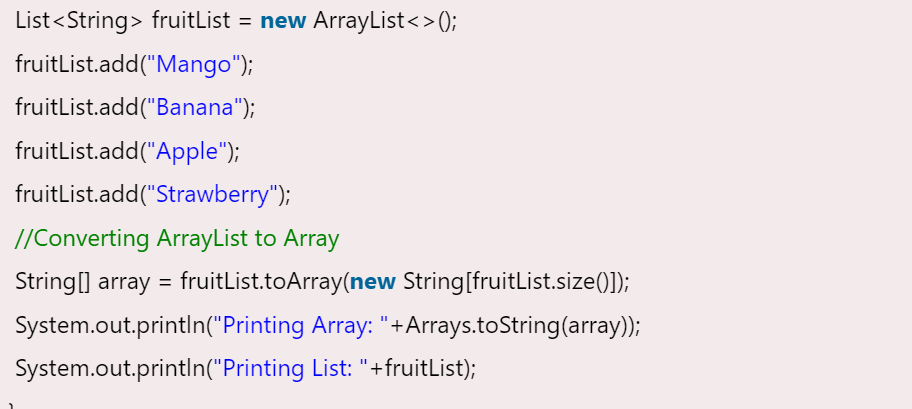
**Java List**

**List** in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:





**ListIterator Interface is used to traverse the element in a backward and forward direction.**

