**String**

In Java String is an Array of characters or object that represent sequence of char values.

String is a final class which implements **Serializable, Comparable<String>, CharSequence Interfaces**.

String class provide so many methods for perform an operations on strings such as concat(), compare(), equal() etc.

We can create a string using **String, StringBuffer, StringBuilder class** which implements **CharSequence interface**.

We can create String in Java by two way

1. Using String Literals – String str = “ABC”;
2. Using New Keyword – String str = new String(“ABC”);

Now Using Literals for creating Strings then it will store this String into String Constant Pool which is present in Heap memory.

String Constant Pool is memory area in Heap memory for storing a string literals.

Whenever we will create a string by literals the firstly JVM will check this String is present in String Constant Pool or not if.

If string is present in pool then reference is return for that instance.

If string is not present in pool then new string will create in this pool.

For Example

String str1 = “ABC”;

String str2 = “ABC”;

Here we create String str1, It JVM will first check in pool this “ABC” is present or not. Initially pool is empty the it will store this string in String constant pool.

Now we create String str2 now again JVM will check this “ABC” in pool but now this time this String is present in pool so it will return this reference to it.

There for those both are pointing to the same reference and because of that hashcode is same for both.

HashCode is an integer value that is associated with each object in Java.

Now if using new Keyword

String str1 = new String(“ABC”);

String str2 = new String(“ABC”);

Using new Keyword JVM will create new string object in heap memory and then literals will placed in poll.

This is known as Immutability of String.

Advantage of making String Immutable in Java is

1. Memory Efficiency.
2. Security for Data.

Why String class is final – because no one can override the method of string class.

Three ways a compare a string in Java.

1. equals() – compare the value of that string objects
2. == operator – it compare the reference of that object.
3. compareTo() – it will compare Lexicographically of strings and return 0 if equal, 1 if greater and -1 if smaller.

**toString()** Method – If we want to represent any object as string toString() method will return the string representation of object.

**StringBuffer and StringBuilder**

Both those classes is use for creating mutable String object in Java.

So many methods are present in this classes for manipulating to Original String.

| **Feature** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- |
| Thread Safety | Thread-safe | Not thread-safe |
| Synchronization | Methods are synchronized, leading to slower performance in multi-threaded environments | Not synchronized, faster performance in single-threaded environments |
| Mutable | Mutable (contents can be changed) | Mutable (contents can be changed) |
| Performance | Slower due to synchronization overhead | Faster due to lack of synchronization overhead |
| Introduced in | Introduced in Java 1.0 | Introduced in Java 5.0 |

**How to create Immutable Class in Java?**

To create an immutable object in Java, you should follow these steps:

Step 1: Declare all fields of the class as final and private.

Step 2: Initialize the fields in the constructor.

Step 3: Do not provide setter methods for changing the state of the object.

| **Comparable** | **Comparator** |
| --- | --- |
| Use for Single Sorting | Use for multiple Sorting |
| Having compareTo() method | Having compare() method |
| Affect original class | Doesn’t affect original class |
| Java.lang | Java.util |
| Collections.sort(List) | Collections.sort(List,Comparator) |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Immutable  An **immutable in** Java is one whose instances cannot be modified after creation. Once an object of an immutable class is instantiated, then its state remains constant throughout its lifecycle.  **Here are some In-built immutable classes in Java:**In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String classes are immutable.  The **String** class in Java is a classic example of an immutable class.  Create Immutable Class in Java  **Make all Fields private and final:**  - **private** restricts direct access to the fields from outside the class.  - **final** ensures that the fields can be assigned a value only once.  **Provide a Constructor for Initialization**:  - All fields should be initialized through the constructor. This ensures that the object's state is fully defined at creation.  **Do Not Provide Setter Methods:**  **-** Setter methods allow modification of the object's state after creation, which contradicts immutability.  **public final class ImmutablePerson {**  **private final String name;**  **private final int age;**  **private final List<String> hobbies; // Mutable field**  **public ImmutablePerson(String name, int age, List<String> hobbies) {**  **this.name = name;**  **this.age = age;**  **// Deep copy for mutable field**  **this.hobbies = new ArrayList<>(hobbies);**  **}**  **public String getName() {**  **return name;**  **}**  **public int getAge() {**  **return age;**  **}**  **public List<String> getHobbies() {**  **// Return a defensive copy for mutable field**  **return Collections.unmodifiableList(new ArrayList<>(hobbies));**  **}**  **}** | | **Feature** | **String** | **StringBuffer** | **StringBuilder** | | --- | --- | --- | --- | | **Mutability** | Immutable (cannot be changed after creation) | Mutable (can be changed without creating new object) | Mutable (can be changed without creating new object) | | **Thread Safety** | Thread-safe (immutable objects are inherently safe) | Thread-safe (methods are synchronized) | **Not thread-safe** (methods are not synchronized) | | **Performance** | Slower for repeated modifications (new object each time) | Slower than StringBuilder due to synchronization overhead | Faster than StringBuffer (no synchronization overhead) | | **Use Case** | Best for fixed or rarely changing text (e.g., constants, messages, keys) | Best for strings modified by multiple threads concurrently | Best for strings modified frequently in a single-threaded context | | **Package** | java.lang.String | java.lang.StringBuffer | java.lang.StringBuilder | | **Introduced In** | JDK 1.0 | JDK 1.0 | JDK 1.5 | | **Example** | String s = "Hello"; s.concat(" World"); // creates new object | StringBuffer sb = new StringBuffer("Hello"); sb.append(" World"); // modifies same object | StringBuilder sb = new StringBuilder("Hello"); sb.append(" World"); // modifies same object |    Use **String** when the content won’t change.   Use **StringBuffer** when content changes **and** thread safety is required.   Use **StringBuilder** when content changes **a lot** and thread safety is **not** required (faster). | | **Feature** | **final (Keyword)** | **finally (Block)** | **finalize (Method)** | | --- | --- | --- | --- | | **Definition** | A keyword used to restrict variables, methods, and classes. | A block in exception handling that executes whether an exception occurs or not. | A method in Object class, called by Garbage Collector before object destruction. | | **Usage** | - final variable = constant (cannot be reassigned). - final method = cannot be overridden. - final class = cannot be inherited. | Ensures important cleanup code (like closing files, releasing resources) is always executed. | Used for cleanup operations before an object is garbage collected. | | **When Executed** | At **compile-time restrictions**. | At **runtime**, after try/catch (always executes, unless System.exit() is called). | At **runtime**, just before GC destroys the object. | | **Inheritance** | Prevents inheritance/overriding. | Not related to inheritance. | Can be overridden (but not recommended). | | **Example** | java final int age = 18; final void show() { } final class Test { } | java try { int a=10/0; } finally { System.out.println("finally executed"); } | java protected void finalize() { System.out.println("finalize() called"); } |   ✅ **final** → **Restriction** (no change).  ✅ **finally** → **Cleanup block** (always runs).  ✅ **finalize** → **Cleanup method** (called by Garbage Collector, but not guaranteed to run immediately or always).  Since **Java 9**, the use of finalize() is **deprecated**, because it can cause performance and reliability issues. Instead, Java recommends using **try-with-resources** or explicit cleanup methods. |
| **String**  We can create String in Java by two way  - Using String Literals – String str = “ABC”;  - Using New Keyword – String str = new String(“ABC”);  Now Using Literals for creating Strings then it will store this String into String Constant Pool which is present in Heap memory.  String Constant Pool is memory area in Heap memory for storing a string literals.  Whenever we will create a string by literals the firstly JVM will check this String is present in String Constant Pool or not if.  If string is present in pool then reference is return for that instance.  If string is not present in pool then new string will create in this pool.  For Example  String str1 = “ABC”;  String str2 = “ABC”;  Here we create String str1, It JVM will first check in pool this “ABC” is present or not. Initially pool is empty the it will store this string in String constant pool.  Now we create String str2 now again JVM will check this “ABC” in pool but now this time this String is present in pool so it will return this reference to it.  There for those both are pointing to the same reference and because of that hashcode is same for both.  HashCode is an integer value that is associated with each object in Java.  Now if using new Keyword  String str1 = new String(“ABC”);  String str1 = new String(“ABC”);  Using new Keyword JVM will create new string object in heap memory and then literals will placed in poll.  This is known as Immutability of String.  Advantage of making String Immutable in Java is  - Memory Efficiency.  - Security for Data.  Why String class is **final** – because no one can override the method of string class. | **== and equals()**  == is an operator; equals() is a method.  == compares references (for objects) or values (for primitives); equals() compares content (for objects).  == cannot be overridden; equals() can be overridden to define custom equality logic.  equals() is not applicable to primitive types directly.  In Java, == and equals() are both used for comparison, but they serve different purposes and operate on different aspects of data:  == Operator:  **Purpose:** The == operator compares references for objects and values for primitive types.  **Primitive Types:** When used with primitive data types (like int, char, boolean, float, double), == directly compares their literal values.  **Objects:** When used with objects, == checks if two object references point to the exact same memory location (i.e., if they are the same object instance). It does not compare the content of the objects.  equals() Method:  **Purpose:** The equals() method is a method defined in the Object class (and overridden in many other classes like String, Integer, etc.) used to compare the content or value of objects.  **Default Behavior:** By default, the equals() method inherited from Object performs the same comparison as == (i.e., it checks for reference equality).  **Overriding:** For custom classes, you can override the equals() method to define what constitutes "equality" for your objects based on their internal state or attributes. For example, a Person class might override equals() to consider two Person objects equal if they have the same name and age, regardless of whether they are the same object instance.  **Usage with Primitives:** The equals() method cannot be directly used with primitive types as they are not objects and do not have methods. | **Wrapper Class**  Wrapper class is providing a way to use primitive data types as objects.  Each primitive data type has a corresponding wrapper class.  Wrapper class provide a method to convert primitive data type to object (Boxing) and Object into primitive (unboxing).  We are using wrapper class in situation where we need Object such as Collection like ArrayList or working with generic which does not support directly primitive data type.  **Object Cloning**  Object cloning mean the creating of the exact copy of the existing object.  clone() method is use to clone an object.  java.lang.Clonable interface must be implemented by the class which we want to create clone object.  If we do not implement cleanable interface then it will throw CloneNotSupportException.  Main use of cloning is if there is some lengthy or time-consuming task is present so it will reduce this and clone it for us.  When we clone the object then it will create copy of that object.  There is two type which is Shallow copy and Deep copy.  **Shallow Copy**  Simply means it will create copy of that object and also copy all reference of object which is in object.  If we make some changes in copied object then it will reflect main object.  **Deep Copy**  In this copied object will create new objects and if we make changes in copied object then it will not reflect in main object. |
|  |  |  |