Q. Difference between final, finally, finalize ?

**final**, **finally**, and **finalize** are three distinct concepts in Java:

1. **final:**
   * **final** is a keyword in Java that can be applied to variables, methods, and classes.
   * When applied to a variable, it means the variable is a constant, and its value cannot be changed once assigned.
   * When applied to a method, it means the method cannot be overridden by subclasses.
   * When applied to a class, it means the class cannot be subclassed.

Examples:

javaCopy code

final int x = 10; // Constant variable final void myMethod() { /\* Code \*/ } // Cannot be overridden final class MyClass { /\* Code \*/ } // Cannot be subclassed

1. **finally:**
   * **finally** is a block in Java used in conjunction with a **try** block to define code that will be executed regardless of whether an exception is thrown or not.
   * The code within the **finally** block is guaranteed to be executed, providing a mechanism for cleanup or resource release.
   * It is commonly used to close resources like files, database connections, or network connections.

Example:

javaCopy code

try { // Code that may throw an exception } catch (Exception e) { // Handle the exception } finally { // Code that will be executed whether an exception is thrown or not // For example, close resources here }

1. **finalize:**
   * **finalize** is a method in the **Object** class in Java.
   * It is called by the garbage collector before an object is reclaimed, allowing the object to perform cleanup or release resources.
   * It's important to note that relying on **finalize** for resource cleanup is not recommended. It's often better to use **try-finally** or other mechanisms for resource management.

Example:

javaCopy code

class MyClass { // Other class members @Override protected void finalize() throws Throwable { // Code to release resources or perform cleanup super.finalize(); } }

In summary:

* **final**: Keyword used to make variables constants, methods unmodifiable, or classes unextendable.
* **finally**: Block used in exception handling to ensure that certain code is executed regardless of whether an exception is thrown.
* **finalize**: Method in the **Object** class called by the garbage collector before an object is garbage collected. It allows for cleanup or resource release, but it is not the preferred mechanism for resource management.

The keywords final, finally, and finalize are all used in Java, but they have different purposes.

* final

is used to declare variables, methods, and classes that cannot be modified after they are initialized. For example, you might use the final keyword to declare a constant value, such as the number of days in a year.

* finally

is used in exception handling to ensure that a section of code is always executed, even if an exception is thrown. For example, you might use the finally keyword to close a file that was opened in a try block.

* finalize

is a method that is called by the garbage collector before an object is reclaimed. This method can be used to perform cleanup tasks, such as closing database connections.

Q. Difference between string, stringbuilder & stringbuffer in java?

In Java, **String**, **StringBuilder**, and **StringBuffer** are all classes that represent sequences of characters, but they have some key differences in terms of mutability, performance, and synchronization.

1. **String:**
   * Immutable: Once a **String** object is created, its content cannot be changed. Any operation that appears to modify the string actually creates a new string.
   * Thread-safe: Since strings are immutable, they are inherently thread-safe. Multiple threads can safely use the same string object without any issues.
   * Use cases: Suitable for situations where the content does not need to be changed frequently, such as representing constant values or string literals.

Example:

javaCopy code

String str = "Hello"; str = str + " World"; // This creates a new string object

1. **StringBuilder:**
   * Mutable: **StringBuilder** is mutable, meaning that you can modify its content without creating a new object.
   * Not thread-safe: **StringBuilder** is not synchronized, so if multiple threads access a **StringBuilder** concurrently and at least one of the threads modifies it, external synchronization is needed.
   * Better performance: **StringBuilder** is generally more efficient than **String** when it comes to concatenating or modifying strings, as it avoids the overhead of creating new objects.

Example:

javaCopy code

StringBuilder sb = new StringBuilder("Hello"); sb.append(" World"); // Modifies the same object

1. **StringBuffer:**
   * Mutable: Like **StringBuilder**, **StringBuffer** is also mutable, allowing you to modify its content.
   * Thread-safe: Unlike **StringBuilder**, **StringBuffer** is synchronized, making it safe for use in multithreaded environments.
   * Slower performance: Due to synchronization, **StringBuffer** is generally slower than **StringBuilder**. It is advisable to use **StringBuilder** unless thread safety is explicitly required.

Example:

javaCopy code

StringBuffer sbf = new StringBuffer("Hello"); sbf.append(" World"); // Modifies the same object

In summary, choose the appropriate class based on your specific requirements:

* Use **String** when the content is fixed or when immutability is desired.
* Use **StringBuilder** for better performance in a single-threaded environment when frequent modifications are needed.
* Use **StringBuffer** when thread safety is required in a multithreaded environment, even though it may have a performance cost compared to **StringBuilder**.

Here are the differences between String, StringBuilder, and StringBuffer in Java:

* String:

A String is an immutable class, meaning that its value cannot be changed once it is created. This makes Strings thread-safe, as there is no need to worry about synchronization. However, it also means that String operations can be inefficient, as a new String object must be created whenever the value of the String needs to be changed.

* StringBuilder:

StringBuilder is a mutable class, meaning that its value can be changed. StringBuilder is not thread-safe, as it does not provide any synchronization mechanism. However, StringBuilder operations are generally more efficient than String operations, as StringBuilder does not need to create a new object whenever the value of the StringBuilder needs to be changed.

* StringBuffer:

StringBuffer is a mutable class that is thread-safe. This means that StringBuffer provides a synchronization mechanism to ensure that multiple threads can access and modify the StringBuffer object without interfering with each other. However, this synchronization mechanism also makes StringBuffer operations less efficient than StringBuilder operations.

In general, StringBuilder is the best choice for most string operations, as it is efficient and does not need to be synchronized. However, if you need to perform string operations in a multi-threaded environment, you should use StringBuffer to ensure that your code is thread-safe.

Q. Difference between == & .equals() in java?

In Java, **==** and **.equals()** are used to compare objects, but they serve different purposes and behave differently:

1. **== Operator:**
   * The **==** operator is used for reference comparison, i.e., it checks if two object references point to the exact same memory location.
   * For primitive types (like **int**, **char**, etc.), **==** compares the actual values.
   * When used with objects, it checks whether the two references point to the same object in memory.

Example:

javaCopy code

String str1 = new String("Hello"); String str2 = new String("Hello"); System.out.println(str1 == str2); // false, as they are different objects in memory

1. **.equals() Method:**
   * The **.equals()** method is used for content or value comparison. It is overridden by many classes to provide a meaningful comparison of the contents of objects.
   * The default implementation in the **Object** class compares object references, but many classes override this method to compare the contents of the objects.

Example:

javaCopy code

String str1 = new String("Hello"); String str2 = new String("Hello"); System.out.println(str1.equals(str2)); // true, as the content of the strings is the same

When using **.equals()**, it's important to check if the method is overridden in the specific class you are working with. For example, the **String** class overrides **.equals()** to compare the content, while the default implementation in the **Object** class uses reference comparison.

In summary:

* Use **==** for reference comparison, i.e., checking if two references point to the same memory location.
* Use **.equals()** for content or value comparison, but be aware that the behavior depends on whether the method is overridden in the specific class. For commonly used classes like **String**, **equals()** is typically overridden for content comparison.

The main difference between == and .equals() in Java is that == compares the memory location of two objects, while .equals() compares the contents of two objects.

The == operator is used for reference equality, which means that it checks whether two objects refer to the same memory location. The .equals() method is used for value equality, which means that it checks whether two objects have the same content.

For example, if you have two String objects, "Hello" and "Hello", the == operator will return false because the two objects refer to different memory locations. However, the .equals() method will return true because the two objects have the same content.

The .equals() method is generally preferred over the == operator when comparing objects because it is more robust. For example, if you have two String objects that are created using the same literal, the == operator will return false because the two objects refer to different memory locations. However, the .equals() method will return true because the two objects have the same content.

Here is a table that summarizes the difference between == and .equals():

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | Compares the memory location of two objects |
| .equals() | Compares the content of two objects |

In general, you should use the .equals() method to compare objects. The == operator should only be used when you are sure that you want to compare the memory location of two objects.