**EXCEPTION**

Checked exception : - forced developer to handle it. (by throws as signature or try block)

When condition is recoverable. Like : wrong path for file

By – invalid input

**Real-World Scenario:**

Suppose you are building an application that reads user data from a file. If the file does not exist, this is a recoverable problem (e.g., you could prompt the user to provide a correct file path). A custom checked exception forces the developer to handle this scenario.

**Why Checked Exceptions Work Here:**

The code explicitly **forces the developer to address the problem** of a missing file. It makes the application more robust by handling predictable, recoverable issues

Unchecked exception : optional handling

Condition in unrecoverable

By – logical error, wrong argument

**Real-World Scenario:**

Suppose you have a banking application, and you want to throw an exception when a user attempts to withdraw more money than their account balance. This is a programming mistake that should not be recoverable during runtime.

**Why Unchecked Exceptions Work Here:**

* This exception does not require mandatory handling because the scenario reflects a **logical error**. Fixing this would involve correcting the input or business logic, not recovering from the error during runtime.

**Summary of Key Differences:**

| **Type** | **Prompts User for Input** | **When to Use** | **Handling** |
| --- | --- | --- | --- |
| **Checked Exception** | Yes, because the situation is recoverable. Example: Prompt for a correct file path. | When you know the exception can be recovered from, such as when a file is missing or user input is invalid. | Developer is forced to handle the exception. |
| **Unchecked Exception** | Yes, because the situation may be caused by logical errors. Example: Prompt for the correct amount of money. | When the exception reflects a logical or programming error that you want to correct, not necessarily recover from. | Handling is optional; you can still catch it and prompt the user. |

**NaN, undefined and null in java**

In **Java**, NaN, null, and undefined are handled **very differently** compared to JavaScript. Here's a clear explanation:

**✅ 1. null in Java**

**Meaning**: Represents the absence of a value for **object references**.

java

CopyEdit

String str = null;

System.out.println(str); // Output: null

**Key Points:**

* Only applies to **objects**, not primitive types (int, double, etc.).
* Calling a method or accessing a member on a null object will throw a NullPointerException.

java

CopyEdit

String str = null;

System.out.println(str.length()); // ❌ NullPointerException

**✅ 2. NaN in Java**

**Meaning**: Stands for "Not-a-Number", used for invalid **floating-point operations** like 0.0/0.0.

java

CopyEdit

double x = 0.0 / 0.0;

System.out.println(x); // Output: NaN

**Key Points:**

* Only applies to float and double.
* Type is double or float.
* NaN != NaN is true (just like in JavaScript).
* Use Double.isNaN(x) to check.

java

CopyEdit

System.out.println(Double.isNaN(x)); // true

**⚠️ 3. undefined in Java**

**Meaning**: **Does not exist** in Java.

* Java is a statically typed language.
* All variables must be declared and initialized (or they get a default value like 0, false, or null depending on the type).
* If you try to use a variable without initializing it, you'll get a **compile-time error**, not undefined.

java

CopyEdit

int x;

System.out.println(x); // ❌ Compilation error: variable x might not have been initialized

**🧠 Summary Table**

| **Concept** | **Java Equivalent?** | **Applies To** | **Notes** |
| --- | --- | --- | --- |
| null | ✅ Yes | Objects only | No value assigned to a reference |
| NaN | ✅ Yes | double / float | Result of invalid math |
| undefined | ❌ Not in Java | Not applicable | Java variables must be declared & initialized |

**Signed and Unsigned Types — Java vs C**

* In java all datatypes are signed to byte having range -128 to 127
* In C datatypes are as both signed and unsigned so
  + Signed range : -128 to 127
  + Unsigned range : 0 to 255

**1. Java Integer Types**

* All integer types (byte, short, int, long) in Java are **signed** by default.
* Java **does not provide unsigned versions** of these types (except for char).
* The char type in Java is **unsigned 16-bit**, representing Unicode characters from 0 to 65,535.
* **Ranges**:

| **Type** | **Bits** | **Range** |
| --- | --- | --- |
| byte | 8 | –128 to 127 |
| short | 16 | –32,768 to 32,767 |
| int | 32 | –2,147,483,648 to 2,147,483,647 |
| long | 64 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |

* **Unsigned values in Java**:
  + No native unsigned byte, int, or long.
  + You can **simulate unsigned behavior** by using larger types or utility methods:
    - Example: Byte.toUnsignedInt(byteValue) converts signed byte to unsigned int.
  + For example, storing 128 in a byte:

java

CopyEdit

byte b = (byte)128; // Overflow; actual value is -128

int unsignedValue = Byte.toUnsignedInt(b); // unsignedValue = 128

**2. C Integer Types**

* C supports both **signed** and **unsigned** types explicitly.
* Types can be declared as:
  + signed int, unsigned int
  + signed char, unsigned char
  + signed short, unsigned short
  + etc.

**One Complimant using ~**

**Yes, in Java, the ~ operator does give the one's complement of a number.**

**🔁 What ~ does:**

The ~ (bitwise NOT) operator **flips all bits** of a number:

* 0 becomes 1
* 1 becomes 0

This is exactly what **one’s complement** means.

**✅ Example in Java:**

int x = 5; // Binary: 00000000 00000000 00000000 00000101

int result = ~x; // 11111111 11111111 11111111 11111010 (in two's comp = -6)

System.out.println(result); // Output: -6

**Important Note:**

Java uses **two’s complement** to store integers internally. So:

* ~x gives the **one’s complement**
* But the **value printed** is interpreted in **two’s complement form**

So ~5 gives binary ...11111010, which equals -6 in decimal.

**Step-by-Step Breakdown:**

**1. Binary of 5:**

5 in binary (32-bit):

00000000 00000000 00000000 00000101

**2. Apply ~ (bitwise NOT → one’s complement):**

Flip all bits (0 becomes 1, 1 becomes 0):

~5:

11111111 11111111 11111111 11111010

This is the binary result stored in result.

**3. How Java interprets this binary number:**

Java uses **two’s complement** representation for signed integers.

So when you see:

11111111 11111111 11111111 11111010

This is a **negative number** in two’s complement. Let’s convert it to decimal:

**✅ Convert Two's Complement to Decimal:**

To find the decimal of a two’s complement negative binary:

1. Invert the bits (get one’s complement):

00000000 00000000 00000000 00000101

1. Add 1:

+1 = 00000000 00000000 00000000 00000110 (which is 6)

1. Final result: -6

**✅ Therefore:**

int x = 5;

int result = ~x; // flips bits = -6

**Q2. What are wrapper classes?**

[Wrapper classes in Java](https://www.naukri.com/code360/library/wrapper-class-in-java) are classes of primitive data types. They are used to create objects of primitive data types and convert them back into primitive data types. They are needed when an object needs to be made if there is a need to change the arguments passed into a method.

**Q5. Why is the Java platform independent?**

Java is platform-independent because the Java Runtime Environment uses a JVM to create an executable file. The Java code, when compiled, is first converted into a bytecode. The bytecode is platform-independent, and the JVM can convert the bytecode into the machine code of a specific machine. This makes Java platform independent.

**JavaTpoint**

**Why java is robust?**

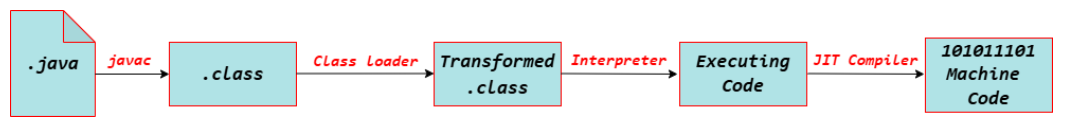
* Strong memory management
* Deficiency of pointer that prevents the security issue
* Write one, runs everywhere
* Garbage collector, that enhance the memory utilization.
* Exception handling and type checking

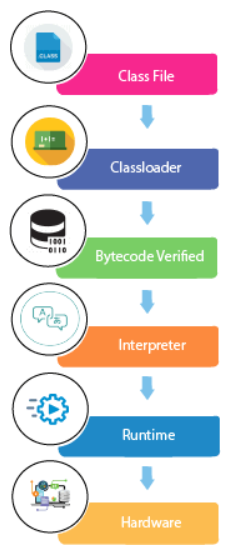
**Architecture-Neutral**

Java is architecture-neutral because it lacks implementation-dependent features; for instance, the size of primitive types is fixed.

In C programming, the int data type occupies 2 bytes of memory in a 32-bit architecture and 4 bytes in a 64-bit architecture. However, it occupies 4 bytes of memory in both 32-bit and 64-bit architectures in Java.

**Process:**



 **Classloader:** It is the subsystem of JVM that is used to load class files.

**Bytecode Verifier:** Checks the code fragments for illegal code that can violate access rights to objects.

**Interpreter:** Read bytecode stream then execute the instructions.

**Q.** **Can you save a Java source file by another name than the class name?**

Yes, if the class is not public. It is explained in the figure given below:



To compile : javac Hard.java

To execute : java Simple

Observe that, we have compiled the code with file name but running the program with class name. Therefore, we can save a Java program other than class name.

**Note : 1.** Javac coverts the source code into byte code by making .class file.

**2.** jvm interprets the byte code and execute.

**JVM (Java Virtual Machine) Architecture**-

* JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which Java bytecode can be executed.
* JVMs are available for many hardware and software platforms (i.e., JVM is platform-dependent).

**What it does?**

The JVM performs following operations:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

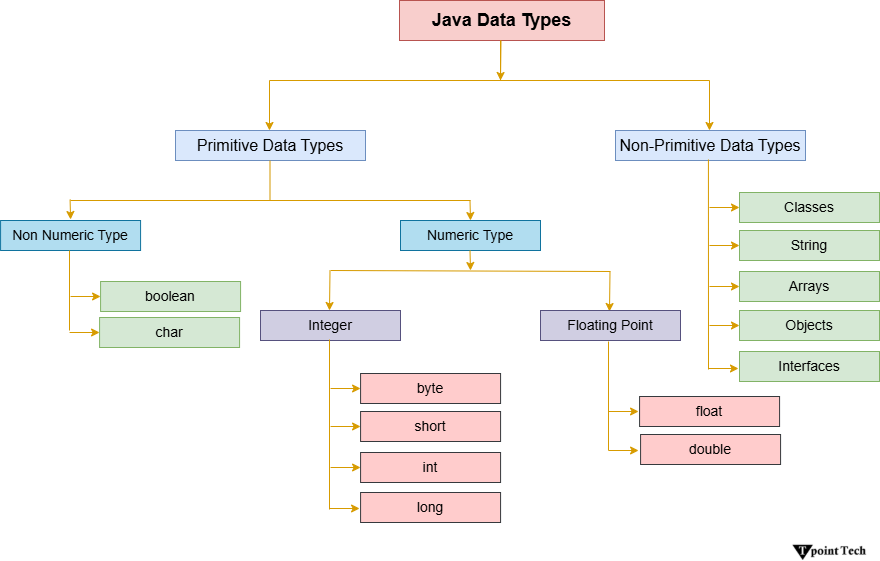
JVM Provides Definitions for The:

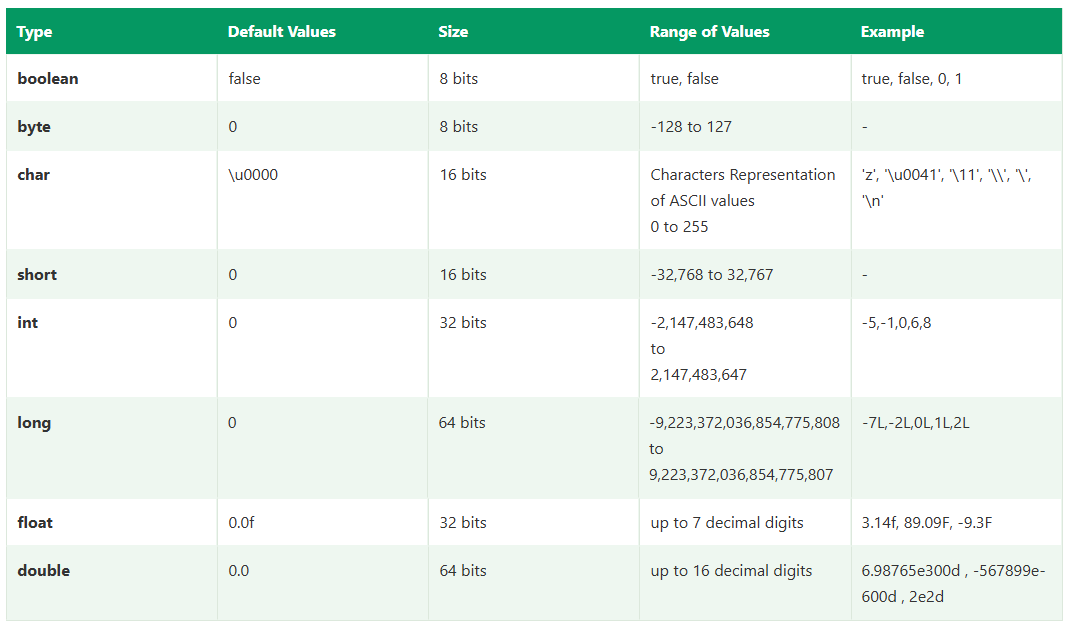
* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.

**Types of variable**

| **Feature** | **Static Variable** | **Instance Variable** | **Local Variable** |
| --- | --- | --- | --- |
| **Scope** | Class level | Object level | Method/block level |
| **Memory** | Method Area | Heap | Stack |
| **Lifetime** | Entire program | Object lifetime | Method execution time |
| **Access** | Class name or object | Object only | Inside method/block |

**DataTypes**





**Non-Primitive Datatypes:**

1. Class
2. Interface
3. String
4. Arrays
5. **Enum** : Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values.

**Syntax** :

enum Grade {

  FIRST,

  SECOND,

  THIRD

}

**Unicode System in Java**

Computer systems internally store data in binary representation. A character is stored using a combination of 0's and 1's. The process is called encoding. A character encoding scheme is important because it helps to represent the same information on multiple types of devices.

**Types of Encoding**

Following are the different types of encoding used before the Unicode system.

* ASCII (American Standard Code for Information Interchange): used for the United States
* ISO 8859-1 used for the Western European Languages
* KOI-8 used for Russian
* GB18030 and BIG-5 used for Chinese and so on.
* Base64 used for binary to text encoding

**Q.** Why does Java use Unicode System?

There were a few limitations to the encoding techniques used before the Unicode system.

1. In every language, different letters are present and the code assigned to every letter is also different which means multiple languages have multiple codes for various letters.
2. Certain languages have many character sets, the code assigned to each character may vary in terms of length. For example, some character can be encoded with a single byte, other might require two or more bytes.

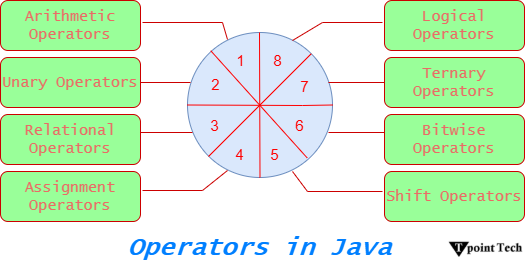
These problems led to finding a better solution for character encoding that is Unicode System.

**Q**. What is Unicode System?

* Unicode system is an international character encoding technique that can represent most of the languages around the world.
* Unicode System is established by Unicode Consortium.
* Hexadecimal values are used to represent Unicode characters.
* There are multiple Unicode Transformation Formats:
  1. UTF-8: It represents 8-bits (1 byte) long character encoding.
  2. UTF-16: It represents 16-bits (2 bytes) long character encoding
  3. UTF-32: It represents 32-bits (4 bytes) long character encoding.
* To access a Unicode character the format starts with an escape sequence \u followed by 4 digits hexadecimal value.
* A Unicode character has a range of possible values starting from \u0000 to \uFFFF.
* Some of the Unicode characters are  
  \u00A9 represent the copyright symbol - ©  
  \u0394 represent the capital Greek letter delta - Δ  
  \u0022 represent a double quote - "

NOTE : java uses the UTF-16 encoding for characters only.

**Java Operators:**



**Keywords:**

There are **50 official keywords**.

**Java Keywords Table**

| **Category** | **Keyword** | **Description** |
| --- | --- | --- |
| **Data Type** | boolean | Declares a boolean variable (true or false). |
|  | byte | 8-bit signed integer. |
|  | char | 16-bit Unicode character. |
|  | double | 64-bit floating-point number. |
|  | float | 32-bit floating-point number. |
|  | int | 32-bit signed integer. |
|  | long | 64-bit signed integer. |
|  | short | 16-bit signed integer. |
| **Control Flow** | break | Breaks loop or switch. |
|  | case | Defines a block in switch. |
|  | continue | Skips to next iteration of loop. |
|  | default | Defines default block in switch. |
|  | do | Starts a do-while loop. |
|  | else | Defines alternative block in if-else. |
|  | for | Starts a for loop. |
|  | if | Starts an if condition. |
|  | return | Returns from a method. |
|  | switch | Starts a switch statement. |
|  | while | Starts a while loop. |
| **Modifiers** | abstract | Declares abstract class or method. |
|  | final | Constant value / prevent inheritance / method override. |
|  | native | Indicates method is written in native code (JNI). |
|  | private | Access modifier (class-level, same class only). |
|  | protected | Access modifier (package + subclass). |
|  | public | Access modifier (accessible everywhere). |
|  | static | Class-level method or variable. |
|  | strictfp | Ensures consistent floating-point calculations. |
|  | synchronized | Thread-safe block or method. |
|  | transient | Skips field during serialization. |
|  | volatile | Variable value can be changed by multiple threads. |
| **Class Related** | class | Declares a class. |
|  | extends | Indicates class inheritance. |
|  | super | Refers to parent class object. |
|  | implements | Implements an interface. |
|  | interface | Declares an interface. |
|  | new | Creates new objects. |
|  | this | Refers to current object. |
| **Exception** | try | Starts exception handling block. |
|  | catch | Handles exceptions. |
|  | finally | Always executes after try-catch. |
|  | throw | Throws an exception. |
|  | throws | Declares exceptions in method signature. |
| **Package** | import | Imports packages/classes. |
|  | package | Defines a package. |
| **Others** | assert | Debugging check for conditions. |
|  | enum | Defines a set of constants. |
|  | instanceof | Tests if object is instance of class. |
|  | void | Specifies no return type. |
|  | null\* | Represents null reference (literal, not a keyword). |

**Note :**

**1.** true, false and null are not keywords because they cannot used as the identifiers but used as literals means as value to assign.

**2.** why goto and const are keyword but not used anymore?

- goto → **reserved but unused** (to avoid unstructured jumps).

- const → **reserved but replaced by final**.

**Switch Case:**

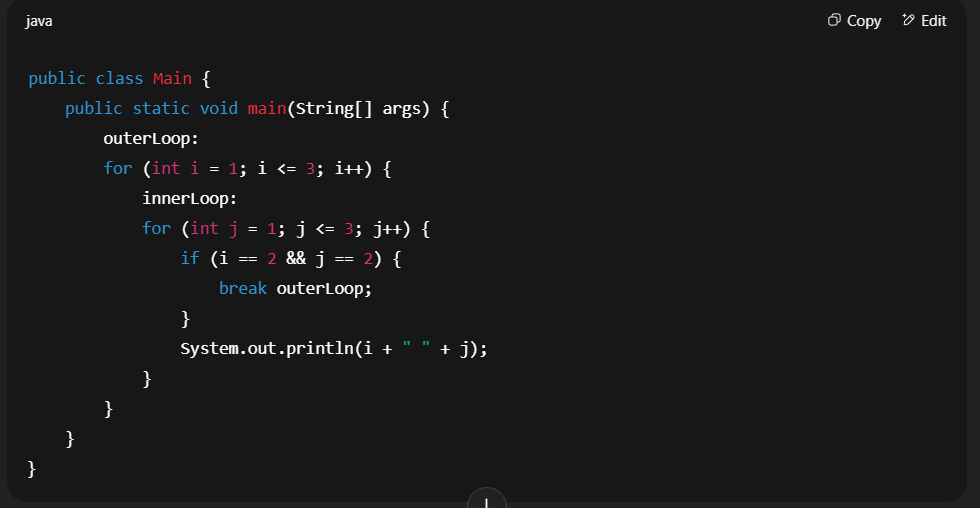
* Fall-through behavior allows the execution of multiple case blocks sequentially until a break statement is encountered.
* Switch statements with String objects are supported in Java starting from version 7.

**For loop:**

There are the following three types of for loops in Java.

* Simple for Loop
* For-each or Enhanced for Loop
* Labelled for Loop

**Example**: Labelled loop



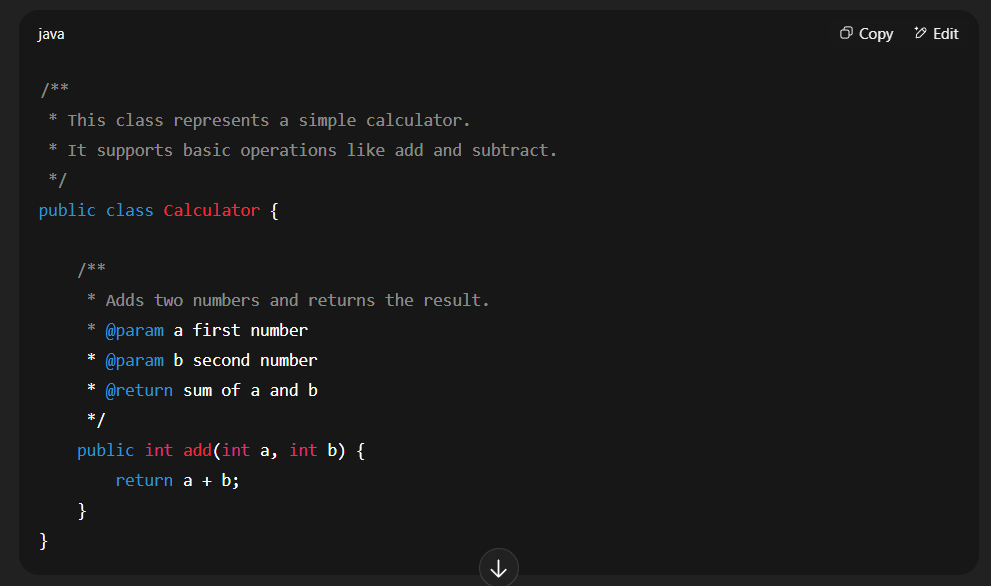
**Working :** when the i==2 and j==2 the inner loop will break for i=2 and then I will increment to I = 3 then again the innerloop j will run.

--if we use the break outerLoop instead of the break innerLoop then outer loop will end and the program will get over.

--Label is used to indicate that from where we have to break the code or have to skip the execution using the continue(as continue outerLoop)

**Java Comments:**

| **Type** | **Syntax** | **Use Case** |
| --- | --- | --- |
| Single-line | // | Small comments or notes |
| Multi-line | /\* ... \*/ | Block explanation or code disabling |
| Documentation (Javadoc) | /\*\* ... \*/ | JavaDoc generation for classes/methods |



OOPS

Characteristics of an Object

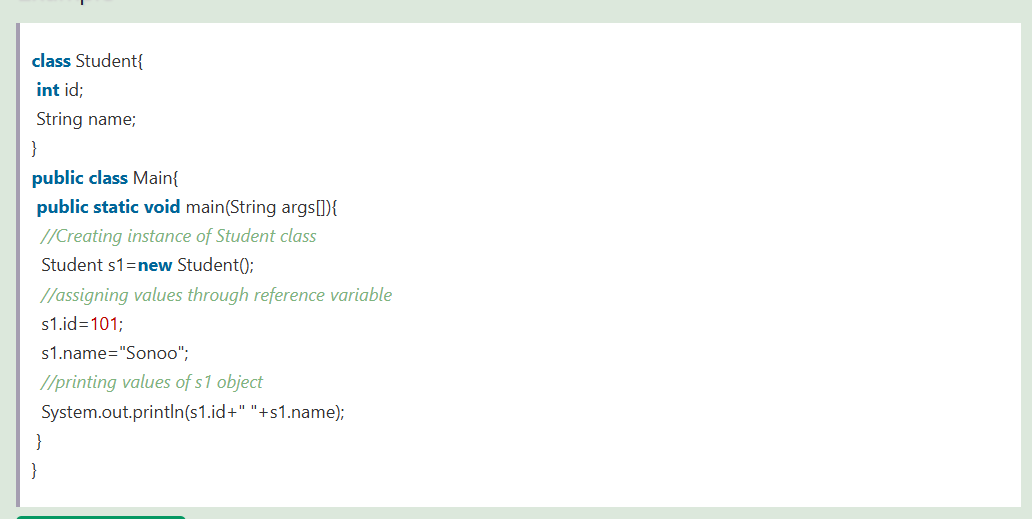
* **State:** It represents the data (value) of an object.
* **Behavior:** It represents the behavior (functionality) of an object, such as deposit, withdraw, etc.
* **Identity:** An object's identity is typically implemented via a unique ID. The ID's value is not visible to the external user; however, it is internally used by the JVM to identify each object uniquely.

**Initializing an Object in Java**

There are the following three ways to initialize an object in Java:

1. By Reference Variable
2. By Method
3. By Constructor

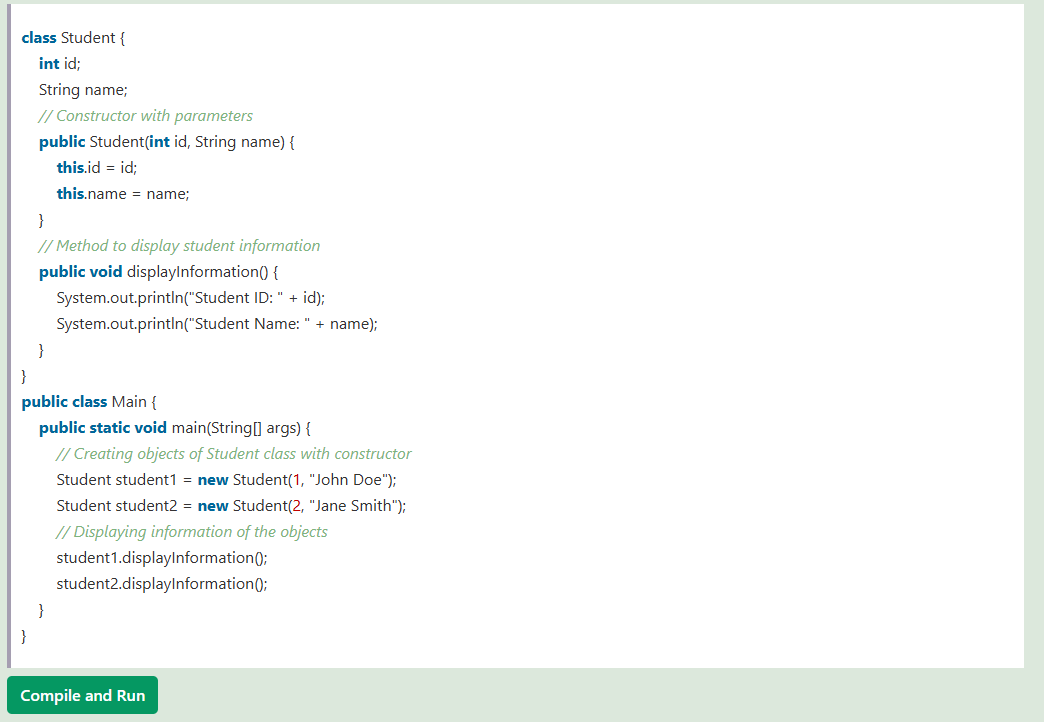
1. By Reference variable



2. By method



3. Through constructor



**Constructor chaining**: To call the super class constructor using the class constructor.

**🧭 Execution Order**

| **Step** | **Component** | **When it runs** |
| --- | --- | --- |
| 1️⃣ | **Static block(s)** | **Once when the class is loaded** |
| 2️⃣ | **Non-static (instance) block(s)** | Every time an object is created |
| 3️⃣ | **Constructor** | Immediately after non-static block(s) |

**🔄 Summary:**

* **Static blocks** → Run **once**, when the class is first loaded by the JVM (before any object is created).
* **Instance (non-static) blocks** → Run **every time** before the constructor, when an object is created.
* **Constructor** → Runs **every time** after instance block, when an object is created.

**This**

This keyword in a constructor is used to invoke another constructor of the same class, enabling constructor chaining.

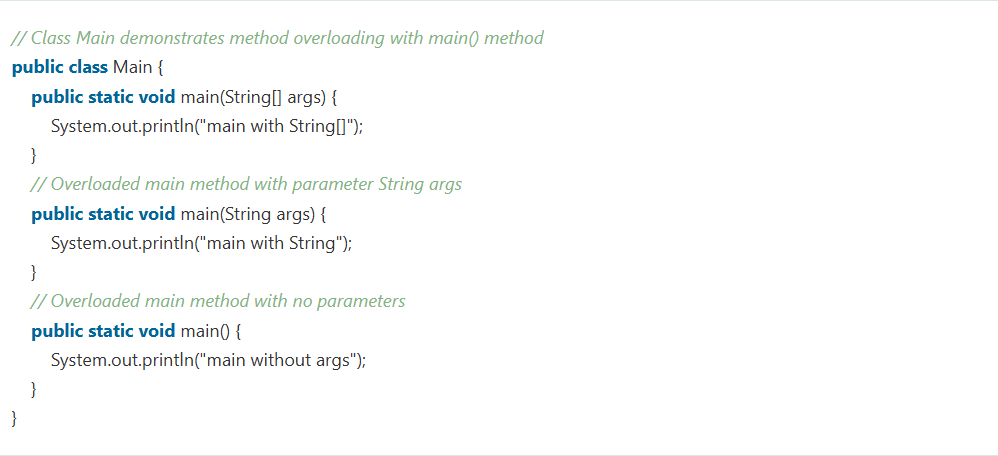


**Method Overlaoding**

1. By changing number of arguments
2. By changing the data type

**Q. Can we overload Java main() method?**

- NO because the jvm has the specific desig structure of the main as public static void main and if we overlaod the main methods would not be considered as the entry point will be considered as the normal method.



NOTE: static methods can be overloaded.

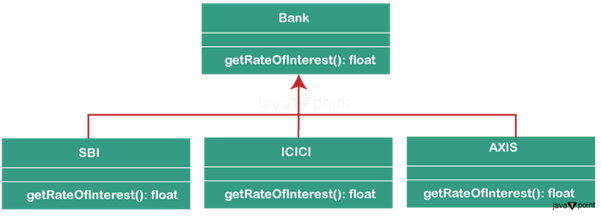
**Q) Why Method Overloading is not possible by changing the return type of method only?**

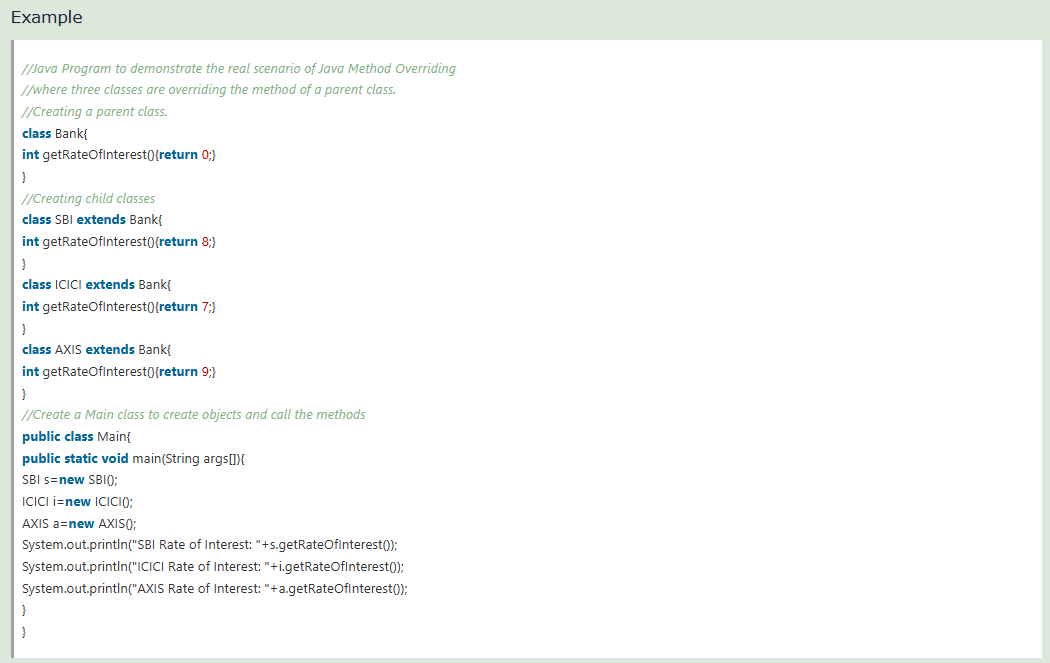
Method overloading in Java is based on the method signature, which includes the method name and parameter list. The return type alone is not sufficient to distinguish between overloaded methods because Java does not consider the return type when resolving method calls. If two methods have the same name and parameter list but different return types, the compiler cannot determine which method to call based solely on the return type.

**Method Overriding**

A Real World Example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

****

****

**Q. Why can we not override static method?**

Because static methods in Java are linked to the class itself rather than any specific instance of the class, we are unable to override them. Java's dynamic dispatch mechanism, which determines the method to be called at runtime depending on the object's actual type, forms the foundation for method overriding.

**Q. Can we override Java main() method?**

No, because the Java main() method is designated as static, we are unable to override it. The main function in Java is declared as public static void main(String[] args) and acts as the program's starting point.

**Super keyword**

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

**Final:**

**Q) Can we inherit final method?**

**Ans)** Yes, the final method can be inherited

**Q) What is a blank or uninitialized final variable?**

**Ans)** A final variable that is not initialized at the time of declaration is known as a blank final variable.

**Q) Can we initialize a blank final variable?**

**Ans)** Yes, but only in the constructor.

**Q) What is static blank final variable?**

**Ans)** A static final variable that is not initialized at the time of declaration is known as a static blank final variable. It can be initialized only in a static block.

**Q) Can we declare a constructor final?**

**Ans)** No, because a constructor never inherited.

**Q. In which scenario can a final variable be left uninitialized in Java?**

 A final variable declared at the class level (instance variable) can be left uninitialized at the point of declaration, but it must be initialized in the constructor of the class. This ensures that each instance of the class can assign a unique value to the final variable.

**Polymorphism**

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.

**Q. In the context of runtime polymorphism, what is dynamic method dispatch?**

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime rather than compile-time.

**Q. Can runtime polymorphism be achieved by data members in Java?**

No, it is only possible by member methods

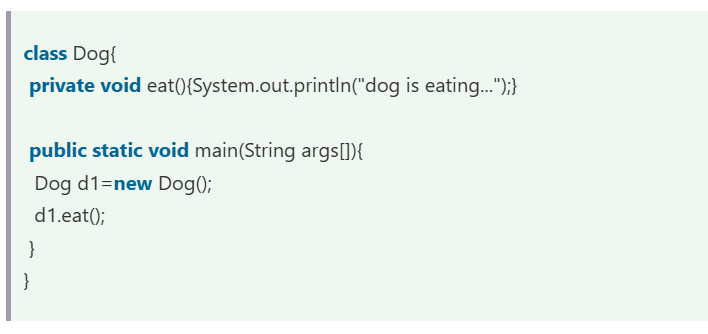
**Static and Dynamic Binding in Java**

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. 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.

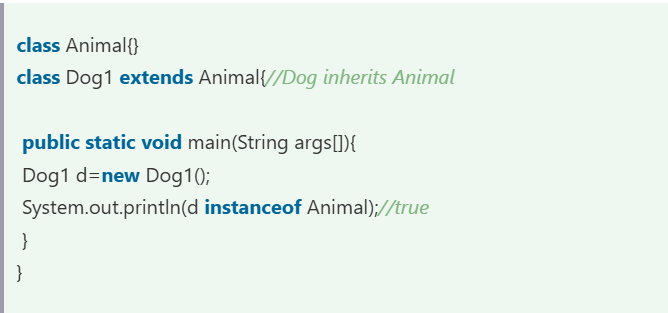


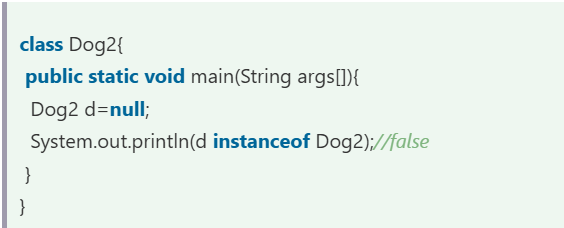
**Dymanic Binding:** When type of the object is determined at run-time, it is known as dynamic binding.



**Java instanceof Keyword**

Simple Example of Java instanceof





**Abstraction**

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

* Ways to achieve Abstraction
* There are two ways to achieve abstraction in Java:
* Using Abstract Class (0 to 100%)
* Using Interface (100%)

**Abstract Class:**

abstract classes play an important role in defining the structure of classes and their behavior in the hierarchy. They provide a blueprint for other teams to follow, and some methods remain undefined.

A class that is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Points to Remember

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

Note : constructor can be created in the abstract class but not in interface.

**Interface**

An **interface in Java** is a blueprint of a class.

The interface in Java is *a* mechanism to achieve [abstraction](https://www.tpointtech.com/abstract-class-in-java)

**Members and Methods in Interfaces (by default):**

**✅ 1. Variables (Fields)**

All variables declared in an interface are **implicitly**:

public static final var\_name

That means:

* public: accessible from anywhere
* static: belong to the interface, not to instances
* final: constant, value can't be changed

✅ So you **must initialize** them when you declare.

interface MyInterface {

int VALUE = 10; // same as: public static final int VALUE = 10;

}

Note: Initialization at time of decleration because final can be initialized in the constructor and interface have no constructor.

**✅ 2. Methods**

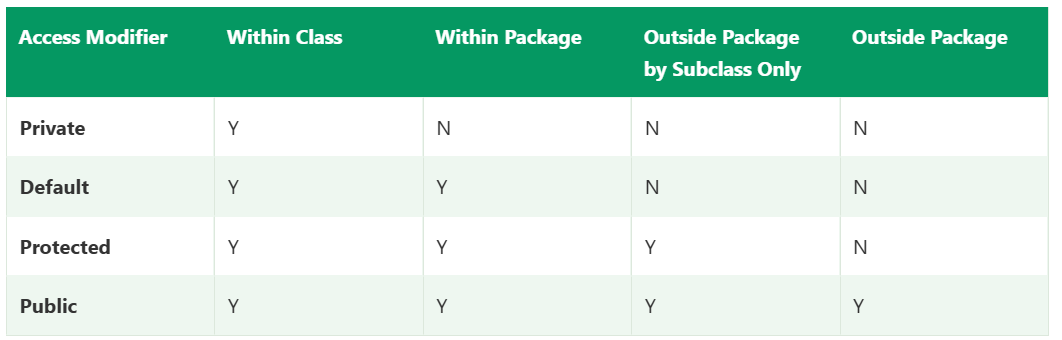
Java interface methods can be of different types:

| **Method Type** | **Default Modifier** | **Java Version** |
| --- | --- | --- |
| Abstract method | public abstract | Java 1.0 |
| Default method | public default | Java 8 |
| Static method | public static | Java 8 |
| Private method | private | Java 9 |

**Diff between Abstract class and interface:**

1. Abstract class have concrete methods cannot declear concrete method as default but interface having the default methods.

**Accessibility of Java Modifiers**



**Encapsulation:**

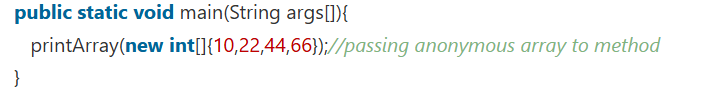
**Q. How does encapsulation promote the concept of data hiding in Java?**

By providing public methods for data access while keeping the data members private.

**Anonymous Array in Java**

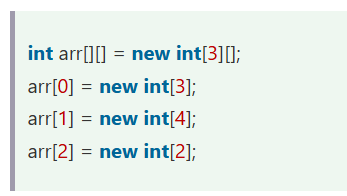
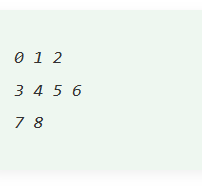
Java's anonymous arrays eliminate the requirement for separate declarations of array variables by enabling developers to build and initialize arrays directly within method calls or other expressions. When working with temporary arrays that are just needed for a single job and don't need a persistent reference within the program, this is quite helpful.

Example:

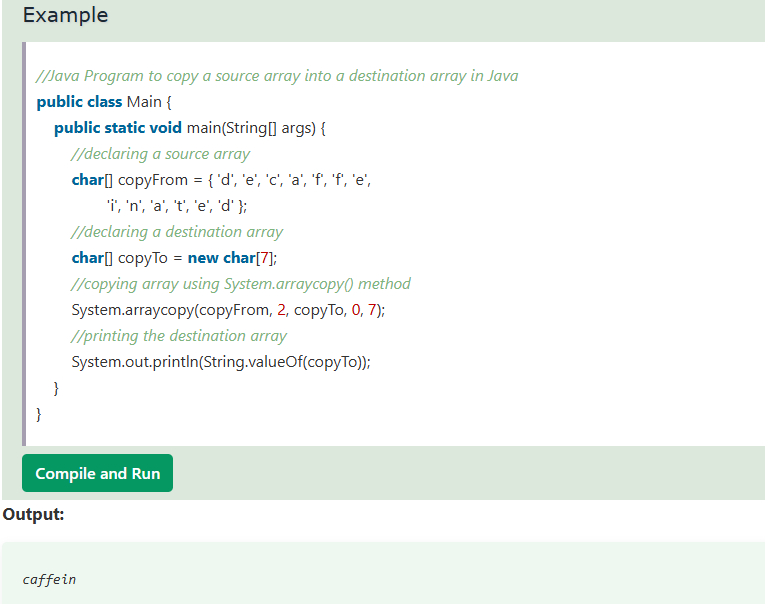


**Jagged Arrays in Java**

In Java, a jagged array is an array of arrays where each row of the array can have a different number of columns. This contrasts with a regular two-dimensional array, where each row has the same number of columns.



**Copying a Java Array**



**Explanation**

The Java program initializes two character arrays, copyFrom and copyTo, with predefined values. It then utilizes the System.arraycopy() method to copy a portion of the copyFrom array into the copyTo array, starting from index 2 and copying 7 elements. Finally, it prints the contents of the copyTo array, resulting in the output "caffein".

**Cloning an Array in Java**

In Java, arrays implement the Cloneable interface, allowing us to create clones of arrays. If we create a clone of a single-dimensional array, it creates a 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 a shallow copy of the Java array, which means it copies the references.

**Q. What does the expression arr.length return in Java?**

Maximum capacity of the array

**📚 java.util.Arrays Class — Common Methods**

| **Method** | **Description** |
| --- | --- |
| Arrays.toString(array) | Returns a string representation of the array. |
| Arrays.sort(array) | Sorts the array in ascending order. |
| Arrays.binarySearch(array, value) | Searches for a value using binary search (must be sorted). |
| Arrays.equals(arr1, arr2) | Compares two arrays for equality (element by element). |
| Arrays.copyOf(array, newLength) | Copies array to a new array of specified length. |
| Arrays.copyOfRange(array, from, to) | Copies a specified range. |
| Arrays.fill(array, value) | Fills the entire array with the given value. |
| Arrays.setAll(array, function) | Sets all elements using a lambda. |
| Arrays.stream(array) | Converts array to a stream (for functional programming). |
| Arrays.deepToString(array) | For printing multidimensional arrays. |
| Arrays.deepEquals(arr1, arr2) | Compares multidimensional arrays. |
| Arrays.asList(array) | Converts array to a fixed-size list. |

**Q. When using the Arrays.sort() method on a String array, how are the elements sorted by default?**

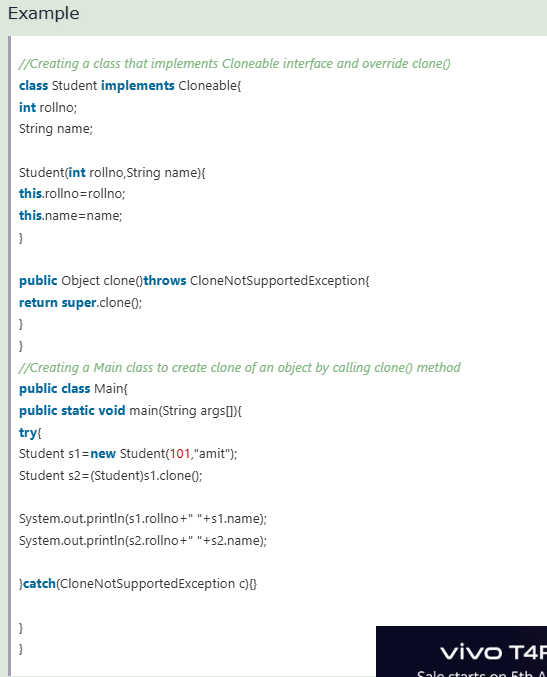
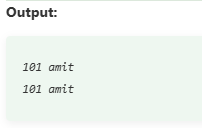
In lexicographical (alphabetical) order

**Cloneable Interface**

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

**Q. Why use clone() method?**

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

****

**Object class in Java**

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

Object obj=getObject();

**Java Math Class**

The **Java Math class** is a fundamental part of the Java language's standard library, offering a wide range of mathematical functions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.abs()](https://www.tpointtech.com/java-math-abs-method) | It will return the Absolute value of the given value. |
| [Math.max()](https://www.tpointtech.com/java-math-max-method) | It returns the Largest of two values. |
| [Math.min()](https://www.tpointtech.com/java-math-min-method) | It is used to return the Smallest of two values. |
| [Math.round()](https://www.tpointtech.com/java-math-round-method) | It is used to round of the decimal numbers to the nearest value. |
| [Math.sqrt()](https://www.tpointtech.com/java-math-sqrt-method) | It is used to return the square root of a number. |
| [Math.cbrt()](https://www.tpointtech.com/java-math-cbrt-method) | It is used to return the cube root of a number. |
| [Math.pow()](https://www.tpointtech.com/java-math-pow-method) | It returns the value of first argument raised to the power to second argument. |
| [Math.signum()](https://www.tpointtech.com/java-math-signum-method) | It is used to find the sign of a given value. |
| [Math.ceil()](https://www.tpointtech.com/java-math-ceil-method) | It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer. |
| [Math.copySign()](https://www.tpointtech.com/java-math-copysign-method) | It is used to find the Absolute value of first argument along with sign specified in second argument. |
| [Math.nextAfter()](https://www.tpointtech.com/java-math-nextafter-method) | It is used to return the floating-point number adjacent to the first argument in the direction of the second argument. |
| [Math.nextUp()](https://www.tpointtech.com/java-math-nextup-method) | It returns the floating-point value adjacent to d in the direction of positive infinity. |
| [Math.nextDown()](https://www.tpointtech.com/java-math-nextdown-method) | It returns the floating-point value adjacent to d in the direction of negative infinity. |
| [Math.floor()](https://www.tpointtech.com/java-math-floor-method) | It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value. |
| [Math.floorDiv()](https://www.tpointtech.com/java-math-floordiv-method) | It is used to find the largest integer value that is less than or equal to the algebraic quotient. |
| [Math.random()](https://www.tpointtech.com/java-math-random-method) | It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |
| [Math.rint()](https://www.tpointtech.com/java-math-rint-method) | It returns the double value that is closest to the given argument and equal to mathematical integer. |
| [Math.negateExact()](https://www.tpointtech.com/java-math-negateexact-method) | It is used to return the negation of the argument, throwing an exception if the result overflows an int or long. |
| [Math.toIntExact()](https://www.tpointtech.com/java-math-tointexact-method) | It returns the value of the long argument, throwing an exception if the value overflows an int. |

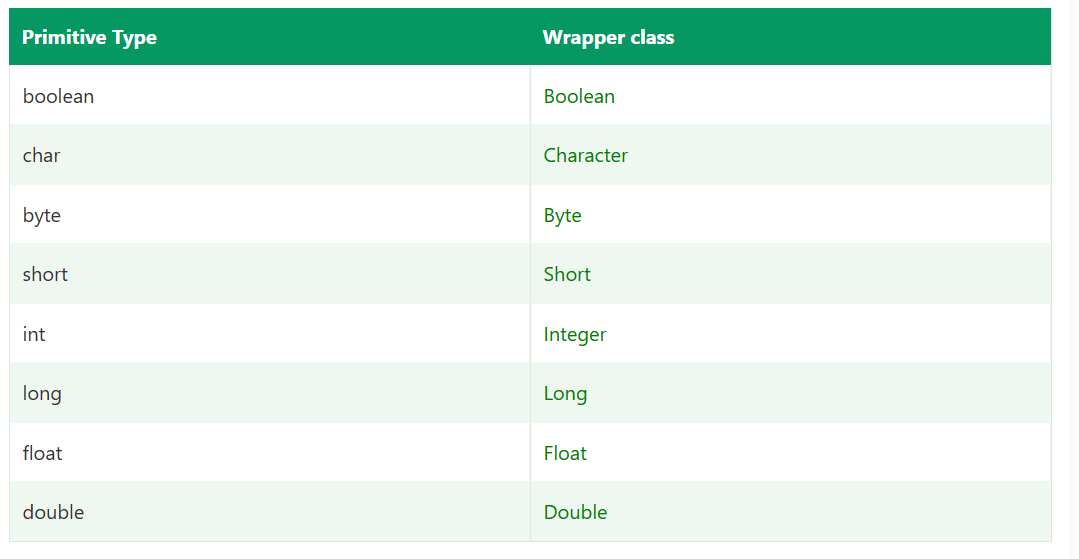
**Wrapper classes in Java**

**Uses:**

1. On object data we can call multiple methods compareTo(), equals(), toString()
2. Cloning process only objects
3. Object data allowed null values.
4. **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
5. **Synchronization:** Java synchronization works with objects in Multithreading.
6. **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

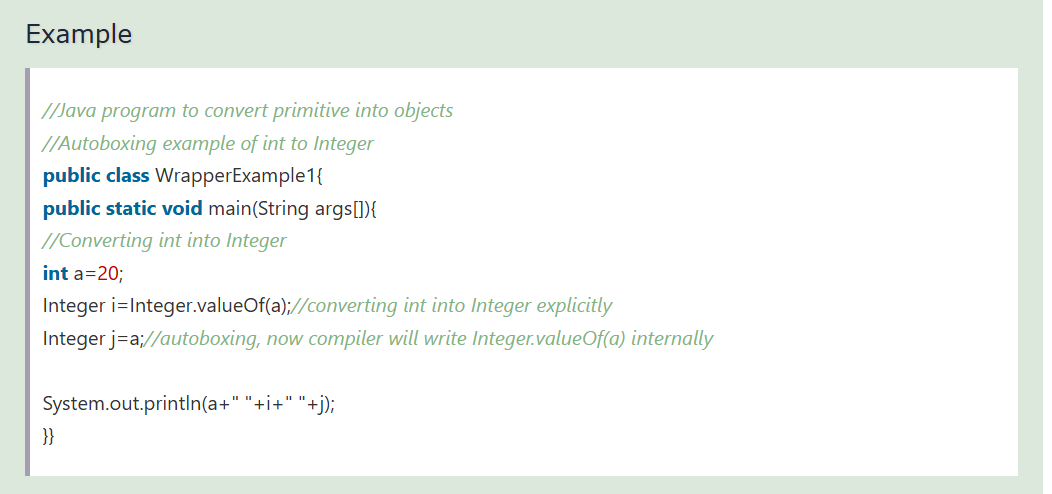
**Significance of Wrapper Class in Java**

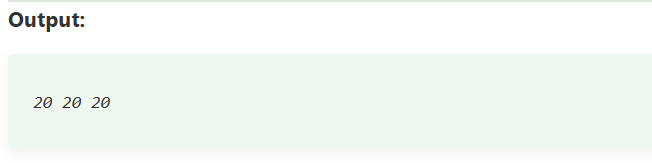
* Encapsulation
* Type Conversion
* Utility Methods
* Null Handling



**Autoboxing :**

automatic conversion of primitive data type into its corresponding wrapper class





**Unboxing:**

automatic conversion of wrapper type into its corresponding primitive type.



**Call by Value and Call by Reference in Java**

**Call by value:**

* Java **always** uses call by value, even for objects.
* When a primitive type (like int, double, float) is passed to a method, a **copy** of the value is made.
* Changes inside the method **do not** affect the original variable

**Call by reference**

* Java **does not** support true call by reference; it passes the reference by value.
* When an object is passed, a **copy of the reference** is made.
* The reference still points to the same object, so modifications **inside the method** affect the original object.

**Java Enums (Enumeration)**

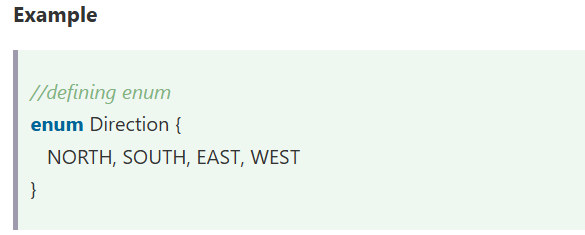
**Java enum** (Enumeration) is a data type that is used when we need to represent a fixed set of constants.

Example of enum

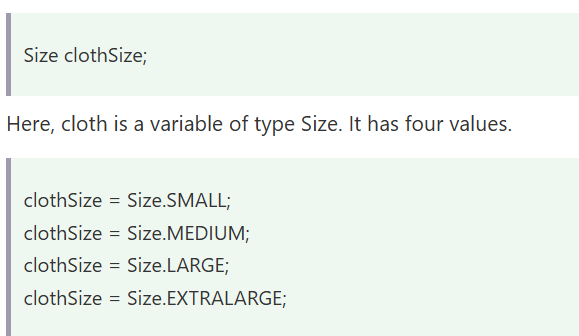
**Days:** SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY

**Directions:** NORTH, SOUTH, EAST, and WEST

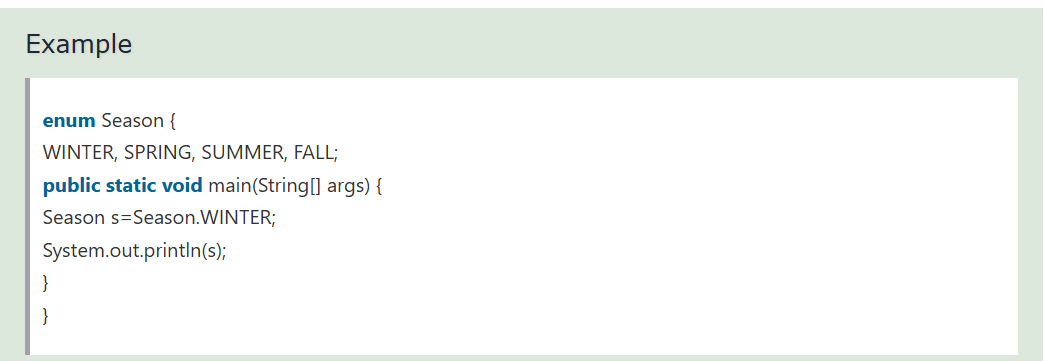
Example:



Using enum as a variable

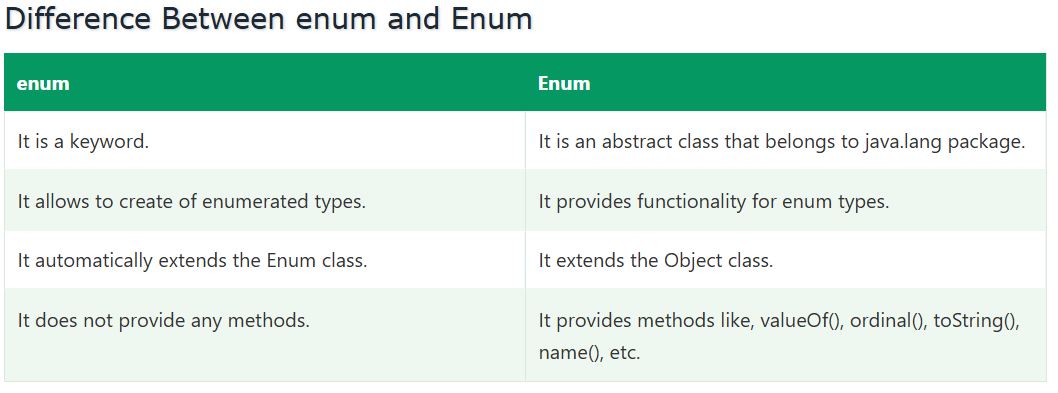


Accessing:









**Q. Enum constructors are \_\_\_\_\_\_\_\_\_ by default?**

Private

**Difference between Object and Class in Java**

**Object:**

* Instance of a Class: An object is a concrete implementation of a class.
* Physical Entity: It occupies memory in the heap at runtime.
* State and Behavior:
  + State is stored in fields (variables).
  + Behavior is exhibited through methods.
* Object Creation: Objects are typically created using the new keyword.

**Class:**

* Blueprint: A class is a blueprint or prototype from which objects are created.
* Logical Entity: It doesn't consume memory until an object is created.
* Defined with class keyword: Every class in Java is declared using the class keyword.
* Abstraction: It abstracts the implementation details and shows only the structure.
* Encapsulation: It can group data (fields) and code (methods) into a single unit.

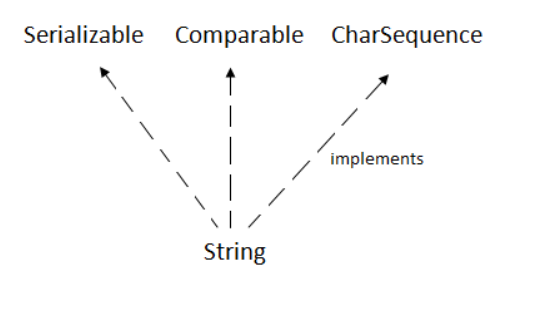
**Dynamic Method Dispatch**

Dynamic method dispatch or run-time polymorphism is the mechanism through which the correct version of an overridden method is called at runtime.

If the reference variable is of the parent class and the object is of the sub class then the method is going to be called of the subclass that is overridden is done by method displacher;

**Java Strings**

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* [interfaces](https://www.tpointtech.com/interface-in-java).



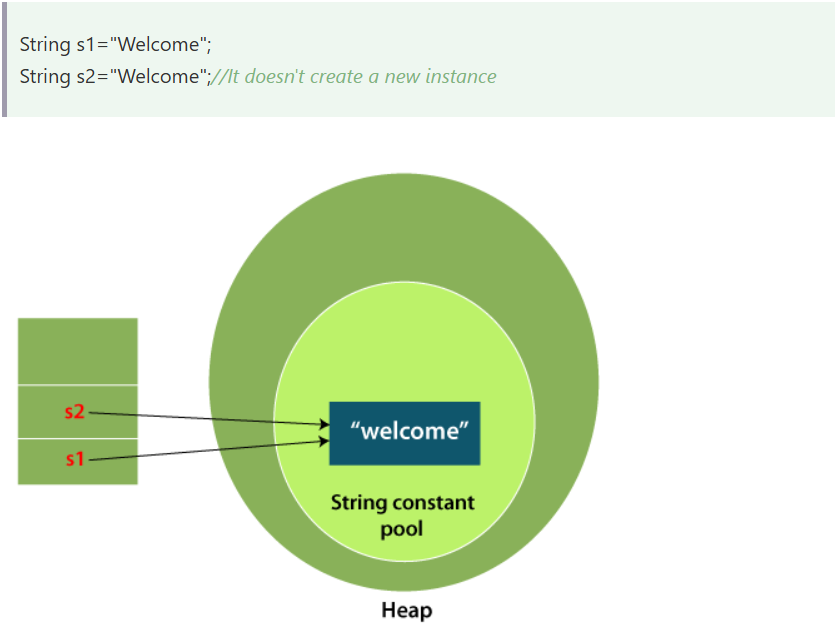
**CharSequence Interface**

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.tpointtech.com/StringBuffer-class) and [StringBuilder](https://www.tpointtech.com/StringBuilder-class) classes implement it. It means, we can create strings in Java by using these three classes.



There are two ways to create String object:

1. By string literal
2. By new keyword



On using the new keyoword separate memory will allocate for the both object in non-heap area.

**✅ Summary Table**

| **Feature** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- |
| **Thread-Safe** | ❌ No | ✅ Yes |
| **Synchronization** | ❌ No | ✅ Yes |
| **Performance** | ✅ Faster | ❌ Slower |
| **Use Case** | Single-threaded | Multi-threaded |

**Q. Which method can be used to compare two strings lexicographically in Java?**

compareTo()

**Q. How can you create a new string object from a character array in Java?**

1. new String(chars)
2. String.valueOf(chars)
3. String.copyValueOf(chars)

**Java String Methods Table**

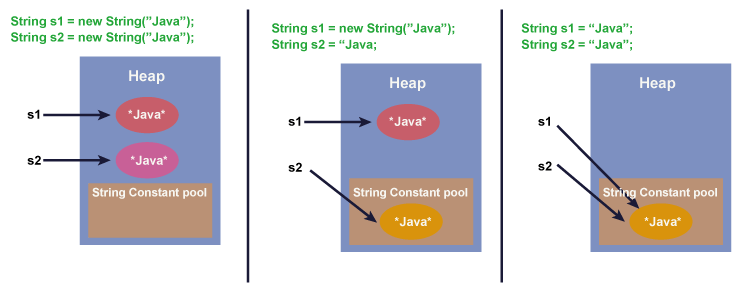
| **Category** | **Method** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Length** | length() | Returns length of the string | "Hello".length() → 5 |
| **Character** | charAt(int index) | Returns char at specified index | "Java".charAt(1) → a |
| **Case** | toUpperCase() | Converts string to uppercase | "hello".toUpperCase() → HELLO |
|  | toLowerCase() | Converts string to lowercase | "JAVA".toLowerCase() → java |
| **Comparison** | equals(String s) | Compares contents (case-sensitive) | "abc".equals("abc") → true |
|  | equalsIgnoreCase(String s) | Compares ignoring case | "Java".equalsIgnoreCase("java") → true |
|  | compareTo(String s) | Lexicographical comparison | "a".compareTo("b") → -1 |
| **Search** | contains(CharSequence s) | Checks if substring exists | "hello".contains("he") → true |
|  | startsWith(String prefix) | Checks if starts with prefix | "hello".startsWith("he") → true |
|  | endsWith(String suffix) | Checks if ends with suffix | "hello".endsWith("lo") → true |
|  | indexOf(String s) | Returns first index of substring | "hello".indexOf("e") → 1 |
|  | lastIndexOf(String s) | Returns last index of substring | "hello".lastIndexOf("l") → 3 |
| **Substring** | substring(int begin) | Returns substring from index to end | "hello".substring(2) → llo |
|  | substring(int begin, int end) | Returns substring between indices | "hello".substring(1,4) → ell |
| **Replace** | replace(char old, char new) | Replaces all old chars with new | "java".replace('a','o') → jovo |
|  | replaceAll(String regex, String repl) | Replaces using regex | "abc123".replaceAll("\\d","") → abc |
| **Trim** | trim() | Removes leading & trailing spaces | " hi ".trim() → hi |
|  | strip() *(Java 11+)* | Similar to trim(), handles Unicode | " hi ".strip() → hi |
| **Split & Join** | split(String regex) | Splits string into array | "a,b,c".split(",") → [a,b,c] |
|  | join(CharSequence delim, elements...) | Joins elements with delimiter | String.join("-", "a","b") → a-b |
| **Conversion** | toCharArray() | Converts to char array | "java".toCharArray() → [j,a,v,a] |
|  | getBytes() | Converts to byte array | "abc".getBytes() → [97,98,99] |

**Q. Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

**Q. Why String class is Final in Java?**

The reason behind the 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.



here are three ways to compare String in Java:

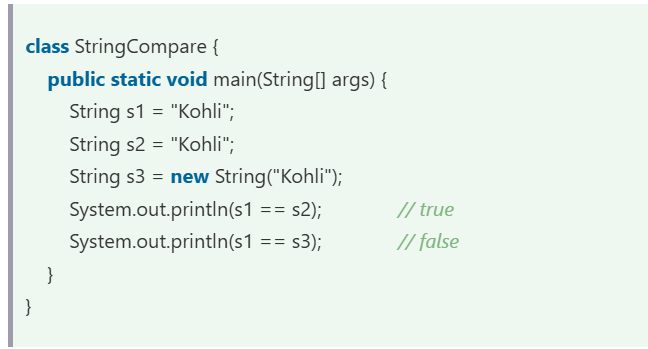
1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method
4. Using startsWith() and endsWith() Method

1) By Using equals() Method

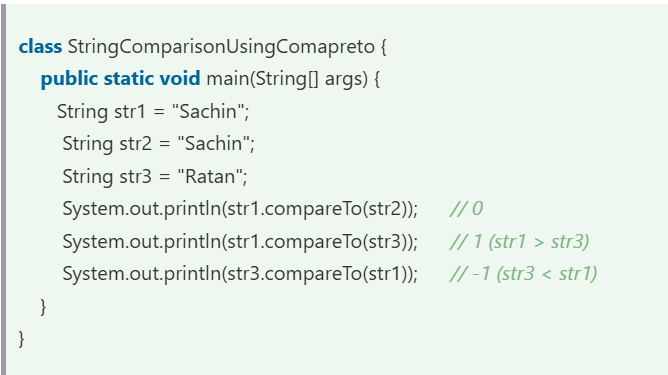
The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

* public boolean equals(Object another
* public boolean equalsIgnoreCase(String another)

2) By Using == Operator

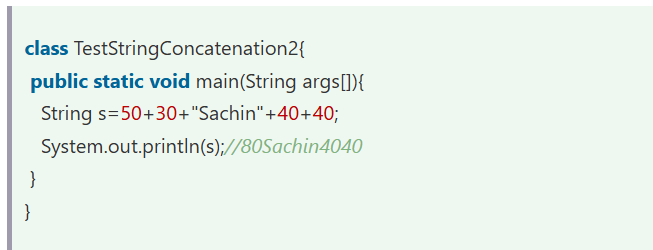


3) String compare by compareTo() Method

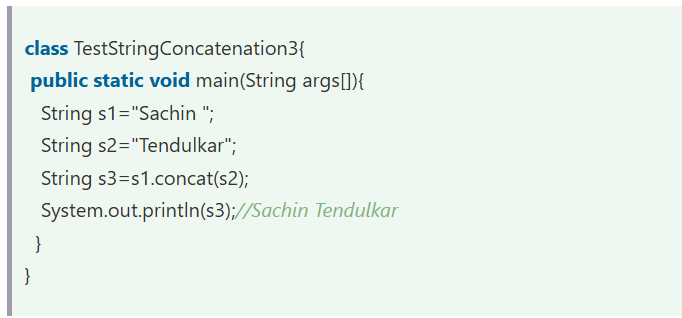


String Concatenation

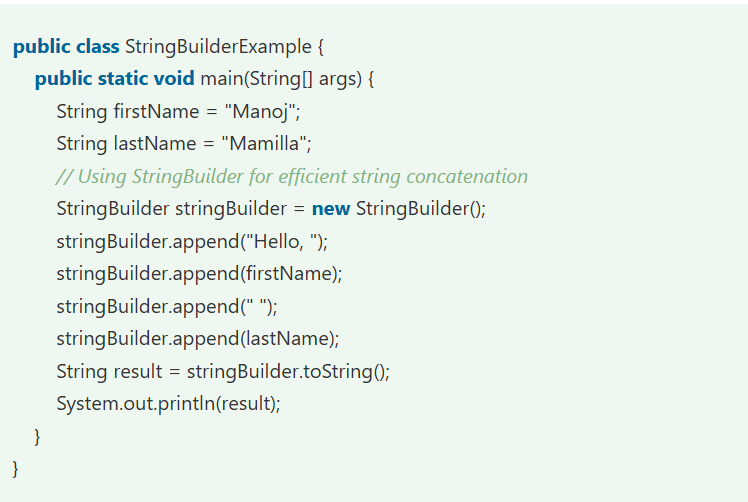
1. Using "+" (String concatenation) Operator



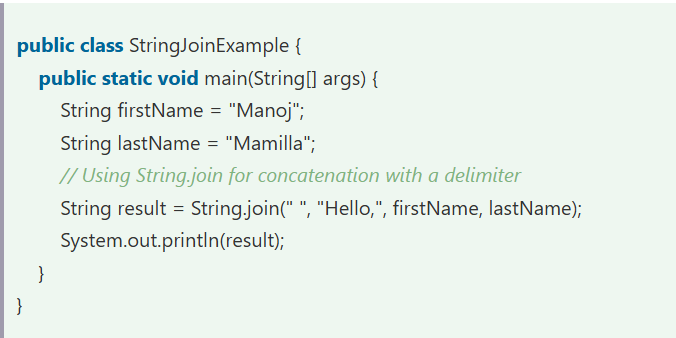
1. Using String.concat() Method



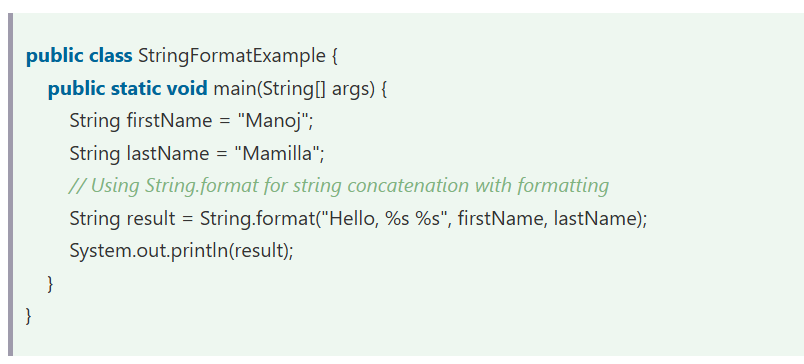
1. Using the StringBuilder or StringBuffer Class



1. Using String.join() Method



1. Using Java String.format() Method

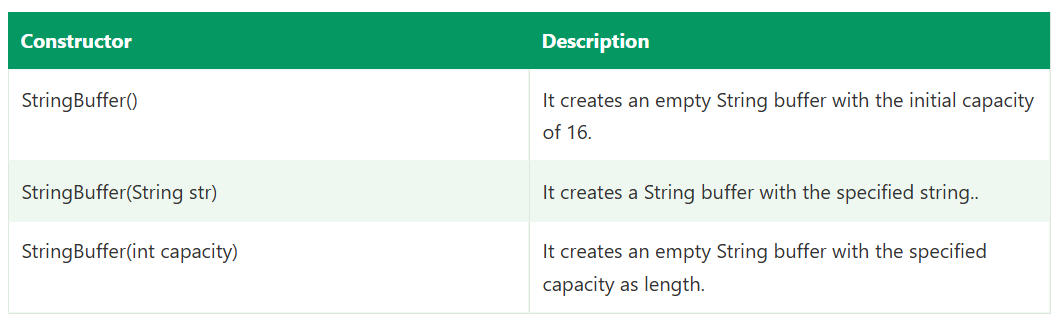


1. Using Collector.joining() Method

Note : If in the substring the start and the end index are same then it will return the empty “” string.

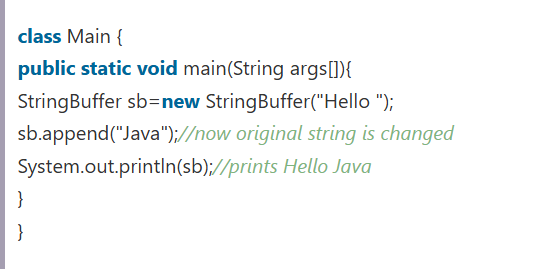
**String Buffer :**

A String that can be modified or changed is known as a mutable String.

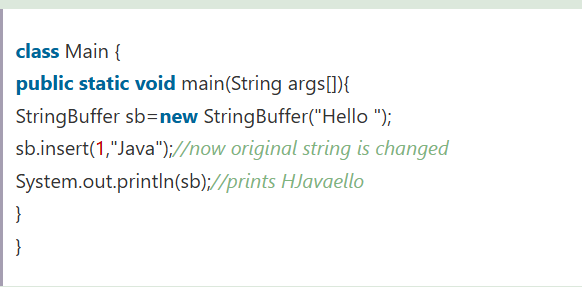


Methods:

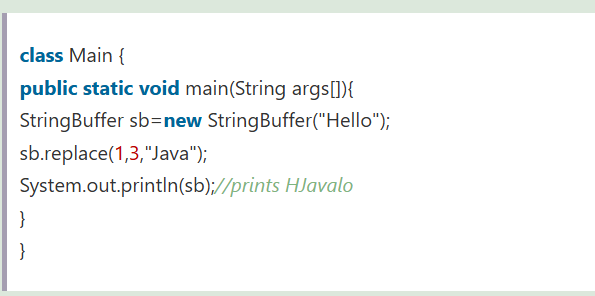
1.append()



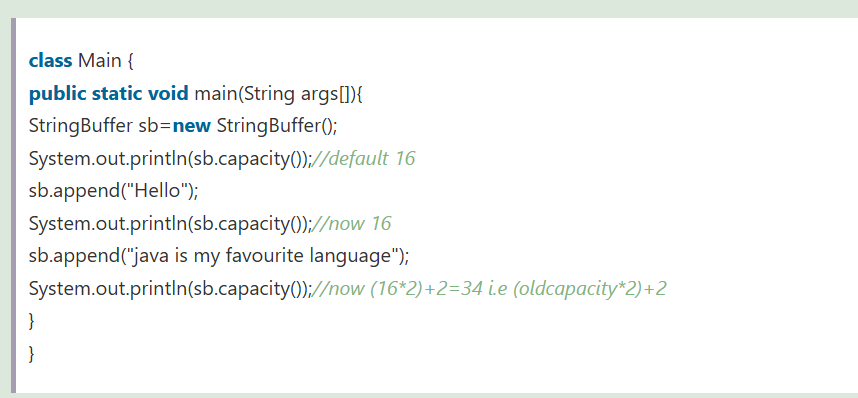
2. insert()



3.replace()



4. capacity()



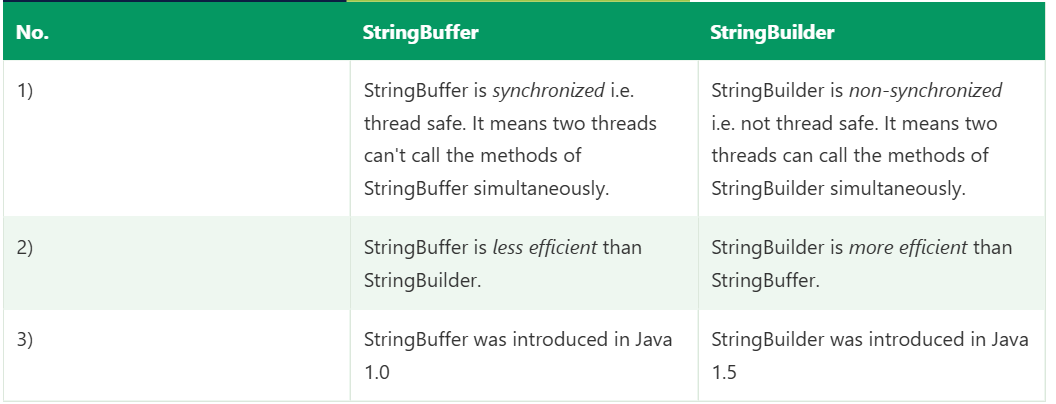
5. reverse()

6. delete()

Q. **Which of the following operations is more efficient when using StringBuffer compared to String?**

Concatenating multiple strings.

**StringBuffer vs StringBuilder**



**Q. Which class should be used for single-threaded applications where performance is a concern?**

StringBuilder

**Q. Which of the following operations will modify the underlying array of a StringBuffer to increase its capacity if it exceeds the current capacity?**

ensureCapacity()

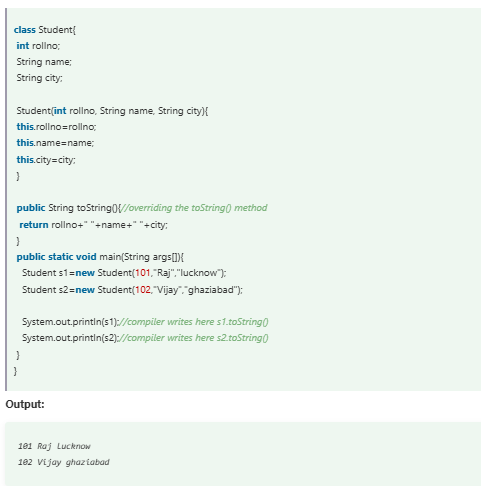
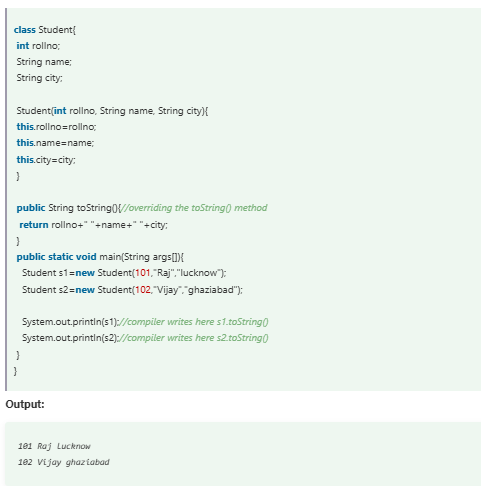
* The ensureCapacity() method ensures that the StringBuffer's capacity is at least equal to the specified minimum. If the current capacity is less than the minimum, a new array with greater capacity is allocated.

Note : Class can be immutable using the final keyword.

**Java toString() Method**

**Q. What is the default behavior of the toString() method in the Object class?**

Returns the object's class name and hashcode



**MultiThreading**

**Fairness of Locks:**

* Ensurity that thread will get chance to run.

**⚖️ What does *fairness* mean in ReentrantLock?**

* **Fair lock (true)**:  
  Threads acquire the lock **in the order they requested it** (FIFO – First In, First Out).  
  If Thread A tries to acquire the lock before Thread B, then A will get the lock before B (assuming both are waiting).
* **Unfair lock (false, the default)**:  
  The lock may **“barge”** and give preference to threads that request it most recently.  
  This can improve throughput (faster overall performance), but it may cause **starvation** (a thread might wait too long if others keep barging in).

**Example:**

Suppose you have 3 threads (T1, T2, T3):

* With **fair lock (true)**:  
  If T1 requested first, then T2, then T3, the lock will be given to them in **T1 → T2 → T3** order.
* With **unfair lock (false)**:  
  Even if T1 is waiting, a newly arrived thread T3 might grab the lock before T1, leading to out-of-order execution.

**Thread Lifecycle**

The **life cycle of a thread** in Java (and most programming models) represents the different states a thread goes through from creation to termination.

Here are the main **thread life cycle states**:

**1. New (Created state)**

* When you create a thread object using the Thread class or by implementing Runnable, but you haven’t started it yet.
* Example:
* Thread t = new Thread();
* At this point, the thread is not yet running.

**2. Runnable (Ready state)**

* When you call start() on the thread, it moves to the runnable state.
* It is ready to run but **waiting for the CPU scheduler** to allocate CPU time.
* Example:
* t.start();
* The thread is now eligible for execution, but whether it runs immediately depends on the CPU scheduler.

**3. Running**

* When the CPU scheduler picks the thread from the runnable state, it starts executing its run() method.
* Only one thread per core can be in the running state at a time.
* Example:
* public void run() {
* System.out.println("Thread is running...");
* }

**4. Blocked / Waiting**

A thread can leave the running state and enter waiting states:

* **Waiting state**:
  + A thread is waiting **indefinitely** for another thread to signal (using notify() or notifyAll()).
  + Example:
  + synchronized(lock) {
  + lock.wait(); // waits indefinitely
  + }
* **Timed Waiting**:
  + A thread waits for a specific period of time before becoming runnable again.
  + Example:
  + Thread.sleep(1000); // waits for 1 second
  + t.join(500); // waits for 0.5 seconds

**5. Terminated (Dead)**

* Once the run() method finishes execution, the thread enters the **terminated state**.
* It cannot be restarted (calling start() again will throw IllegalThreadStateException).

✅ **Thread life cycle diagram (simplified):**

New (Created)

|

v

Runnable <----> Waiting / Timed Waiting / Blocked

|

v

Running

|

v

Terminated (Dead)

**🔹 What is Thread Safety?**

A **thread-safe** piece of code is one that **works correctly when accessed by multiple threads at the same time**.

If code is not thread-safe, two or more threads may:

* Corrupt shared data,
* Produce unpredictable results,
* Cause race conditions or deadlocks

**🔹 What is a Race Condition?**

A **race condition** happens when **two or more threads/processes access shared data at the same time**, and the final outcome depends on **who runs first (the timing of execution)**.

👉 This makes the program’s behavior **unpredictable and inconsistent**.

**Kunal OOPs**

* Final can not be reinitialize and can not be re assigned.

Example:

int a = 10;

a = 20; //no re-initialization

A a = new A();

A b = new A();

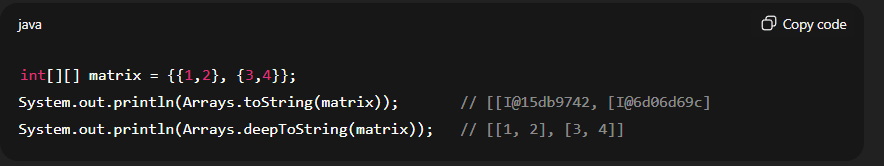
a = b; //no re-assingment

* A method can not be decleared under a method but can be called instead.
* Static method can be called under the non-static because static is class specific and does not depend on the object.

Another reason is that static has a big scope can be passed under low scope.

* Non static can be passed under the static by making the object of the class to call the non-static method.

**Multi-dimensional array : (**Arrays.toString()**)**



**Note:** Arrays.toString() on a 2D array prints object references — for nested arrays, always use Arrays.deepToString().

✅ **Summary**

* Arrays.toString(array) → converts 1D array to readable string.
* Arrays.deepToString(array) → converts multi-dimensional arrays to readable string.