UNIT III

String Handling In Java

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

The most direct way to create a string is to write −

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

Example

Live Demo

```
public class StringDemo {

   public static void main(String args[]) {

      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

      String helloString = new String(helloArray);

      System.out.println( helloString );

   }

}
```

Output

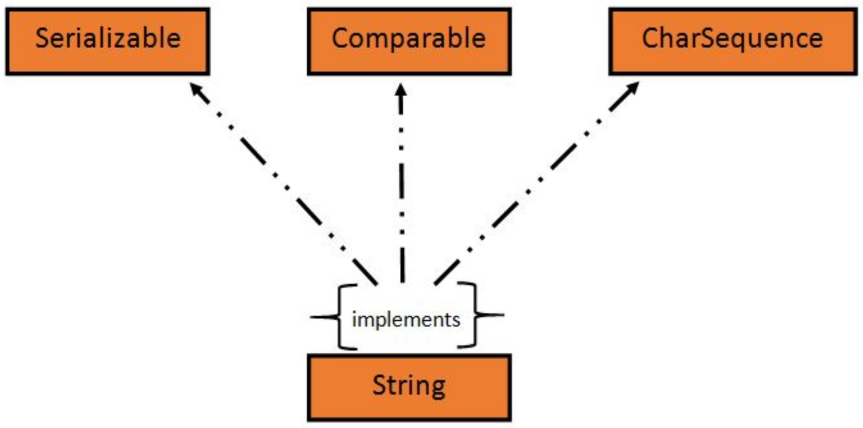
```
hello.
```

Introduction to Java String Handling

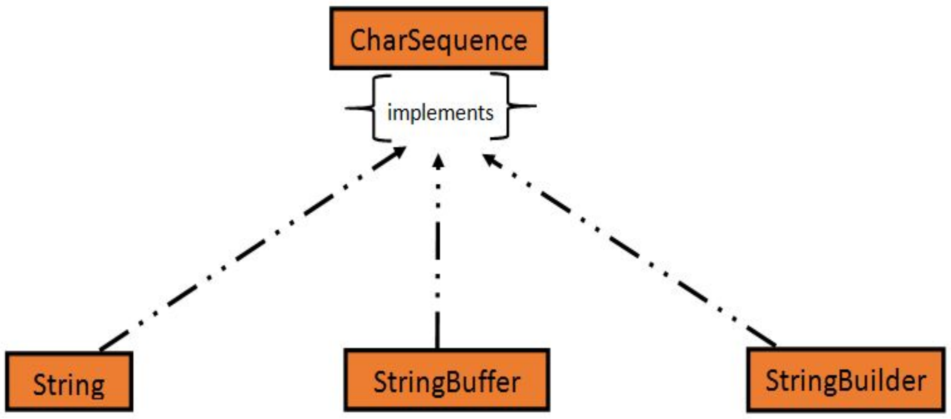
String is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that string objects are **immutable** that means once a string object is created it cannot be changed.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.



In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.



What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

Creating a String object

String can be created in number of ways, here are a few ways of creating string object.

1) Using a String literal

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

```
public class Demo{
   public static void main(String[] args) {
            String s1 = "Hello Java";
            System.out.println(s1);
    }
}
```

```
Hello Java
```

2) Using new Keyword

We can create a new string object by using **new** operator that allocates memory for the object.

```
public class Demo{
    public static void main(String[] args) {
            String s1 = new String("Hello Java");
            System.out.println(s1);
    }
}
```

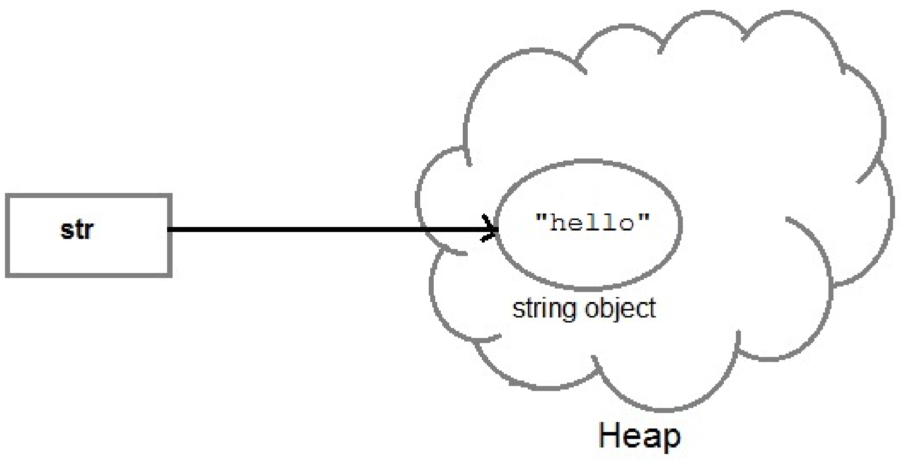
```
Hello Java
```

Each time we create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as **string constant pool** inside the heap memory.

String object and How they are stored

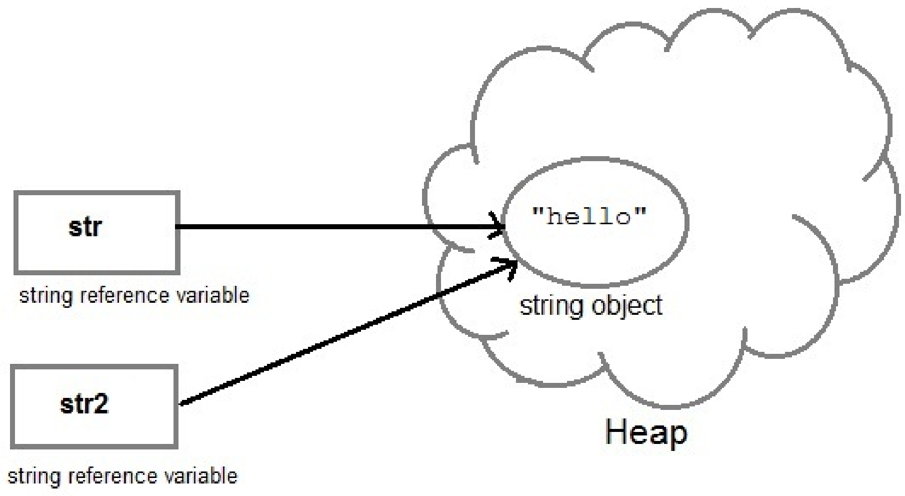
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

```
String str= "Hello";
```



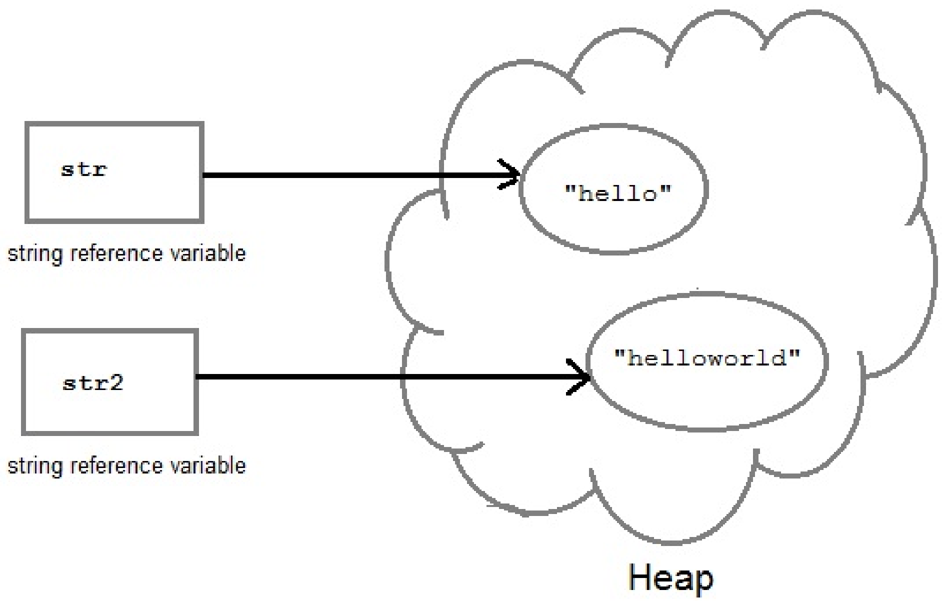
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

```
String str2 = str;
```



But if we change the new string, its reference gets modified.

```
str2=str2.concat("world");
```



Concatenating String

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using + operator

1) Using concat() method

Concat() method is used to add two or more string into a single string object. It is string class method and returns a string object.

```
public class Demo{


   public static void main(String[] args) {

            String s = "Hello";

            String str = "Java";

            String str1 = s.concat(str);

            System.out.println(str1);

   }

}
```

```
HelloJava
```

2) Using + operator

Java uses "+" operator to concatenate two string objects into single one. It can also concatenate numeric value with string object. See the below example.

```
public class Demo{

   public static void main(String[] args) {

            String s = "Hello";

            String str = "Java";

            String str1 = s+str;

            String str2 = "Java"+11;

            System.out.println(str1);

            System.out.println(str2);

   }

}
```

```
HelloJava

Java11
```

String Comparison

To compare string objects, Java provides methods and operators both. So we can compare string in following three ways.

1. Using equals() method
2. Using == operator
3. By CompareTo() method

Using equals() method

equals() method compares two strings for equality. Its general syntax is,

```
boolean equals (Object str)
```

Example

It compares the content of the strings. It will return **true** if string matches, else returns **false**.

```
public class Demo{
   public static void main(String[] args) {
            String s = "Hell";
            String s1 = "Hello";
            String s2 = "Hello";
            boolean b = s1.equals(s2);    //true
            System.out.println(b);
            b =          s.equals(s1) ;   //false
            System.out.println(b);
   }
}
```

```
true
false
```

Using == operator

The double equal (==) operator compares two object references to check whether they refer to same instance. This also, will return **true** on successful match else returns false.

```
public class Demo{
   public static void main(String[] args) {
```

```
            String s1 = "Java";

            String s2 = "Java";

            String s3 = new String ("Java");

            boolean b = (s1 == s2);     //true

            System.out.println(b);

            b =          (s1 == s3);     //false

            System.out.println(b);

   }

}
```

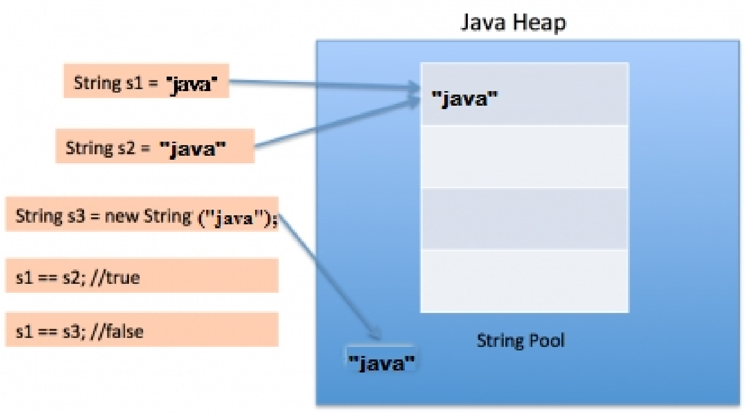
```
true

false
```

Explanation

We are creating a new object using new operator, and thus it gets created in a non-pool memory area of the heap. s1 is pointing to the String in string pool while s3 is pointing to the String in heap and hence, when we compare s1 and s3, the answer is false.

The following image will explain it more clearly.



By compareTo() method

String compareTo() method compares values and returns an integer value which tells if the string compared is less than, equal to or greater than the other string. It compares the String based on natural ordering i.e alphabetically. Its general syntax is.

Syntax:

```
int compareTo(String str)
```

Example:

```
public class HelloWorld{
   public static void main(String[] args) {
            String s1 = "Abhi";
            String s2 = "Viraaj";
            String s3 = "Abhi";
            int a = s1.compareTo(s2);    //return -21 because s1 < s2
            System.out.println(a);
            a = s1.compareTo(s3);    //return 0 because s1 == s3
            System.out.println(a);
            a = s2.compareTo(s1);    //return 21 because s2 > s1
            System.out.println(a);
   }
}
```

```
-21
0
21
```

String Operations in Java

Introduction

Simply put, a `String` is used to store text, i.e. a sequence of characters. Java's most used class is the `String` class, without a doubt, and with such high usage, it's mandatory for Java developers to be thoroughly acquainted with the class and its common operations.

String

There's a lot to say about `String`s, from the ways you can initialize them to the *String Literal Pool*, however in this article we'll focus on common operations, rather than the class itself.

Although, if you'd like to read more about various ways of creating strings in Java you should check out String vs StringBuilder vs StringBuffer.

Here, we're assuming that you're familiar with the fact that `String`s are *immutable*, as it's a very important thing to know before handling them. If not, refer to the previously linked article where it's explained in detail.

The `String` class comes with many helper methods that help us process our textual data:

String Concatenation

Before we begin using any of these methods on strings, we should take a look at String concatenation as it's a fairly common thing to do. Let's start with the `+` operator. The `String` class overloads that operator and it is used to concatenate two strings:

```
String aplusb = "a" + "b";


// The operands can be String object reference variables as well

String a = "a";

String b = "b";

aplusb = a + b;
```

The `+` operator is very slow. `String` objects are immutable, so every time we wish to concatenate *n* strings Java has to copy the characters from all strings into a new `String` object. This gives us quadratic *(O(n^2))* complexity.

This isn't a problem with small strings, or when we're concatenating just several strings at the same time (`String abcd = "a" + "b" + "c" + "d";`). Java automatically uses `StringBuilder` for concatenating several strings at once, so the source of the performance loss is concatenating in loops. Usually, for something like that, we'd use the aforementioned `StringBuilder` class.

It works like a *mutable* `String` object. It bypasses all the copying in string concatenation and gives us linear *(O(n))* complexity.

```
int n = 1000;



// Not a good idea! Gives the right result, but performs poorly.

String result = "";

for (int i = 0; i < n; i++) {

    result += Integer.valueOf(i);

}

// Better, performance-friendly version.

StringBuilder sb = new StringBuilder("");

for (int i = 0; i < n; i++) {

    sb.append(i);

}
```

We can also concatenate using the `concat()` method:

```
String str1 = "Hello";

System.out.println(str1.concat("World"));
```

Output:

```
Hello World
```

String Buffer class

StringBuffer class is used to create a **mutable** string object. It means, it can be changed after it is created. It represents growable and writable character sequence.

It is similar to String class in Java both are used to create string, but stringbuffer object can be changed.

So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously. StringBuffer defines 4 constructors.

1. **StringBuffer**(): It creates an empty string buffer and reserves space for 16 characters.
2. **StringBuffer**(int size): It creates an empty string and takes an integer argument to set capacity of the buffer.
3. **StringBuffer**(String str): It creates a stringbuffer object from the specified string.
4. **StringBuffer**(charSequence []ch): It creates a stringbuffer object from the charsequence array.

Example: Creating a StringBuffer Object

In this example, we are creating string buffer object using StrigBuffer class and also testing its mutability.

```
public class Demo {

	public static void main(String[] args) {

		StringBuffer sb = new StringBuffer("study");
		System.out.println(sb);
		// modifying object
		sb.append("tonight");
		System.out.println(sb);    // Output: studytonight

	}
}
```

Output

```
Study
studytonight
```

StringBuilder

The **StringBuilder** in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters. The function of

StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. However the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does. Therefore this class is designed for use as a drop-in replacement for StringBuffer in places where the StringBuffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations. Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used.

Class Hierarchy:

```
java.lang.Object
 ↳ java.lang
   ↳ Class StringBuilder
```

Syntax:

```
public final class StringBuilder
   extends Object
   implements Serializable, CharSequence
```

Constructors in Java StringBuilder:

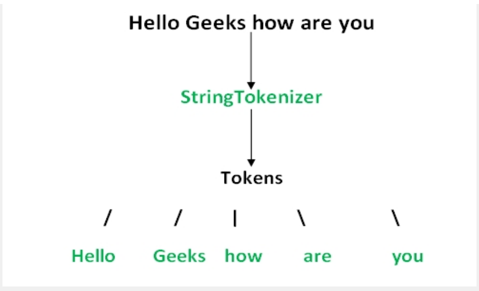
- **StringBuilder():** Constructs a string builder with no characters in it and an initial capacity of 16 characters.
- **StringBuilder(int capacity):** Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
- **StringBuilder(CharSequence seq):** Constructs a string builder that contains the same characters as the specified CharSequence.
- **StringBuilder(String str):** Constructs a string builder initialized to the contents of the specified string.

Below is a sample program to illustrate StringBuilder in Java:

StringTokenizer class in Java

StringTokenizer class in Java is used to break a string into tokens.

Example:



A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

Constructors:

StringTokenizer(String str) :
str is string to be tokenized.
Considers default delimiters like new line, space, tab,
carriage return and form feed.

StringTokenizer(String str, String delim) :
delim is set of delimiters that are used to tokenize
the given string.

StringTokenizer(String str, String delim, boolean flag):
The first two parameters have same meaning. The flag
serves following purpose.

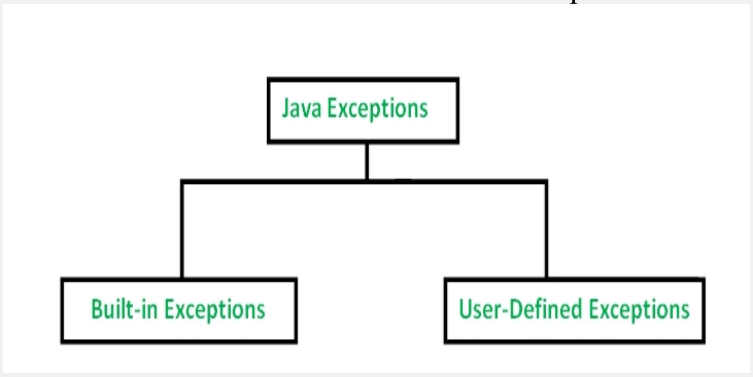
If the **flag** is **false**, delimiter characters serve to
separate tokens. For example, if string is "hello geeks"
and delimiter is " ", then tokens are "hello" and "geeks".

If the **flag** is **true**, delimiter characters are
considered to be tokens. For example, if string is "hello
geeks" and delimiter is " ", then tokens are "hello", " "
and "geeks".

filter_none
edit
play_arrow
brightness_4

Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in Exceptions

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException**
    It is thrown by String class methods to indicate that an index is either negative than the size of the string

Examples of Built-in Exception:

- **Arithmetic exception**

filter_none

edit

play_arrow

brightness_4

```
// Java program to demonstrate ArithmeticException

class ArithmeticException_Demo

{

    public static void main(String args[])

    {

        try {

            int a = 30, b = 0;

            int c = a/b;  // cannot divide by zero

            System.out.println ("Result = " + c);

        }

        catch(ArithmeticException e) {

            System.out.println ("Can't divide a number by 0");

        }

    }

}
```

Output:

```
Can't divide a number by 0
```

Java try-catch block

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

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

```
public class TryCatchExample1 {

    public static void main(String[] args) {

        int data=50/0; //may throw exception

        System.out.println("rest of the code");

    }

}
```

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

```java
public class TryCatchExample2 {

    public static void main(String[] args) {
        try
        {
        int data=50/0; //may throw exception
        }
            //handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        System.out.println("rest of the code");
    }

}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

Try Catch in Java – Exception handling

By Chaitanya Singh | Filed Under: Exception Handling

In the previous tutorial we discussed what is exception handling and why we do it. In this tutorial we will see try-catch block which is used for exception handling.

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try{

  //statements that may cause an exception
}
```

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes

Syntax of try catch in java

```
try
{
   //statements that may cause an exception
}
catch (exception(type) e(object))
{
   //error handling code
}
```

Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).
The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

```
class Example1 {
  public static void main(String args[]) {
    int num1, num2;
    try {
      /* We suspect that this block of statement can throw
       * exception so we handled it by placing these statements
       * inside try and handled the exception in catch block
       */
      num1 = 0;
      num2 = 62 / num1;
      System.out.println(num2);
      System.out.println("Hey I'm at the end of try block");
    }
    catch (ArithmeticException e) {

      System.out.println("You should not divide a number by zero");
    }
    catch (Exception e) {
      System.out.println("Exception occurred");
    }
```

```
     System.out.println("I'm out of try-catch block in Java.");
  }
}
```

Output:

```
You should not divide a number by zero
I'm out of try-catch block in Java.
```

Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples. To read this in detail, see catching multiple exceptions in java.

1. As I mentioned above, a single try block can have any number of catch blocks.
2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: Exception Handling example programs.

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.
4. Corresponding catch blocks execute for that specific type of exception:
catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException
catch(NullPointerException e) is a catch block that can handle NullPointerException
5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: user defined exception, throws keyword, throw vs throws.

Example of Multiple catch blocks

```
class Example2{
   public static void main(String args[]){
     try{
         int a[]=new int[7];
         a[4]=30/0;
         System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
        System.out.println("Warning: ArithmeticException");
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Warning: ArrayIndexOutOfBoundsException");
     }
     catch(Exception e){
        System.out.println("Warning: Some Other exception");
      }
    System.out.println("Out of try-catch block...");
  }
}
```
Output:

```
Warning: ArithmeticException

Out of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

Finally block

I have covered this in a separate tutorial here: java finally block. For now you just need to know that this block executes whether an exception occurs or not. You should place those statements in finally blocks, that must execute whether exception occurs or not.

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.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement);
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

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

```
return_type method_name() throws exception_class_name{
//method code
}
```

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.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```java
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow...
```

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.

Java throw example

```java
void m(){
throw new ArithmeticException("sorry");

}
```

5. **Java throws example**
6.

```
void m()throws ArithmeticException{
//method code
}
```

Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

Difference between throw and throws in java

By Chaitanya Singh | Filed Under: Exception Handling

In this guide, we will discuss the difference between throw and throws keywords. Before going though the difference, refer my previous tutorials about throw and throws.

Throw vs Throws in java

1. **Throws clause** is used to declare an exception, which means it works similar to the try-catch block. On the other hand **throw** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.
For example:

```
throw new ArithmeticException("Arithmetic Exception");
```

and

```
throws ArithmeticException;
```

3. Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.

For example:
Throw:

```
...
void myMethod() {
  try {
    //throwing arithmetic exception using throw
    throw new ArithmeticException("Something went wrong!!");
  }
  catch (Exception exp) {
    System.out.println("Error: "+exp.getMessage());
  }
}
...
```

Throws:

```
...
//Declaring arithmetic exception using throws
void sample() throws ArithmeticException{
  //Statements
}
...
```
4. You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.
For example:
Throw:

```
void myMethod() {
  //Throwing single exception using throw
  throw new ArithmeticException("An integer should not be divided by zero!!");
}
..
```
Throws:

```
//Declaring multiple exceptions using throws
void myMethod() throws ArithmeticException, NullPointerException{
  //Statements where exception might occur
}
```
These were the main **differences between throw and throws in Java**. Lets see complete examples of throw and throws keywords.

Throw Example
To understand this example you should know what is throw keyword and how it works, refer this guide: throw keyword in java.

```
public class Example1{
  void checkAge(int age){
	if(age<18)
	  throw new ArithmeticException("Not Eligible for voting");
	else
	  System.out.println("Eligible for voting");
  }
  public static void main(String args[]){
	Example1 obj = new Example1();
	obj.checkAge(13);
	System.out.println("End Of Program");
  }
}
```
Output:

```
Exception in thread "main" java.lang.ArithmeticException:
Not Eligible for voting
at Example1.checkAge(Example1.java:4)
at Example1.main(Example1.java:10)
```

Throws Example
To understand this example you should know what is throws clause and how it is used in method declaration for exception handling, refer this guide: throws in java.

```
public class Example1{
  int division(int a, int b) throws ArithmeticException{
            int t = a/b;
            return t;
  }
  public static void main(String args[]){
            Example1 obj = new Example1();
            try{
              System.out.println(obj.division(15,0));
            }
            catch(ArithmeticException e){
              System.out.println("You shouldn't divide number by zero");
            }
  }
}
```

Output:

```
You shouldn't divide number by zero
```

If you have any prior experience of Java Interviews, then you might have noticed that the interviewers tend to ask tricky questions usually picked up from the basic concepts. One such question which is mostly asked is to differentiate between final, finally and finalize in Java. Through the medium of this article, I will be drawing a clear line between final, finally and finalize in Java which will help you in gaining better insights.

In this article I will be covering the following topics:

- Final Keyword
- Finally Block
- Finalize Method
- Comparison Table - Final, Finally and Finalize in Java

So, let's get started with the very first keyword among final, finally and finalize in Java.

Final Keyword

In Java, final is a keyword which can also be used as an access modifier. In other words, the final keyword is used to restrict a user's access. It can be used in various contexts like:

1. Final Variable
2. Final Method
3. Final Class

With each of these, the final keyword has a different effect. Let's now see how it affects each of them one by one.

1. Final Variable

Whenever the final keyword in Java is used with a variable, field or parameter it means that once the reference is passed on or the instantiation is done then its value cannot be changed throughout the execution of the program. In case a variable without any value has been declared as final then it is known as blank/uninitialized final variable and can be initialized only through a constructor.

Let’s now see an example.

```
public class A {
int var1 = 123;
//declaring final variables
final int var2 = 345;
final int var3;

//Trying to initialize a blank final variable
var = 555; //Error

A (){
var1 = 111; //No Error
var2 = 333; //Compilation Error

//Initializing a blank final variable
var3 = 444; //No Error
}

//passing final parameters
void avg(int param1, final int param2){
param1 = 2345; //No Error
param2 = 1223; //Compilation Error
}

//declaring final fields
void show(){
final int fieldVal = 300000;
fieldVal = 400000; //Error
}
}
```

So, this was all about how the final keyword affects a variable, let’s now see how a method is affected by it.

2. Final Method

In Java, whenever a method is declared as final it cannot be overridden by any child class throughout the execution of the program.

Let’s see an example.

```
//FINAL METHOD
class A {
final void method_abc(){
System.out.println("This is a Final method and cannot be overridden");
}

void method_xyz(){
System.out.println("This is a normal method and can be overridden");
}
}

class B extends A {
void method_abc{
//Compile Time Error
}

void method_xyz(){
System.out.println("This is an overridden method in class B");
}
}
```

Till now you have already seen the results of declaring a variable and a method final, lets now move further and see what happens when a class is declared as final in Java.

3. Final Class

In Java, whenever a class is declared as final, it cannot be inherited by any subclass. This is because, once a class is declared as final, all the data members and methods contained within the class will be implicitly declared as final. Also, once a class is declared as final it can no longer be declared as abstract. In other words, a class can be either of the two, final or abstract.

Let's see an example.

```
//FINAL CLASS
final class A {
//class body
}

class B extends A{ //Compilation Error
//class body
}
```

I hope by now, you have clearly understood the working of the final keyword. So, let's now move ahead with this article on final, finally and finalize in Java to find out the role of finally keyword.

Finally Block

In Java, finally is an optional block which is used for the Exception Handling. It is generally preceded by a try-catch block. Finally block is used to execute an important code such as resource cleanup or free the memory usage, etc. A finally block will be executed irrespective of the fact whether an exception is handled or not. Thus, wrapping the cleanup codes in a finally block is considered as a good practice. You can also use it with a try block without needing any catch block along with it.

Let's now see an example of the same.

```
class A {
public static void main(String args[]) {
try {
System.out.println("Try Block");
throw new Exception();
} catch (Exception e) {
System.out.println("Catch Block");
} finally {
```

```
System.out.println("Finally Block");
}
}
}
```

Till now, I have already discussed the final and finally keywords in Java. Let's now throw some light on the last keyword among the three that is, finalize keyword in Java.

Finalize Method

Finalize is a protected non-static method that is defined in the Object class and thus is available for any and all the objects in Java. This method is called by the garbage collector before an object is completely destroyed. As sometimes, an object might have to complete some important task like closing an open connection, freeing up a resource, etc before it gets destroyed. If these tasks are not done, it might decrease the efficiency of the program. Thus, the garbage collector calls it for the objects that aren't referenced anymore and have been marked for garbage collection.

This method is declared as protected to restrict its use from outside the class. But you can override it from within the class to define its properties at the time of garbage collection.

Let's see an example of the same.

```
public class A {
public void finalize() throws Throwable{
System.out.println("Object is destroyed by the Garbage Collector");
}
public static void main(String[] args) {

Edureka test = new Edureka();
test = null;
System.gc();
}
}
```

With this, we come to an end of this article on final, finally and finalize in Java. To conclude this, I have added a comparison between all the three keywords which will help you in fetching the major differences at a glance.

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

```java
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}}
```

Java finally example

```java
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}}
```

Java finalize example

```java
class FinalizeExample{
public void finalize(){System.out.println("finalize called");}
public static void main(String[] args){
FinalizeExample f1=new FinalizeExample();
FinalizeExample f2=new FinalizeExample();
f1=null;
f2=null;
System.gc();
}}
```

Method Overriding in Java

1. Understanding the problem without method overriding
2. Can we override the static method
3. Method overloading vs. method overriding

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```java
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

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.

```java
class InvalidAgeException extends Exception{
 InvalidAgeException(String s){
   super(s);
  }
 }
class TestCustomException1{

   static void validate(int age)throws InvalidAgeException{
     if(age<18)
      throw new InvalidAgeException("not valid");
     else
      System.out.println("welcome to vote");
   }

   public static void main(String args[]){
      try{
      validate(13);
      }catch(Exception m){System.out.println("Exception occured: "+m);}

      System.out.println("rest of the code...");
  }
}
```