**Summary: String vs StringBuilder**

| **Aspect** | **String** | **StringBuilder** |
| --- | --- | --- |
| **Mutability** | Immutable (cannot be changed after creation) | Mutable (can be modified) |
| **Performance** | Less efficient for frequent modifications | More efficient for frequent modifications |
| **Thread Safety** | Thread-safe (since immutable) | Not thread-safe |
| **Memory Usage** | More memory-intensive (creates new objects) | Less memory-intensive (modifies in place) |
| **Use Case** | When the string is not modified often | When the string needs to be modified often |
| **Example** | String s = "Hello"; | StringBuilder sb = new StringBuilder(); |

**Conclusion and Summary:**

| **Aspect** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- |
| **Thread Safety** | Not thread-safe (no synchronization) | Thread-safe (synchronized methods) |
| **Performance** | Faster (no synchronization overhead) | Slower (due to synchronization) |
| **Memory Usage** | More memory-efficient | Slightly more memory overhead due to synchronization |
| **Use Case** | Single-threaded environments, high performance needed | Multi-threaded environments, thread safety required |

Constructor usage in java selenium : -

In Java Selenium, constructors are used to initialize objects. They are particularly useful

when you want to set up initial conditions for your Selenium tests. Here's a simple example to illustrate how you might use a constructor in a Selenium test class:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class SeleniumTest {

WebDriver driver;

// Constructor

public SeleniumTest() {

// Set the path for the ChromeDriver

System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

// Initialize the WebDriver

driver = new ChromeDriver();

}

public void openWebsite() {

// Use the driver to open a website

driver.get("https://www.example.com");

}

public static void main(String[] args) {

// Create an instance of the SeleniumTest class

SeleniumTest test = new SeleniumTest();

// Call the method to open the website

test.openWebsite();

}

}

In this example:

* The constructor SeleniumTest() sets the path for the ChromeDriver and initializes the WebDriver instance.
* The openWebsite() method uses the driver to open a website.
* In the main method, an instance of SeleniumTest is created, and the openWebsite() method is called.

This setup ensures that every time you create an instance of SeleniumTest, the WebDriver is properly initialized and ready to use.

This & Super Keywords : -

Sure! The this and super keywords in Java are used to refer to the current class instance and the parent class instance, respectively. They can be quite useful in Selenium tests for managing inheritance and object-oriented principles. Here's how you can use them:

**this Keyword**

The this keyword refers to the current instance of the class. It's often used to differentiate between class attributes and parameters with the same name, or to call other constructors in the same class.

Example:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class SeleniumTest {

WebDriver driver;

// Constructor

public SeleniumTest(String driverPath) {

// Use 'this' to refer to the current class instance variable

this.driver = new ChromeDriver();

System.setProperty("webdriver.chrome.driver", driverPath);

}

public void openWebsite(String url) {

// Use 'this' to call another method in the same class

this.driver.get(url);

}

public static void main(String[] args) {

SeleniumTest test = new SeleniumTest("path/to/chromedriver");

test.openWebsite("https://www.example.com");

}

}

**super Keyword**

The super keyword refers to the parent class instance. It's used to call methods and constructors from the parent class.

Example:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

// Parent class

class BaseTest {

WebDriver driver;

// Constructor

public BaseTest(String driverPath) {

System.setProperty("webdriver.chrome.driver", driverPath);

this.driver = new ChromeDriver();

}

public void openWebsite(String url) {

driver.get(url);

}

}

// Child class

public class SeleniumTest extends BaseTest {

// Constructor

public SeleniumTest(String driverPath) {

// Use 'super' to call the parent class constructor

super(driverPath);

}

public static void main(String[] args) {

SeleniumTest test = new SeleniumTest("path/to/chromedriver");

test.openWebsite("https://www.example.com");

}

}

In this example:

* The BaseTest class initializes the WebDriver and has a method to open a website.
* The SeleniumTest class extends BaseTest and uses the super keyword to call the parent class constructor.

Using this and super helps manage code more efficiently, especially when dealing with inheritance and complex class structures.

Sure! The final, finally, and finalize keywords in Java serve different purposes and are used in different contexts. Here's a breakdown of each:

**final**

The final keyword can be used with classes, methods, and variables to indicate that they cannot be changed or overridden.

* **Final Class**: A class declared as final cannot be subclassed.
* public final class FinalClass {
* // Class content
* }
* **Final Method**: A method declared as final cannot be overridden by subclasses.
* public class BaseClass {
* public final void display() {
* System.out.println("This is a final method.");
* }
* }
* **Final Variable**: A variable declared as final cannot be reassigned once it has been initialized.
* public class Example {
* public final int MAX\_VALUE = 100;
* }

**finally**

The finally block is used in conjunction with try and catch blocks. It provides a mechanism to execute code regardless of whether an exception is thrown or not. It's typically used for cleanup activities, such as closing files or releasing resources.

Example:

public class Example {

public static void main(String[] args) {

try {

int data = 25 / 5;

System.out.println(data);

} catch (ArithmeticException e) {

System.out.println(e);

} finally {

System.out.println("Finally block is always executed.");

}

}

}

**finalize**

The finalize method is called by the garbage collector before an object is destroyed. It provides an opportunity to perform cleanup operations before the object is removed from memory. However, its use is generally discouraged in favor of other resource management techniques like try-with-resources.

Example:

public class Example {

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize method called.");

}

public static void main(String[] args) {

Example obj = new Example();

obj = null;

System.gc(); // Requesting JVM to call garbage collector

}

}

In summary:

* **final**: Prevents modification (classes, methods, variables).
* **finally**: Ensures a block of code is executed after a try block, regardless of exceptions.
* **finalize**: Called by the garbage collector before object destruction (not commonly used).

In the banking domain, document validation is a crucial process that ensures the authenticity and accuracy of documents submitted by customers. This process typically falls under the broader category of **compliance and risk management** rather than being specifically categorized under cash or trading.

Here are some key areas where document validation is important:

1. **KYC (Know Your Customer)**: This involves verifying the identity of customers to prevent fraud and money laundering. Documents like ID proofs, address proofs, and financial statements are validated.
2. **Loan Processing**: When customers apply for loans, their financial documents, credit history, and other relevant documents are validated to assess their creditworthiness.
3. **Account Opening**: During the account opening process, banks validate the documents submitted by customers to ensure they meet regulatory requirements.
4. **Trading Accounts**: For customers opening trading accounts, banks validate documents to comply with regulatory standards and ensure the legitimacy of the account holder.

Document validation is a part of the bank's efforts to comply with regulatory requirements and manage risks effectively

In Java Selenium, both interfaces and abstract classes are used to achieve abstraction, but they serve different purposes and have distinct characteristics. Here's a comparison and some guidance on when to use each:

**Interface**

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors.

* **Purpose**: To define a contract that implementing classes must follow.
* **Methods**: Can only have abstract methods (until Java 8, which introduced default and static methods).
* **Multiple Inheritance**: A class can implement multiple interfaces, allowing for multiple inheritance.

Example:

public interface WebDriver {

void get(String url);

void quit();

}

public class ChromeDriver implements WebDriver {

@Override

public void get(String url) {

System.out.println("Opening URL in Chrome: " + url);

}

@Override

public void quit() {

System.out.println("Closing Chrome browser.");

}

}

**Abstract Class**

An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can contain both abstract methods (without a body) and concrete methods (with a body).

* **Purpose**: To provide a common base class with shared code and to define methods that must be implemented by subclasses.
* **Methods**: Can have both abstract and concrete methods.
* **Single Inheritance**: A class can extend only one abstract class.

Example:

public abstract class BaseTest {

WebDriver driver;

public BaseTest(WebDriver driver) {

this.driver = driver;

}

public abstract void runTest();

public void openWebsite(String url) {

driver.get(url);

}

}

public class ChromeTest extends BaseTest {

public ChromeTest(WebDriver driver) {

super(driver);

}

@Override

public void runTest() {

openWebsite("https://www.example.com");

System.out.println("Running test in Chrome.");

}

}

**When to Use Each**

* **Use an Interface** when you want to define a contract that multiple classes can implement, especially when those classes are not related through inheritance. Interfaces are ideal for defining capabilities that can be shared across different class hierarchies.
* **Use an Abstract Class** when you have a base class that should not be instantiated on its own but provides common functionality to its subclasses. Abstract classes are useful when you want to share code among related classes while still enforcing certain methods to be implemented by subclasses.

**In Selenium**

In Selenium, interfaces are commonly used to define the behavior of different browser drivers (e.g., WebDriver, TakesScreenshot). Abstract classes can be used to create base test classes that provide common setup and teardown methods, which can be extended by specific test classes.

Do you have a specific scenario in mind where you're deciding between using an interface or an abstract class?

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Uses | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | The keyword throw is used within the method. | The keyword throws is used with the method signature. |
| 5. | Internal Implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

*Rule: For each try block there can be zero or more catch blocks, but only one finally block.*

*Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).*

***List interface:****List interface extends the Collection interface, and it is an ordered collection of objects. It contains duplicate elements. It also allows random access of elements.*

***Set interface:****Set (java.util.Set) interface is a collection which cannot contain duplicate elements. It can only include inherited methods of Collection interface.*

***Queue interface:****Queue (java.util.Queue) interface defines queue data structure, which stores the elements in the form FIFO (first in first out).*

***Dequeue interface:****it is a double-ended-queue. It allows the insertion and removal of elements from both ends. It implants the properties of both Stack and queue so it can perform LIFO (Last in first out) stack and FIFO (first in first out) queue, operations.*

***Map interface:****A Map (java.util.Map) represents a key, value pair storage of elements. Map interface does not implement the Collection interface. It can only contain a unique key but can have duplicate elements. There are two interfaces which implement Map in java that are Map interface and Sorted Map.*