Hierarchy of Java Exception classes

The java. lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

## Difference between Checked and Unchecked Exceptions

### **1) Checked Exception**

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

et's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=100/0;
6. }**catch**(ArithmeticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

### **1) A scenario where ArithmeticException occurs**

If we divide any number by zero, there occurs an ArithmeticException.

1. **int** a=50/0;//ArithmeticException

### **2) A scenario where NullPointerException occurs**

If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

### **3) A scenario where NumberFormatException occurs**

The wrong formatting of any value may occur NumberFormatException. Suppose I have a [string](https://www.javatpoint.com/java-string) variable that has characters, converting this variable into digit will occur NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

### **4) A scenario where ArrayIndexOutOfBoundsException occurs**

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

## Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

### **Syntax of Java try-catch**

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

### **Syntax of try-finally block**

1. **try**{
2. //code that may throw an exception
3. }**finally**{}

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

1. **public** **class** MultipleCatchBlock2 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
8. System.out.println(a[10]);
9. }
10. **catch**(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. **catch**(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBounds Exception occurs");
17. }
18. **catch**(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

# **Java Nested try block**

The try block within a try block is known as nested try block in java.

### **Why use nested try block**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Why use java finally

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

## Usage of Java finally

Let's see the different cases where java finally block can be used.

### **Case 1**

Let's see the java finally example where **exception doesn't occur**.

1. **class** TestFinallyBlock{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. **int** data=25/5;
5. System.out.println(data);
6. }
7. **catch**(NullPointerException e){System.out.println(e);}
8. **finally**{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

# **Java throw exception**

## Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

1. **throw** exception;

Let's see the example of throw IOException.

1. **throw** **new** IOException("sorry device error);

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

1. **public** **class** TestThrow1{
2. **static** **void** validate(**int** age){
3. **if**(age<18)
4. **throw** **new** ArithmeticException("not valid");
5. **else**
6. System.out.println("welcome to vote");
7. }
8. **public** **static** **void** main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

Output:

Exception in thread main java.lang.ArithmeticException:not valid

# **Java Exception propagation**

|  |
| --- |
| An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation. |

#### **Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).**

***Program of Exception Propagation***

1. **class** TestExceptionPropagation1{
2. **void** m(){
3. **int** data=50/0;
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();
15. obj.p();
16. System.out.println("normal flow...");
17. }
18. }

Output:exception handled

normal flow...

In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

# **Java throws keyword**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

### **Syntax of java throws**

1. return\_type method\_name() **throws** exception\_class\_name{
2. //method code
3. }

### **Which exception should be declared**

**Ans)** checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### **Advantage of Java throws keyword**

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

### **Case2: You declare the exception**

* A)In case you declare the exception, if exception does not occur, the code will be executed fine.
* B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

# **Difference between throw and throws in Java**

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

# **Difference between final, finally and finalize**

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

Java final example

1. **class** FinalExample{
2. **public** **static** **void** main(String[] args){
3. **final** **int** x=100;
4. x=200;//Compile Time Error
5. }}

Java finally example

1. **class** FinallyExample{
2. **public** **static** **void** main(String[] args){
3. **try**{
4. **int** x=300;
5. }**catch**(Exception e){System.out.println(e);}
6. **finally**{System.out.println("finally block is executed");}
7. }}

Java finalize example

1. **class** FinalizeExample{
2. **public** **void** finalize(){System.out.println("finalize called");}
3. **public** **static** **void** main(String[] args){
4. FinalizeExample f1=**new** FinalizeExample();
5. FinalizeExample f2=**new** FinalizeExample();
6. f1=**null**;
7. f2=**null**;
8. System.gc();
9. }}

# **ExceptionHandling with MethodOverriding in Java**

|  |
| --- |
| There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |

### **If the superclass method does not declare an exception**

# **Java Custom Exception**

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

1. **class** InvalidAgeException **extends** Exception{
2. InvalidAgeException(String s){
3. **super**(s);
4. }
5. }
6. **class** TestCustomException1{
8. **static** **void** validate(**int** age)**throws** InvalidAgeException{
9. **if**(age<18)
10. **throw** **new** InvalidAgeException("not valid");
11. **else**
12. System.out.println("welcome to vote");
13. }
15. **public** **static** **void** main(String args[]){
16. **try**{
17. validate(13);
18. }**catch**(Exception m){System.out.println("Exception occured: "+m);}
20. System.out.println("rest of the code...");
21. }
22. }

Output:Exception occured: InvalidAgeException:not valid

rest of the code...

JAVA,

Java is a high level programming language

* Object oriented
* Platform independent
* Simple
* Secure
* Architecture neutral
* Portable
* Robust

Applications of Java

* Multithreaded
* Interpreted
* High performance
* Distributed
* Dynamic

Environment Setting

Install JAVA and add the installation Bin path to environment variables in System path

Java Editors

* Notepad
* NEtbeans IDE
* Eclipse IDE

## Basic Syntax

About Java programs, it is very important to keep in mind the following points.

* **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
* **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

**Example:** *class MyFirstJavaClass*

* **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

**Example:** *public void myMethodName()*

* **Program File Name** − Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

**Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

* **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

## Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.

## Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

* **Access Modifiers** − default, public , protected, private
* **Non-access Modifiers** − final, abstract, static

## Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap.

## Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

## Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | assert | boolean | break |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | final | finally | float |
| for | goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |
| short | static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | while |  |  |

## Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

## Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

## Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

### **Rules for creating Java constructor**

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

#### **Note: We can use**[**access modifiers**](https://www.javatpoint.com/access-modifiers)**while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.**

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

## Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

* Byte
* Short
* Int
* Long
* Float
* Double
* Boolean
* Char
* int a, b, c; // Declares three ints, a, b, and c.
* int a = 10, b = 10; // Example of initialization
* byte B = 22; // initializes a byte type variable B.
* double pi = 3.14159; // declares and assigns a value of PI.
* char a = 'a'; // the char variable a iis initialized with value 'a'

## Reference Datatypes

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
* Class objects and various type of array variables come under reference datatype.
* Default value of any reference variable is null.
* A reference variable can be used to refer any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

## Local Variables

* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor, or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are −

* Visible to the package, the default. No modifiers are needed.
* Visible to the class only (private).
* Visible to the world (public).
* Visible to the package and all subclasses (protected).

Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables.
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads.

Loops: (Control Structures)

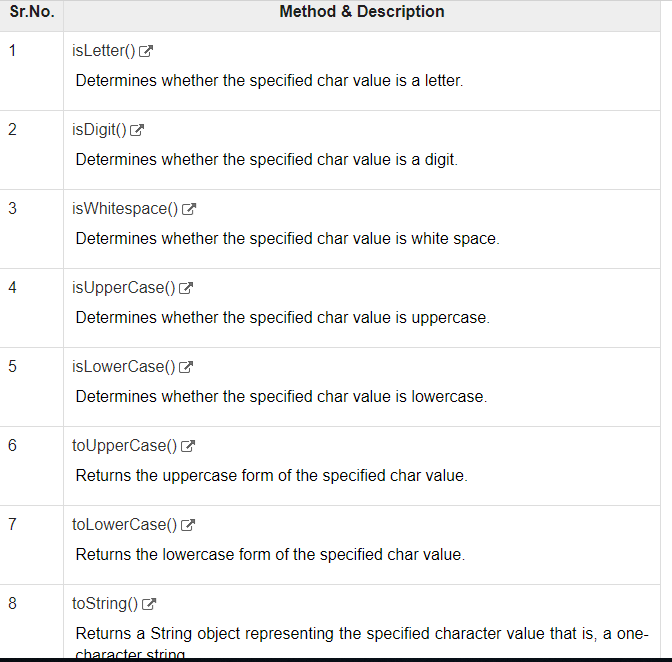
* For loop – Repeats for specific number of iterations
* While – Executes until the condition is satisfied
* Do While – Executes once for sure, then till condition satisfies

Decision making Structures:

* If
* IF Else
* Nested If
* Switch

Number Methods

Character Methods



String Length

String palindrome = "Dot saw I was Tod";

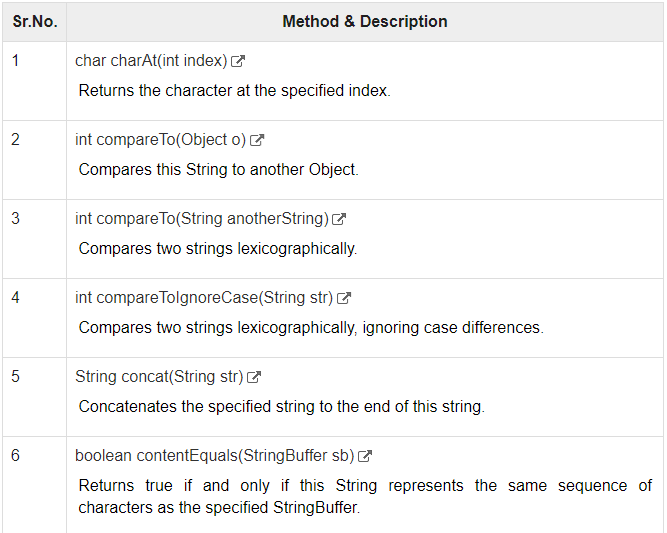
int len = palindrome.length();

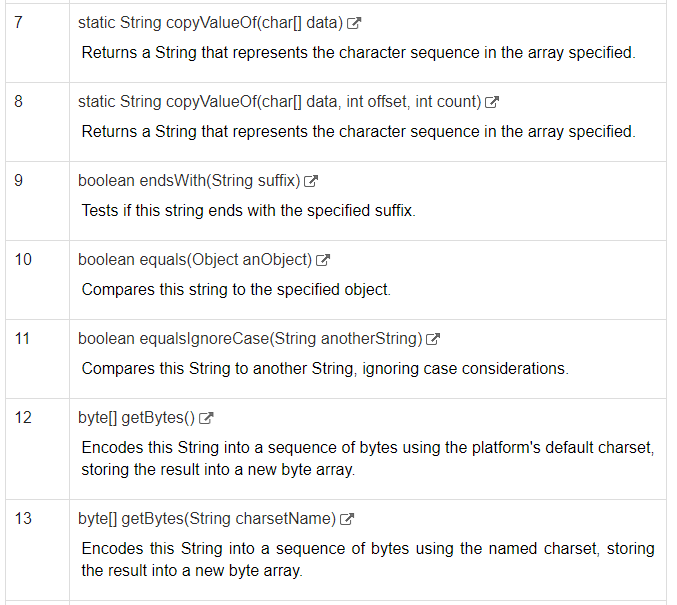
## Concatenating Strings

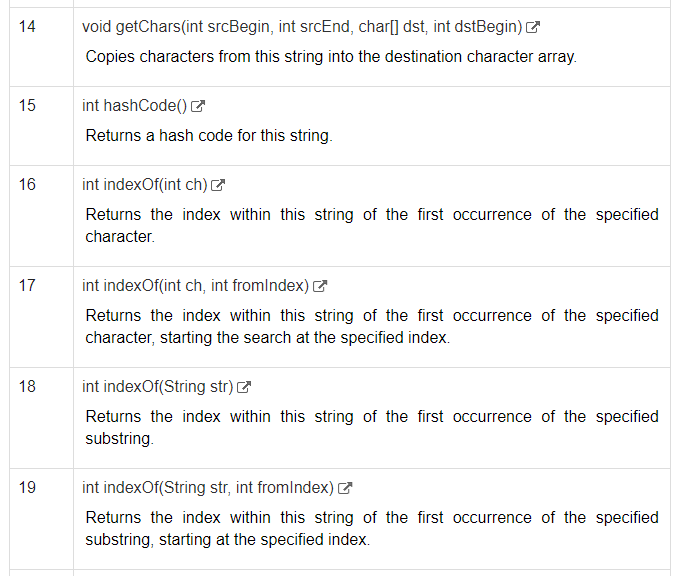
The String class includes a method for concatenating two strings −

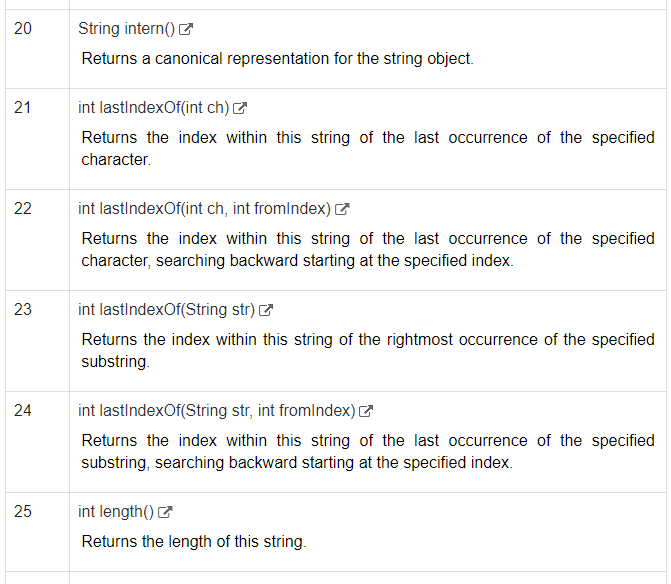
string1.concat(string2);

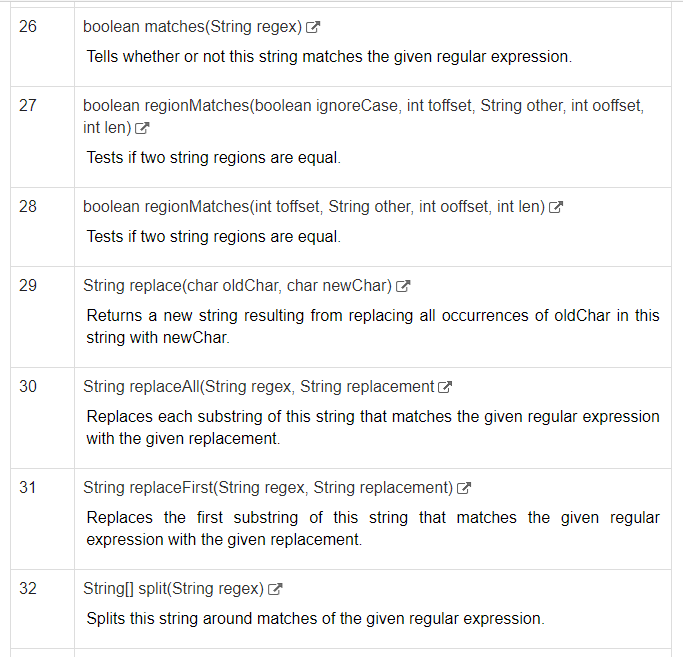
String Methods

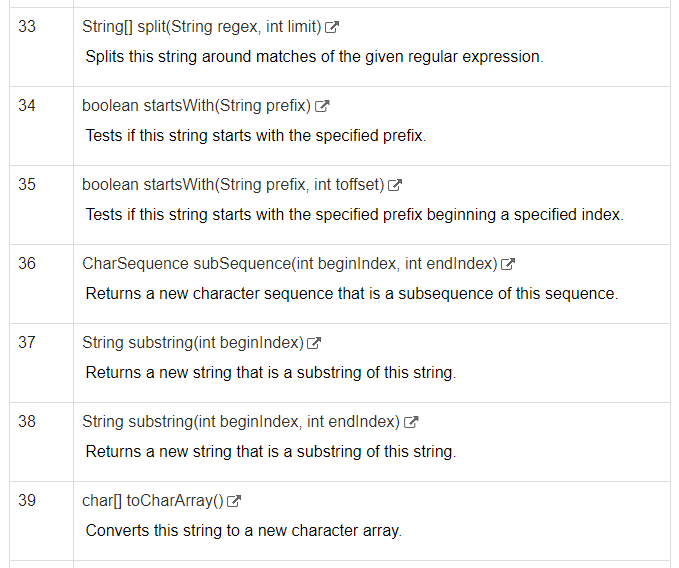


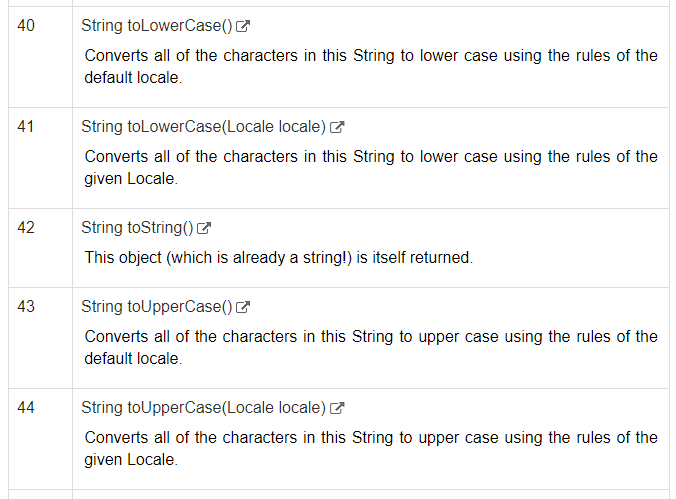


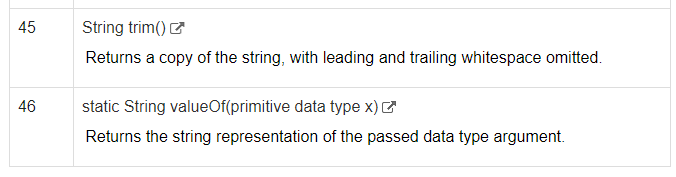












Two strings are **lexicographically** equal if they are the same length and contain the same characters in the same positions. In this case, stringA. compareTo( stringB ) returns 0. Otherwise, stringA. compareTo( stringB ) returns a negative value if StringA comes first and a positive value if StringB comes first.

Arrays

Creating array

double[] myList = new double[10];

Declaring Array

double[] myList; // preferred way.

or

double myList[]; // works but not preferred way.

Date

[Live Demo](http://tpcg.io/62SHhJ)

import java.util.\*;

import java.text.\*;

public class DateDemo {

public static void main(String args[]) {

Date dNow = new Date( );

SimpleDateFormat ft =

new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

System.out.println("Current Date: " + ft.format(dNow));

}

}

Creating Method

* **public static** − modifier
* **int** − return type
* **methodName** − name of the method
* **a, b** − formal parameters
* **int a, int b** − list of parameters

## The this keyword

**this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

**Note** − The keyword *this* is used only within instance methods or constructors

## Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* **InPutStream** − The InputStream is used to read data from a source.
* **OutPutStream** − The OutputStream is used for writing data to a destination.

FileInputStream in = **new** FileInputStream("C:\\Users\\Cyril.s\\Desktop\\input.txt");

FileOutputStream out = **new** FileOutputStream("C:\\Users\\Cyril.s\\Desktop\\Output.txt");

|  |  |
| --- | --- |
| ElementNotVisibleException | This type of Selenium exception occurs when an existing element in DOM has a feature set as hidden. |
| ElementNotSelectableException | This Selenium exception occurs when an element is presented in the DOM, but you can be able to select. Therefore, it is not possible to interact. |
| NoSuchElementException | This Exception occurs if an element could not be found. |
| NoSuchFrameException | This Exception occurs if the frame target to be switched to does not exist. |
| NoAlertPresentException | This Exception occurs when you switch to no presented alert. |
| NoSuchWindowException | This Exception occurs if the window target to be switch does not exist. |
| StaleElementReferenceException | This Selenium exception occurs happens when the web element is detached from the current DOM. |
| SessionNotFoundException | The WebDriver is acting after you quit the browser. |
| TimeoutException | Thrown when there is not enough time for a command to be completed. For Example, the element searched wasn't found in the specified time. |
| WebDriverException | This Exception takes place when the WebDriver is acting right after you close the browser. |
| ConnectionClosedException | This type of Exception takes place when there is a disconnection in the driver. |
| ElementClickInterceptedException | The command may not be completed as the element receiving the events is concealing the element which was requested clicked. |
| ElementNotInteractableException | This Selenium exception is thrown when any element is presented in the DOM. However, it is impossible to interact with such an element. |
| ErrorInResponseException | This happens while interacting with the Firefox extension or the remote driver server. |
| ErrorHandler.UnknownServerException | Exception is used as a placeholder in case if the server returns an error without a stack trace. |
| ImeActivationFailedException | This expectation will occur when IME engine activation has failed. |
| ImeNotAvailableException | It takes place when IME support is unavailable. |
| InsecureCertificateException | Navigation made the user agent to hit a certificate warning. This can cause by an invalid or expired TLS certificate. |
| InvalidArgumentException | It occurs when an argument does not belong to the expected type. |
| InvalidCookieDomainException | This happens when you try to add a cookie under a different domain instead of current URL. |
| InvalidCoordinatesException | This type of Exception matches an interacting operation that is not valid. |
| InvalidElementStateExceptio | It occurs when command can't be finished when the element is invalid. |
| InvalidSessionIdException | This Exception took place when the given session ID is not included in the list of active sessions. It means the session does not exist or is inactive either. |
| InvalidSwitchToTargetException | This occurs when the frame or window target to be switched does not exist. |
| JavascriptException | This issue occurs while executing JavaScript given by the user. |
| JsonException | It occurs when you afford to get the session when the session is not created. |
| NoSuchAttributeException | This kind of Exception occurs when the attribute of an element could not be found. |
| MoveTargetOutOfBoundsException | It takes place if the target provided to the ActionChains move() methodology is not valid. For Example, out of the document. |
| NoSuchContextException | ContextAware does mobile device testing. |
| NoSuchCookieException | This Exception occurs when no cookie matching with the given pathname found for all the associated cookies of the currently browsing document. |
| NotFoundException | This Exception is a subclass of WebDriverException. This will occur when an element on the DOM does not exist. |
| RemoteDriverServerException | This Selenium exception is thrown when the server is not responding because of the problem that the capabilities described are not proper. |
| ScreenshotException | It is not possible to capture a screen. |
| SessionNotCreatedException | It happens when a new session could not be successfully created. |
| UnableToSetCookieException | This occurs if a driver is unable to set a cookie. |
| UnexpectedTagNameException | Happens if a support class did not get a web element as expected. |
| UnhandledAlertException | This expectation occurs when there is an alert, but WebDriver is not able to perform Alert operation. |
| UnexpectedAlertPresentException | It occurs when there is the appearance of an unexpected alert. |
| UnknownMethodException | This Exception happens when the requested command matches with a known URL but and not matching with a methodology for a specific URL. |
| UnreachableBrowserException | This Exception occurs only when the browser is not able to be opened or crashed because of some reason. |
| UnsupportedCommandException | This occurs when remote WebDriver does n't send valid commands as expected. |

***Throwable***: Throwable is a parent class for error and exception. Generally, it is difficult to handle errors in java. If a programmer is not sure about the type of error and exception, then it is advised to use the Throwable class which can catch both error and exception.

**Example**:

|  |
| --- |
| try {     br = new BufferedReader(new FileReader("Data"));       } catch (Throwable t)       {         t.printStackTrace();       } |

## Java Iterator

An Iterator is an object that can be used to loop through collections, like [ArrayList](https://www.w3schools.com/java/java_arraylist.asp) and [HashSet](https://www.w3schools.com/java/java_hashset.asp). It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the java.util package.

ArrayList<String> Bikes = new ArrayList<String>();

Bikes.add("YAMAHA");

Bikes.add("SUZUKI");

Bikes.add("TVS");

// Get the iterator

Iterator<String> it = cars.iterator();

String m = it.next();

System.out.println(m);

// Print the first item

System.out.println(it.next());

## Java HashSet

A HashSet is a collection of items where every item is unique, and it is found in the java.util package:

### **Example**

Create a HashSet object called **cars** that will store strings:

import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();

## Add Items

The HashSet class has many useful methods. For example, to add items to it, use the add() method:

### **Example**

// Import the HashSet class

import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> cars = new HashSet<String>();

cars.add("Volvo");

cars.add("BMW");

cars.add("Ford");

cars.add("BMW");

cars.add("Mazda");

System.out.println(cars);

}

}

**Note:** In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

## Java User Input

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

class Main {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in); // Create a Scanner object

System.out.println("Enter username");

String userName = myObj.nextLine(); // Read user input

System.out.println("Username is: " + userName); // Output user input

}

So, in our example above,**we can suppress the warning associated with our raw type usage**:

**public** **class** **Machine** {

**private** List versions;

@SuppressWarnings("unchecked")

// or

@SuppressWarnings({"unchecked"})

**public** **void** **addVersion**(String version) {

versions.add(version);

}

}

To suppress a list of multiple warnings, we set a *String* array containing the corresponding warning list:

@SuppressWarnings({"unchecked", "deprecated"})