**CLASS**

A **class** in Java is a blueprint or template for creating objects.

(or)

class is a [data type](https://www.tutorialspoint.com/java/java_basic_datatypes.htm) of an object type.

Each class has its [methods](https://www.tutorialspoint.com/java/java_class_methods.htm) and [attributes](https://www.tutorialspoint.com/java/java_class_attributes.htm) that can be accessed and manipulated through the objects.

**Creating (Declaring) a Java Class**

To create (declare) a class, you need to use [*access modifiers*](https://www.tutorialspoint.com/java/java_access_modifiers.htm) followed by **class** keyword and *class\_name*.

**Object**

we will create object by using 'new' operator creating an object is nothing but allocating some memory space to class variables,methods,blocks and constructor

**Types of Class Variables**

A class can contain any of the following variable types.

* **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class/Static variables** -Class variables also known as static variables are declared with the **static keyword** within a class, but outside a method, constructor or a block.

**Basic Questions:**

**1. What is a class in Java?**

* A class in Java is a blueprint or template for creating objects. It defines the properties (fields/variables) and behaviors (methods) of an object.

**2. How do you define a class in Java?**

* A class is defined using the class keyword:

**3. What is the difference between a class and an object?**

* A class is a blueprint, while an object is an instance of a class.
* For class no memory is allocated ,but for object memory is allocated.

**4.What are the different types of classes in Java?**

* Regular Class (Standard class with fields and methods)
* Abstract Class (Cannot be instantiated, used for inheritance)
* Final Class (Cannot be extended)
* Static Class (Only allowed as a static nested class)
* Nested/Inner Class (Defined inside another class)
* Anonymous Class (A class without a name, often used with interfaces)

**3. What is the difference between an Abstract Class and an Interface?**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| Methods | Can have both abstract & concrete methods | Only abstract methods (before Java 8), can have default & static methods (Java 8+) |
| Variables | Can have instance variables | Only constants (public static final variables) |
| Inheritance | Supports single inheritance (extends) | Supports multiple inheritance (implements) |
| Constructor | Can have a constructor | Cannot have a constructor |

**. What is the this and super keyword in Java?**

**this keyword:**

* Refers to the **current instance** of the class.
* Used to differentiate **instance variables** from **local variables**.
* Can be used to **call another constructor** in the same class.

**super keyword:**

* Used to refer to the **parent class**.
* Can call **superclass constructors** or **superclass methods**.

**How does Java handle memory allocation for objects and classes?**

* **Heap Memory**: Stores objects.
* **Stack Memory**: Stores method calls and local variables.
* **Method Area (Class Area)**: Stores class-level data (static variables, class definitions).
* **Garbage Collection**: Automatically removes unused objects.

**What is the difference between == and .equals() when comparing objects?**

* + == compares object references (memory addresses), while .equals() compares object content (values).

**7. What is the final keyword in a class?**

The final keyword **restricts modification**.

| **Use** | **Effect** |
| --- | --- |
| final variable | Cannot be changed after initialization |
| final method | Cannot be overridden in a subclass |
| final class | Cannot be inherited |

**9. Can a class extend multiple classes in Java?**

No, Java **does not support multiple inheritance** with classes to avoid **ambiguity issues**.

However, you can **achieve multiple inheritance using interfaces**:

**METHODS**

**Methods** in Java are blocks of code designed to perform specific tasks. They are essential for organizing code, making it reusable, and improving readability. A method is declared within a class and can be called multiple times from different parts of the program.

(OR)

A **method** in Java is a block of code that performs a specific task. It can take input (parameters), process data, and return a result.

**Method Declaration Syntax**

Java

modifier returnType methodName(parameterList) {  
 *// method body*  
 return value; *// if returnType is not void*  
}

* **modifier**: Defines the access type (e.g., public, private, protected) and other attributes (e.g., static, final).
* **returnType**: Specifies the data type of the value returned by the method (e.g., int, String, void).
* **methodName**: A unique identifier for the method, following naming conventions.
* **parameterList**: A comma-separated list of input parameters with their data types (optional).
* **method body**: Contains the statements that perform the method's task.
* **return value**: If the return type is not void, the method must return a value of the specified type.

**Call a Method**

* To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

**Types of Methods**

* **Pre-defined Methods**:

Built-in methods available in Java libraries (e.g., Math.sqrt(), String.length()).

* **User-defined Methods**:

Methods created by programmers to perform custom tasks.

* **Instance Methods**:

Methods that operate on specific objects of a class and require an instance of the class to be call the method. call instance methods using an object reference

* **Static Methods**:

static methods belong to the class itself and can be called without creating an instance, and can be called directly using the class name.

* **Abstract Methods**:

Methods declared without implementation in abstract classes, requiring subclasses to provide concrete implementations.

* **Factory Methods**:

Methods that return objects of a class, often used to create different types of objects based on input.

* **Bridge Methods**:

Synthetic methods generated by the compiler to handle type erasure in generics.

**What is method overloading in Java?**

**Method Overloading** allows multiple methods with the **same name** but **different parameters** in the same class.

**Example:**

java

CopyEdit

class MathOperations {

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

int add(int a, int b, int c) { return a + b + c; }

}

**Key Points:**

✔ Return type **can** be different  
✔ Parameters **must** be different  
❌ Method signature must be unique

**What is method overriding in Java?**

**Method Overriding** occurs when a subclass provides a **specific implementation** for a method defined in its superclass.

**Example:**

java

CopyEdit

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() { // Overriding the method

System.out.println("Dog barks");

}

}

**Key Points:**

✔ Same method name & parameters  
✔ @Override annotation is used  
✔ Subclass **cannot** override a final or static method

**Passing Parameters by Value in Java Methods**

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

**The this Keyword inside Java Methods**

This keyword is used to Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

* **Distinguish between instance variables and method parameters:** When a method parameter has the same name as an instance variable, this clarifies that you are referring to the instance variable.

**Can a method be declared as final? What happens if we try to override it?**

* Yes, a method can be declared as **final**, meaning it **cannot be overridden** in subclasses.

**What is a recursive method in Java?**

* A **recursive method** is a method that **calls itself**.

What is a constructor in Java? How is it different from a method?

What is the difference between a static method and an instance method?

**What Happens if We Change the Reference Inside a Method in Java?**

In **Java**, method arguments are always **passed by value**, including **object references**. This means that if we change the reference inside a method, it does **not** affect the original reference outside the method.

**Java - Variable Scopes**

**Scope of Java Local Variables**

All other variables that are not instance and class variables are treated as local variables, including the parameters in a method. The scope of a local variable is within the block in which it is declared, and the lifetime of a local variable is until the control leaves the block in which it is declared.

**Scope of Java Class (Static) Variables**

A variable that is declared inside a class, outside all the blocks, and is marked static is known as a class variable. The general scope of a class variable is throughout the class, and the lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

**Scope of Java Instance Variables**

A variable that is declared inside a class but outside all the methods and blocks is an instance variable. The general scope of an instance variable is throughout the class except in static methods. The lifetime of an instance variable is until the object stays in memory.

**Can a method be declared as final? What happens if we try to override it?**

Yes, a method can be declared as **final**, meaning it **cannot be overridden** in subclasses.

**What is a constructor in Java? How is it different from a method?**

A **constructor** is a special method used to initialize objects.

| **Feature** | **Constructor** | **Method** |
| --- | --- | --- |
| Name | Same as class | Can have any name |
| Return Type | No return type | Has a return type |
| Call | Called automatically during object creation | Called explicitly |

**What is a recursive method in Java?**

A **recursive method** is a method that **calls itself**.

**How to shut-down JVM manually?**

we can shut-down JVM manually by using System.exit(0)

Java - Data Types

**Java Primitive Data Types**

Primitive data types are predefined by the language and named by a keyword. There are eight primitive data types supported by Java. Below is the list of the primitive data types:

* [byte](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#byte)
* [short](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#short)
* [int](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#int)
* [long](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#long)
* [float](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#float)
* [double](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#double)
* [boolean](https://www.tutorialspoint.com/java/java_basic_datatypes.htm#boolean)

**Java Non-Primitive (Reference/Object) Data Types**

The non-primitive data types are not predefined. The reference data types are created using defined [constructors](https://www.tutorialspoint.com/java/java_constructors.htm) of the [classes](https://www.tutorialspoint.com/java/java_object_classes.htm). They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.

The following are the non-primitive (reference/object) data types −

* [**String**](https://www.tutorialspoint.com/java/lang/java_lang_string.htm): The string is a class in Java, and it represents the sequences of characters.
* [**Arrays**](https://www.tutorialspoint.com/java/java_arrays.htm): Arrays are created with the help of primitive data types and store multiple values of the same type.
* [**Classes**](https://www.tutorialspoint.com/java/java_object_classes.htm): The classes are the user-defined data types and consist of variables and methods.
* **Interfaces**: The interfaces are abstract types that are used to specify a set of methods.

The default value of any reference variable is null. A reference variable can be used to refer to any object of the declared type or any compatible type.

Wrapper classes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Day-18

Wrapper classes are used to convert primitive to object or vice-versa.

wrapper classes are immutable in nature

Java Guys provide wrapper classfor every primitives

Primitive

\*\*\*\*\*\*\*\*\*

byte

short

int

long

float

double

char

boolean

Wrapper

\*\*\*\*\*\*\*\*\*\*

Byte

Short

Integer

Long

Float

Double

Character

Boolean

**converting primitive 2 object is called as Boxing/Auto-Boxing**

**converting object 2 primitive is called as Un-Boxing/Auto-UnBoxing**

**Boxing**

\*\*\*\*\*\*

converting primitive 2 object by the developer is called as Boxing

Wrapper class contains constructor which takes primitive as i/p and

convert into corresponding

public class Driver {

@SuppressWarnings("removal")

public static void main(String[] args)

{

int n1=10;

int n2=20;

//Boxing

Integer obj1 = new Integer(n1);

Integer obj2 = new Integer(n2);

Integer sum=obj1+obj2;

System.out.println(sum);

}

}

**Auto-Boxing**

\*\*\*\*\*\*\*\*\*\*\*

Auto-Boxing introduced from java1.5 onwards

converting primitive 2 object by the java machine is called as Boxing

public class Driver {

@SuppressWarnings("removal")

public static void main(String[] args)

{

int n1=10;

int n2=20;

//Boxing

Integer obj1 = n1;

Integer obj2 = n2;

Integer sum=obj1+obj2;

System.out.println(sum);

}

}

**Un-Boxing**

\*\*\*\*\*\*\*\*\*

converting object 2 primitive by the developer is called as Un-Boxing

public class Driver {

@SuppressWarnings("removal")

public static void main(String[] args)

{

**Integer n1=10;**

**Integer n2=20;**

**int a = n1.intValue();**

**int b = n2.intValue();**

int sum=a+b;

System.out.println(sum);

}

}

**Auto-UnBoxing**

\*\*\*\*\*\*\*\*\*\*\*\*\*

converting object 2 primitive by the java machine is called as Auto Un

Boxing

public class Driver {

public static void main(String[] args)

{

**Integer n1=10;**

**Integer n2=20;**

**int a = n1;**

**int b = n2;**

int sum=a+b;

System.out.println(sum);

}

}

use-case

\*\*\*\*\*\*\*\*

**collections will support objects only and doesn't support primitives**

**primitive -->collections -->doesn't support**

**object-->collections -->support**

**primitive -->Wrapper classes-->objects-->collections -->support**

Ex

\*\*

public class Driver {

public static void main(String[] args)

{

List l=new ArrayList();

l.add(10);

l.add(20);

l.add(30);

}

}

Req: **How to convert String which contains numeric into integer**

format?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*

String s1="10"

String s2="90"

public class Driver {

public static void main(String[] args)

{

String s1="10";

String s2="90";

**int n1=Integer.parseInt(s1);**

**int n2=Integer.parseInt(s2);**

int sum=n1+n2;

System.out.println(sum);

}

}

what is difference between parseInt() and valueOf()?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

parseInt() will take String as i/p and convert into primitive where as

valueOf() will take String as i/p and convert into object format.

public class Driver {

public static void main(String[] args)

{

String s1="10";

String s2="90";

Integer n1=Integer.valueOf(s1);

Integer n2=Integer.valueOf(s2);

Integer sum=n1+n2;

System.out.println(sum);

}

}

**Constructor**

**Constructor** is used to intialize the instance variables

* class name and constructor name should be same
* constructor shouldn't return anything even void also as a developer no need to call
* constructor constructor get called automatically at the time of object creation by using **'new' keyword**

constructor is divided into 2 types

1. **default constructor**

The constructor without arguments is called as default constructor default constructor purpose is to initialize default values to instance variables

2.**parameterized constructor**

The constructor with arguments is called as parameterized constructor parameterized constructor is used to initialize the specific values to instance variables

**3.Copy Constructor (Manually Implemented)**

* Creates a copy of an existing object

**constructor chaining**

calling one constructor from another constructor inside the class is called as constructor chaining this()-->is used to call default constructor this(v1,v2)-->is used to call parameterized constructor constructor call should be first statement inside constructor definition.

**Can a constructor be private?**

Yes, a **constructor can be private**. This is used in **Singleton Design Pattern** to prevent multiple object creation.

✔Prevents external instantiation  
✔ Ensures **only one instance** of the class exists

**What happens if we don’t define a constructor?**

If a class does not define a constructor, **Java automatically provides a default constructor**.

✔ Default constructor **assigns default values** (0 for int, null for objects, false for boolean)

**Can a constructor call another constructor? (Constructor Chaining)**

Yes! A constructor can call another constructor in the same class using this().

this() is used for **constructor chaining** within the same class

**Can a constructor be overloaded?**

Yes, **constructor overloading** allows multiple constructors with **different parameters** in the same class.

✔ Constructor overloading helps **create objects flexibly**

**Can a constructor be static?**

❌ **No**, constructors cannot be **static** because:

* A static method belongs to the class, while constructors belong to objects.
* A constructor runs when an **object** is created, but static methods run without objects.

**Can we inherit constructors?**

No, constructors are **not inherited** in Java. However, a subclass can call the parent class constructor using super().

uper() is used to call the parent class constructor

**Summary of Java Constructor Concepts**

✔ **Constructors initialize objects** and have the **same name as the class**  
✔ Java provides a **default constructor** if none is defined  
✔ **Parameterized constructors** initialize objects with values  
✔ **Copy constructors** create a copy of an object  
✔ **Constructor chaining** is done using this() or super()  
✔ **Private constructors** prevent direct instantiation (used in Singleton)  
✔ **Constructors cannot be static, inherited, or return values**

**Strings**

* String is class in java

sequence of characters placed inside double quotes is called as **String**

"Narendra"

**we can create String in 2 ways**

* by using 'new' operator
* by using string literal

**by using 'new' operator**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

String s=new String("Narendra");

2 objects get created whenever we will use 'new' operator to create String object

first object blindly created in **heap area** without checking

second object stored in SCP

**SCP -->String constant pool**

**SCP is part of Heap area**

before creating second object in SCP

we will check whether object(Narendra) exist in SCP or not by using equals()

equals() return boolean value

suppose object exist in SCP then equals() returns true

suppose object doesn't exist in SCP then equals() returns false

suppose object doesn't exist in SCP then object copied from heap area to SCP by using intern()

but reference variable still will pointing to heap object

**by using string literal**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

String s="Narendra";

String literals are stored in SCP

SCP -->String constant pool

SCP is part of Heap area

verification is required before storing string literal in SCP by using equals()

we will check whether object(Narendra) exist in SCP or not by using equals()

equals() return boolean value

suppose object exist in SCP then equals() returns true

suppose object doesn't exist in SCP then equals() returns false

suppose object doesn't exist in SCP then Java Machine will create brand new object and place in SCP

suppose object exist in SCP then Java Machine will not create any object in SCP.

**object is divided into 2 types**

* mutable object
* immutable object

**immutable object**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

object state(data) doesn't get changed whenever we will perform operation on existing object.

PAN Number,Passport Number,Adhar Number

**String is example for immutable object.**

**mutable object**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

object state(data) get changed whenever we will perform operation on existing object

user-defined classes are example for mutable object

**StringBuilder and StringBuffer are examples for mutable object**

**StringBuffer vs StringBuilder**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**StringBuffer**

\*\*\*\*\*\*\*\*\*\*\*\*

StringBuffer is class in java

StringBuffer is mutable object

we can create StringBuffer object by using 'new' operator

StringBuffer object get stored in Heap area

Example

\*\*\*\*\*\*\*

StringBuffer sb=new StringBuffer("VJ");

StringBuffer mutable object proof

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public class Driver {

public static void main(String[] args)

{

StringBuffer sb=new StringBuffer("welcome to ");

sb.append("VJ");

System.out.println(sb);

}

}

**StringBuffer methods**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

append()

append() is used to merge 2 StringBuffer objects.

reverse()

reverse() is used to reverse String.

insert()

insert() is used to data at specific index

delete(startIndex,endIndex)

delete() is used to delete data between start & end index

deleteCharAt(index)

deleteCharAt() will delete corresponding character based on index

replace()

replace() is used to replace the content between start & ending indexes

**StringBuilder**

\*\*\*\*\*\*\*\*\*\*\*\*\*

StringBuilder is class in java

StringBuilder is mutable object

we can create StringBuilder object by using 'new' operator

StringBuilder object get stored in Heap area

Example

\*\*\*\*\*\*\*

StringBuilder sb=new StringBuilder("VJ");

what is difference between StringBuffer & StringBuilder?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

StringBuffer is synchronized where as

StringBuilder is not synchronized

StringBuffer is thread safe

where as StringBuilder is not thread safe

synchronization

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

suppose 10 threads perform operation on same object then we will get consistent results.

How to convert mutable object to immutable?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

define class as final so that final class can't participate in inheritance

declare class variables as final

final variables are initialized via constructor

final variables can't re-initialized once initialization done for final variables

we can write getters for final variables

we can't write setters for final variables

what is difference between length and length()?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

'length' is property in array to find out no of elements exist on given array

'length()' is method in String to find out number of characters exist in given String

why String is immutable?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 **Security** - Used in class loading, caching, and authentication.

 **Performance** - Saves memory using **String Pooling**.

 **Thread Safety** - No need for synchronization in multi-threaded environments.

 **HashCode caching** - Since String objects are used as keys in HashMaps.

How to convert user-defined class to immutable class?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

declare class as final so that final class can't participate in inheritance.

declare class variables as final

final variables must be initialized through constructor

we can write getters only for final variables

There is no option to design setters for final variables

**What is the difference between == and .equals() when comparing objects?**

* + == compares object references (memory addresses), while .equals() compares object content (values).

**How do you convert a String to a char[]?**

**Answer:**  
Using .toCharArray():

**How do you convert a char[] to a String?**

**Answer:**  
Using String constructor:

java

CopyEdit

char[] chars = {'J', 'a', 'v', 'a'};

String str = new String(chars);

**How do you split a String in Java?**

**Answer:**  
Using .split() method:

String str = "Java,Python,C++";

String[] languages = str.split(",");

**How do you check if a string contains a specific substring?**

**Answer:**  
Using .contains() method:

**Why do we need String Pool in Java?**

It is created to decrease the number of string objects created in the memory. Whenever a new string is created, JVM first checks the string pool. If it encounters the same string, then instead of creating a new string, it returns a reference existing string to the variable

**Stringpool**

Every string specified in the code as string Literal is stored in an area of memory called stringpool.

Stringpool is a special array for storing strings its purpose is to optimize string storage

**String intern() method**

The method *intern()* creates an exact copy of a *String* object in the heap memory and stores it in the *String* constant pool.

Note that, if another *String* with the same contents exists in the *String* constant pool, then a new object won’t be created and the new reference will point to the other *String.*

**Java String intern() method is used to manage memory by reducing the number of String objects created. It ensures that all same strings share the same memory.**

**For example, creating a string “hello” 10 times using intern() method would ensure that there will be only one instance of “Hello” in the memory and all the 10 references point to the same instance.**

**How intern() Works**

When you use intern() method on a String, it does following in order:

1. It checks if an identical string is already present in the intern pool.
2. If It finds an identical string in the pool, instead of creating new string, it returns the reference from the pool.
3. If it doesn’t find an identical string, it adds the string to the pool and returns the reference to this new entry(object).
4. public class Example{
5. public static void main(String args[]){
6. String str1 = new String("BeginnersBook");
7. String str2 = new String("BeginnersBook");
8. // Output: false as these strings are created
9. // without calling intern() method
10. System.out.println(str1 == str2);
11. String str3 = str1.intern();
12. String str4 = str2.intern();
13. // Output: true as these strings are created
14. // after calling intern() method on identical strings
15. System.out.println(str3 == str4);
16. }
17. }

**StringTokenizer**

The StringTokenizer class in Java is often used when we need to split a string using multiple delimiters, whereas the split() method of the String class is more suited for cases where we have a single delimiter or need regex-based splitting.

In Java, StringTokenizer is a class used to break a string into tokens. These tokens are smaller parts of the string, separated by delimiters. The StringTokenizer class is found in the java.util package. While it's a legacy class and its use is discouraged in new code, it remains available for compatibility reasons. It's recommended to use the split method of the String class or the java.util.regex package instead.

Here's how StringTokenizer works:

* **Constructors:**
  + StringTokenizer(String str): Constructs a string tokenizer for the specified string, using default delimiters (space, tab, newline, carriage return).
  + StringTokenizer(String str, String delim): Constructs a string tokenizer for the specified string, using the specified delimiter.
  + StringTokenizer(String str, String delim, boolean returnDelims): Constructs a string tokenizer for the specified string, using the specified delimiter, and optionally returning delimiters as tokens.
* **Methods:**
  + hasMoreTokens(): Checks if there are more tokens available.
  + nextToken(): Returns the next token.
  + nextToken(String delim): Returns the next token, using the specified delimiter.
  + countTokens(): Returns the number of remaining tokens.

**Below is an example:**

Java

import java.util.StringTokenizer;  
  
public class StringTokenizerExample {  
 public static void main(String[] args) {  
 String str = "apple,banana,orange";  
 StringTokenizer tokenizer = new StringTokenizer(str, ",");  
  
 while (tokenizer.hasMoreTokens()) {  
 String token = tokenizer.nextToken();  
 System.out.println(token);  
 }  
 }  
}

**Java String format() method**

**Java String format()** :- Stringformat method helps to create a string with specific content and formatting.

It has a static format() method that lets you specify a pattern for assembling a string with data.

**String name =String.format(pattern , parameter);**

**Instead of joining parts of a string with “+” symbol we can neatly format entire string in one go.**

**For example if we want to print a product name**(string) **and and its price**(integer value) **in only one line it is used**

**Example:-**

String result = String.format("Product: %s, Price: %d", product, price);

System.out.println(result);

**Common Format Specifiers**

* %s – String
* %d – Integer
* %f – Floating point number
* %t – Date/Time
* %x – Hexadecimal

**ARRAYS**

An **array** in Java is a data structure that store multiple values of the same data type. They are fixed in size once declared and can hold primitive types or object references.

**Declaration and Initialization**

Java

*// Declaration*  
int[] iArray;  
  
*// Initialization with a size*  
Array = new int[10];  
  
*// Declaration and initialization in one line*  
int[] Array2 = new int[10];  
  
*// Initialize with values directly*  
int[] Array3 = {1, 2, 3, 4, 5};

**Accessing Elements**

Array elements are accessed using their index, which starts from 0.

Java

int[] numbers = {10, 20, 30};  
System.out.println(numbers[0]); *// Output: 10*  
System.out.println(numbers[2]); // Output: 30

Types of Arrays

* **Single-dimensional arrays:** Store elements in a single row.
* **Multi-dimensional arrays:** Store elements in rows and columns (like a matrix).

Java

*// 2D array*  
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
System.out.println(matrix[1][2]); // Output: 6

**Common Operations**

* **Arrays.sort():** Sorts the array in ascending order.
* **Arrays.binarySearch():** Searches for a specific value in a sorted array.
* **Arrays.equals():** Checks if two arrays are equal.
* **Arrays.fill():** Fills an array with a specific value.
* **Arrays.copyOf():** Copies an array to a new array.

**Important Notes**

* Arrays in Java are objects.
* Array size is fixed after declaration.
* The base index of Java arrays is always 0.
* ArrayIndexOutOfBoundsException is thrown if you try to access an index outside the array's bounds.

**Finding the Length of an Array**

The length of an array can be accessed using **array.length** (without parentheses).

java

CopyEdit

int[] numbers = {10, 20, 30, 40, 50};

System.out.println("Length: " + numbers.length); // Output: 5

**Looping Through Arrays**

**Using for Loop**

int[] numbers = {10, 20, 30, 40, 50};

for (int i = 0; i < numbers.length; i++) {

System.out.println(numbers[i]);

}

**Using for-each Loop (Enhanced for Loop)**

for (int num : numbers) {

System.out.println(num);

}

This loop **automatically iterates** over each element.

**Arrays.toString() Method in Java**

The Arrays.toString() method is a convenient way to convert an array into a readable string representation. It is part of the java.util.Arrays class and is used to print or display arrays easily instead of looping through elements manually.

import java.util.Arrays;

public class ToStringExample {

public static void main(String[] args) {

int[] numbers = {10, 20, 30, 40, 50};

System.out.println(Arrays.toString(numbers));

} }

Output:

[10, 20, 30, 40, 50]

**Without Arrays.toString(), printing an array directly (System.out.println(numbers)) would return something like [I@15db9742, which is the memory reference, not the actual values.**

**Arrays.equals() Method in Java**

The Arrays.equals() method is used to **compare two arrays** and check if they have the **same elements in the same order**.

Example:

import java.util.Arrays;

public class EqualsExample {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3};

int[] arr2 = {1, 2, 3};

int[] arr3 = {1, 2, 4};

System.out.println(Arrays.equals(arr1, arr2)); // Output: true

System.out.println(Arrays.equals(arr1, arr3)); // Output: false

}

}