**How are the exceptions handled in java?**

When an exception occurs, the execution of the program is aborted and handed over to an appropriate exception handler. The **try-catch-finally** block is used to handle the exception.

The code in which the exception may occur is enclosed in a **try block**, also called as **guarded region**.

The **catch** clause matches a specific exception to a block of code which handles that exception.

The **finally** block executes even no exception occurs and usually hold clean up code such are closing database connection, file etc.

**Difference between Error and Exception in Java.**

An error is an irrecoverable condition occurring at runtime, for example, OutOfMemory error. These JVM errors cannot be fixed at runtime. Though an error can be caught in catch block, the execution of application will halt as errors are not recoverable.

An Exception could be handled by using either try-catch block or throwing exception back to caller.

**What are checked and unchecked exceptions?**

In Java, there are two types of exceptions.

**Checked exceptions**: Exceptions that inherit from the Exception class are checked exceptions. Client code must handle the checked exceptions thrown by the API, either in a catch clause or by forwarding it outward with the throws clause. Examples? SQLException, IOxception.

**Unchecked exceptions**: RuntimeException also extends from Exception. However, all of the exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them, and hence they are called unchecked exceptions. Example Unchecked exceptions are NullPointerException, OutOfMemoryError, DivideByZeroException typically, programming errors.

**How do we handle more Than One Type of Exception using catch block in Java?**

Using multiple catch blocks we may specify multiples exception to be handled in Java. For example, the below code handles both IOException and SQLException.

catch (IOException ioEx) {

logger.log(ioEx);

throw ioEx;

catch (SQLException sqlEx) {

logger.log(sqlEx);

throw sqlEx;

}

Although the above code handles multiple exceptions, we could notice that both blocks has same code and handling multiple exception sometimes lead to code duplication. Also, in releases prior to Java 7, it is difficult to create a common method to eliminate the duplicated code because the exception variable has different types.

In Java SE 7 and later, **multiple exception by a single catch block** is introduced. This feature eliminates code duplication and reduce the possiblity to catch an overly broad exception.

The following example, which will execute in Java SE 7 and later, eliminates the duplicate code.

catch (IOException|SQLException exception) {

logger.log(exception);

throw exception;

}

The catch clause specifies the types of exceptions that each block can handle, and each exception type is separated with a pipe delimiter (|).

Note that if a catch block handles more than one exception type, then the catch parameter is implicitly final i.e. the exception variable value cannot be changed within the catch block.

**What happens if an exception is thrown from the finally or catch block in Java?**

The new exception thrown at a catch or finally block will propagate out of that block, resulting the current exception be aborted or ignored as the new exception is propagated outward.

The new exception starts unwinding up the stack as usual just like any other exceptions, stepping out of the current block (the catch or finally block) and subject to any applicable catch or finally blocks along the way.

One of the desired approaches is handling the exception of catch or finally block within the same block itself.

**Will the finally block be executed when the catch clause throws exception in Java?**

Yes. Finally, clause is executed even when an exception is thrown from anywhere in either try or catch block. See the below example.

**package** net.javapedia.exceptions;

**import** **java.io.IOException**;

**public** **class** **FinallyException** {

**public** **static** **void** **main**(String[] args) **throws** IOException {

**try** {

System.out.println("javapedia.net: printed from try block.");

**throw** **new** **Exception**();

} **catch** (Exception e) {

System.out.println("javapedia.net: printed from catch block.");

**throw** **new** **IOException**();

} **finally** {

System.out.println("javapedia.net: printed from Finally clause.");

}

}

**What is a user defined/custom exception in Java?**

User-defined exceptions can be implemented by,

* defining a class to respond to an exception and
* embedding a throw statement in the try clause where the exception can occur or declaring that the method to throw the exception to invoking method where it is handled.

A new exception can be defined by deriving it from the Exception class as follows.

**public** **class** **UserDefinedException** **extends** Exception {

**public** **UserDefinedException**() {

**super**();

}

**public** **UserDefinedException**(String errorMessage) {

**super**(errorMessage);

}

}

The throw statement is used to signal the occurance of the exception within a try block. Often, exceptions are instantiated in the same statement in which they are thrown using the syntax.

**throw** **new** **UserDefinedException** ("throwing User Defined Exception.")

To handle the exception within the method where it is thrown, a catch statement that handles UserDefinedException, must follow the try block. If the developer does not want to handle the exception in the method itself, the method must pass the exception using the syntax: **public** **myMethodName**() **throws** UserDefinedException

**How to create unchecked exception in Java?**

If the custom exception needs to be unchecked, then have the user defined exception class extend RuntimeException.

**public** **class** **UserDefinedException** **extends** RuntimeException {

UserDefinedException(String msg) {

**super**(msg);

}}

**What will happen to the Exception object after exception handling in Java?**

The Exception object will be garbage collected in the next garbage collection process.

How does finally block differ from finalize () method in Java?

A finally block will be executed whether an exception is thrown and is used to release those resources held by the application.

Finalize is a protected method of the Object class, which is called by the Java Virtual Machine (JVM) just before an object is garbage collected.

Does a finally block always run in Java?

If the JVM exits while the try or catch code is being executed, then the finally block may not execute. **System.exit** cause the finally block to not execute.

If the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application continues.

Does return statement allow finally block to execute in Java?

Yes. The return statement executes the finally block.

**public** **class** **FinallyReturn** {

**public** **static** **void** **main**(String[] args) {

System.out.println("Value returned = " + myMethod());

}

**public** **static** **int** **myMethod**() {

**try** {

**return** **0**;

}

**finally** {

**return** **1**;

}

}

}

Output:

Value returned = 1

**Should a catch block always follow try block in Java for Exception handling?**

The try block needs to be followed by either catch or finally block or both. Any exception thrown from a try block needs to be either caught in the catch block or else any specific tasks to be performed are placed in the finally block. The exceptions that are likely to be thrown should be declared in the throws clause of the method.

**Explain finally block in Java.**

Java finally block is used to execute important code such as closing connection, stream and files.

Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.

**What is an Exception in Java?**

Exception is an abnormal condition which occurs during the execution of a program and disrupts normal flow of the program. This exception must be handled properly. If it is not handled, program will be terminated abruptly.

In Java 7, can we catch by grouping exceptions that are hierarchically related?

We can group only un-related exceptions together. It is illegal to group exceptions which has parent-child relationship. For example, it is illegal to write a multi-catch statement like this:

**try** {

...

} **catch** (FileNotFoundException | IOException ex) {

System.err.println("File not found.");

}

FileNotFoundException is a subclass of IOException so this grouping is invalid and the compiler reports a problem that "error: Alternatives in a multi-catch statement cannot be related by subclassing".

What is try-with-resources statement in Java?

The try-with-resources statement is a try statement that declares one or more resources, where resources are objects that must be closed, and try-with-resources statement ensures that each resource is closed at the end of the statement.

This statement is introduced in Java 7.

**Can we have multiple resources inside a try-with-resources in Java?**

Yes. We can use multiple resources inside a try-with-resources block and have them all automatically closed.

**What is the order the resources are closed using try-with-resources?**

The resources will be closed in reverse order of the order in which they are created / listed inside the parentheses in try statement.

Advantages of using try-with-resources in Java.

* Minimize lines of code,
* Automatic resource management,
* No need of finally block just to close the resources,
* Manage multiple resources.

**What happens when the Auto Closeable resource is null in Java Try-With-Resource?**

A resource is closed only if it is initialized to a non-null value. If the reference is null, no attempt is made to call close () on it, no NullPointerException is thrown, and it works as expected.

**Difference between Error and runtime exceptions in Java.**

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The Thread Death error, though a "normal" condition, is also a subclass of Error because most applications should not try to catch it.

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

**Difference between throw and throws clause in Java.**

The **throw** clause explicitly throws as an exception while **throws** clause intimate the compiler that exceptions are being handled and this method might throw.

The throws need to be used in the methods definition and while invoking the method that raises checked exceptions.

Can try block exist without any catch and finally block in Java?

No, the try block must be followed by **either catch or finally block** and only try block results in compile time error.

Will the resources be closed at finally/catch in try with resources?

In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

Order of closing resources in try with resources Java statement.

In case of multiple resources, the close methods of resources are called in the opposite order of their creation.

**What is exception chaining in Java?**

Exception chaining is the technique of handling exceptions by re-throwing a caught exception after wrapping it inside a new exception. This is very helpful in wrapping unchecked exceptions into a checked exception. The entire trace of errors is captured in the stack trace of the exception.

The below constructors of Throwable class support chained exceptions in java:

* Throwable (Throwable cause), cause is the exception that causes the current exception.
* Throwable (String msg, Throwable cause), where msg is the exception message and cause is the exception that causes the current exception.

The below methods support chained exception.

* getCause() method :- This method returns actual cause of an exception.
* initCause(Throwable cause) method :- This method sets the cause for the calling exception.

The following example shows how to use a chained exception.

**try** {

} **catch** (IOException e) {

**throw** **new** **AnotherWrappedSampleException**("Other IOException", e);

}

In this example, when an IOException is caught, a new AnotherWrappedSampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher-level exception handler.

**What is stack trace?**

A stack trace provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred.

A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

**What is the order of catch blocks when catching more than one exception?**

When you are handling multiple catch blocks, make sure that you are specifying exception sub classes first, then followed by exception super classes. Otherwise we will get compile time error.

**Can we have an empty catch block?**

We can have an empty catch block but it’s a bad practice. Never have an empty catch block as if the exception is caught by that block, we will have no information about the exception and it wil be difficult to debug it. There should be at least a logging statement to log the exception details in log files.

**When do you subclass an exception?**

If the exception type is not represented by existing Exception in the Java platform, or if you need to provide more information to client code to treat it in a more precise manner, then you should create a custom exception.

Deciding whether a custom exception should be checked or unchecked depends entirely on the business case. However, as a rule of thumb; if the code using your exception can be expected to recover from it, then create a checked exception otherwise make it unchecked.

Also, you should inherit from the most specific Exception subclass that closely relates to the one you want to throw. If there is no such class, then choose Exception as the parent.

**Give few examples of checked exceptions.**

The checked exceptions are ClassNotFoundException, SQLException, IOException and FileNotFoundException.

Give few examples of unchecked exceptions.

The unchecked exception includes NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException and RuntimeException.

**What is OutOfMemoryError in java?**

OutOfMemoryError is a sub class of java.lang.Error which occurs when JVM runs out of memory.

What is StackOverflowError in java?

StackOverflowError is a subclass of java.lang.Error which is thrown by the JVM when stack overflows.

**Explain exception handling when overriding a method?**

Overriding method cannot throw higher Exception than the overridden method. If the original method throws IOException then the overriding method cannot throw superclass of IOException, I.e., Exception but it can throw any subclass of IOException or does not throw any Exception at all.

This rule only applies to only checked Exception, overridden method is free to throw any unchecked Exception.

Can overridden method throw RuntimeException when original method throws ArithmeticException?

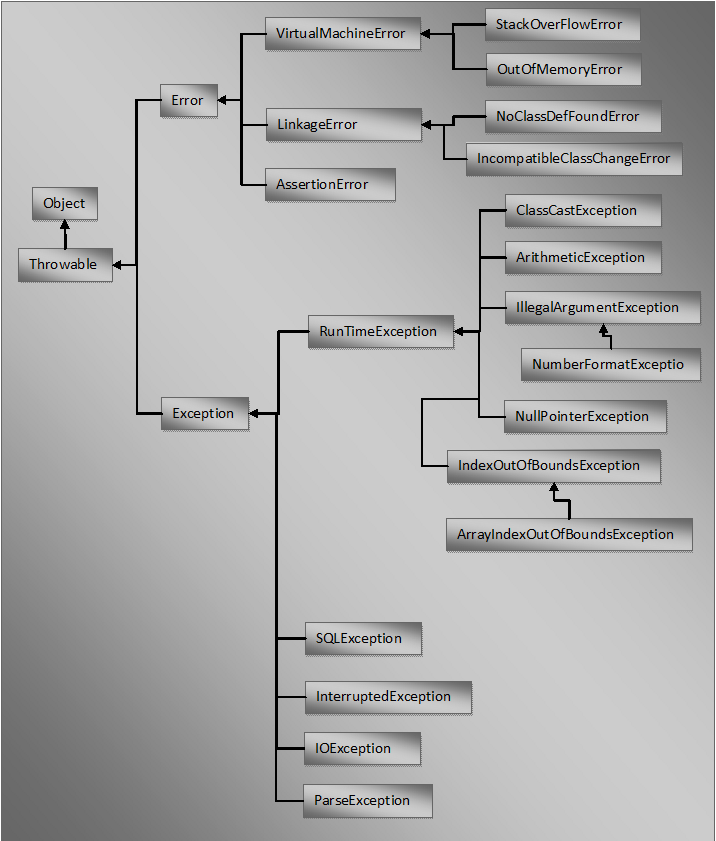
Yes, in case of unchecked exception, overridden method can throw.

**Explain Throwable class.**

The **Throwable class** is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or its subclass) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

A throwable contains a **snapshot of the execution stack** of its thread at the time it was created. It can also contain a **message string** that gives more information about the error. It also can contain a **cause**: another throwable that caused this throwable to get thrown.

**Java Exception Class hierarchy.**



**Difference between final, finally and finalize in Java.**

**final** keyword is used to make a variable or a method or a class as "unchangeable".

**Final variable**: A variable which is declared as final, its value cannot be changed once it is initialized.

**class** **FinalKeywordExample** {

**public** **static** **void** **main**(String[] args) {

**final** **int** x = **10**;

x = **200**;// Compile Time Error

}

}

A **final method** declared as final cannot be overridden or modified in the subclass.

**class** **Parent** {

**final** **void** **parentMethod**() {

}

}

**class** **Child** **extends** Parent {

//Compilation error

**public** **void** **parentMethod**() {

}

}

A **final class** cannot be extended.

**final** **class** **Parent** {

**void** **parentMethod**() {

}}

//Compilation error

**class** **Child** **extends** Parent {

}

**finally,** block is used for exception handling along with try and catch blocks. finally, block is always executed whether an exception is raised or not and raised exception is handled or not. Most of the time, this block is used to close the resources like database connection, I/O resources.

**try** {

// do stuff

} **catch** {

// handle errors

} **finally** {

// clean up connections etc.

}

**finalize** method is a protected method of java.lang.Object class. It is inherited by every class you create in Java. This method is called by garbage collector thread before an object is removed from the memory. finalize () method is used to perform some cleanup operations on an object before it is removed from the memory.

**public** **class** **FinalizeTest** {

**public** **static** **void** **main**(String[] args) **throws** Throwable {

FinalizeTest fTest = **new** FinalizeTest();

FinalizeTest fTest2 = **new** FinalizeTest();

fTest = **null**;

fTest2 = **null**;

System.gc();

System.out.println("Main Thread complete.");

}

**@Override**

**protected** **void** **finalize**() **throws** Throwable {

**super**.finalize();

System.out.println("finalize method overriden");

}

}

**What is rethrowing an exception?**

Exceptions that raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

**try**

{

String s = **null**;

System.out.println(s.equals(“hello”));

//This statement throws NullPointerException

}

**catch** (NullPointerException ex)

{

System.out.println("NullPointerException is caught.");

**throw** ex; //Re-throwing NullPointerException

}

**Explain the rules of Exception Handling in terms of Method Overriding?**

Broadly there are 2 rules.

If superclass method has not declared any exception using throws clause then subclass overridden method cannot declare any checked exception though it can declare unchecked exception.

If superclass method has declared an exception using throws clause then subclass overridden method can do one of the following.

* sub-class can declare the same exception as declared in the super-class method.
* subclass can declare the subtype exception of the exception declared in the superclass method. But subclass method cannot declare any exception that is up in the hierarchy than the exception declared in the super class method.
* subclass method can choose not to declare any exception at all.

Types of OutOfMemoryError in Java.

* Exception in thread thread\_name: java.lang.OutOfMemoryError: **Java heap space**.
* java.lang.OutOfMemoryError: **GC Overhead limit exceeded**.
* java.lang.OutOfMemoryError: **Requested array size exceeds VM limit**.
* java.lang.OutOfMemoryError: **Metaspace.**
* java.lang.OutOfMemoryError: **request size bytes for reason. Out of swap space?**
* java.lang.OutOfMemoryError: **Compressed class space.**
* java.lang.OutOfMemoryError: **reason stack\_trace\_with\_native\_method.**

**Name some of the tools for probing Java Memory Leaks?**

JProbe, JHat and OptimizeIt.

**What does close method throw: Closeable vs AutoCloseable?**

Closeable extends AutoCloseable and both are interfaces. Closeable throws IOException and AutoCloseable throws Exception.

**Strings:**

**Can String be referred as a datatype?**

Yes. However, it is derived datatype (Predefined object) and not the primitive data type like int, char, boolean etc.

**What is toString() method?**

Returns a string representation of an(y) object. The toString() method returns a string that textually represents this object.

The toString method for any Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

getClass().getName() + '@' + Integer.toHexString(hashCode())

**Difference between String, StringBuffer and StringBuilder.**

String object is immutable whereas StringBuffer and StringBuilder objects are mutable.

StringBuffer is synchronized while StringBuilder is not which makes StringBuilder faster than StringBuffer.

**Difference between StringBuffer and StringBuilder.**

All the methods in StringBuffer is synchronized whereas in StringBuilder, it is not synchronized. **StringBuffer is threadsafe while StringBuilder is not**. If you need to manipulate a string in a single thread, use StringBuilder instead.

**How to create our own immutable class in Java?**

* Do not provide any setters.
* Mark all fields as private.
* Make the class final.

**Other examples of Immutable classes.**

* java.lang.StackTraceElement
* java.io.File
* java.util.Locale

**What is Immutable?**

An object is immutable if its state cannot be altered once it is created.

**Is String being Immutable?**

Yes. It is also final.

**How do you create a String object?**

We can create String object using new operator like any other class. Alternatively, you can create by assigning String literals using double quote.

**What is the difference between creating String object using new and String literals?**

When a String object is created using String literals, JVM searches at the String pool to find if any other String object is stored already with the same value. If found, it returns the reference to that String object from String pool otherwise it creates a new String object with given value and stores it in the String pool.

When we use new operator, JVM creates the new String object but don’t store it into the String Pool or try to reuse the existing objects with same value.

**String Interning**

The method of storing only one copy of each distinct string value, which must be immutable. The distinct values are stored in a string intern pool. string intern pool allows a runtime to save memory by preserving immutable strings in a pool so that areas of the application can reuse instances of common strings instead of creating multiple instances of it.

**Java String Pool**

The string pool is the JVM's implementation of the string interning. String Pool is a pool of Strings stored in Java Heap Memory.

**Flyweight Pattern**

Flyweight is a Structural design pattern that helps minimizes memory use by sharing as much data as possible with other similar objects String interning is an example of flyweight design pattern.

**Immutable objects are thread-safe?**

Yes. Because immutable objects cannot be changed.

**Why String is often used as key in HashMap?**

Immutability nature of String objects prevent hash collision is HashMap.

**Name the interfaces that Java String class implements**.

String implements Serializable and Comparable interfaces.

**How do I compare strings in Java?**

== tests for reference equality to find whether they are the same object. equals () tests for value equality to find whether they are logically "equal".

**Where are the string created with the toString() method in memory?**

It is based on jvm implementation per object. Usually toString method creates object in heap.

Integer.toString() method creates in heap but not for all toString() methods. Boolean.toString(), for example, returns strings from the string pool.

**Which design pattern is based on object cloning?**

Prototype design pattern.

**What is the default implementation of equals method in Object class?**

The equals () method provided in the Object class uses the identity operator (==) to determine whether two objects are equal.

**How do I iterate through every character in a String? (or) How to parse each character in a String object?**

The convenient and simple way is to create a for-each statement that runs through every character at the String object after the character array is generated.

String myStr = "HelloWorld!";

**char** myCharArray[] = myStr.toCharArray(); // convert the String to char array

// iterate over the array using the for-each loop.

**for** (**char** c : myCharArray) {

System.out.println(c);

}

**toCharArray() Method**

converts the string to a new character array.

**public** **char**[] **toCharArray**()

**public** **class** **CharArrayExample** {

**public** **static** **void** **main**(String args[]) {

String myString = **new** String("JavaPedia.net");

**char**[] myCharArray = myString.toCharArray();

System.out.println("Length of the Array=" + myCharArray.length);

System.out.println(("Element at the first index= " + myCharArray[**0**]));

}

}

**Output:**

Length of the Array=13

Element at the first index= J

**How do I convert a char (primitive data type variable) to string in Java?**

The below are the recommended ways to implement the conversion.

Using String.valueOf() method,

String myString = String.valueOf('c');

Using Character.toString(char) method,

String myString = Character.toString('c');

**Can we use String in switch case control statement?**

**Java 7** featured the capability of switch case to use Strings, earlier java versions haven't supported it.

**Example**

**package** org.javatutorials.conditional;

**public** **class** **SwitchCaseUsingString** {

**public** **static** **void** **main**(String[] args) {

String stringObj = "ONE";

**switch** (stringObj) {

**case** "ONE":

System.out.println("1 is entered"); **break**;

**case** "TWO":

System.out.println("2 is entered"); **break**;

**case** "THREE":

System.out.println("3 is entered"); **break**;

**default**:

System.out.println("Something else is entered");

}

}

}

**How do I split a pipe delimited Java String using split method?**

The pipe (|) symbol need to be escaped using \\ to treat it as a normal character since split method uses regular expression and in regex | (pipe) is a meta character representing OR operator.

String pipeDelimitedStr = "Tag1|Tag2|Tag3";

String [] tags = pipeDelimitedStr.split("\\|");

for (String tag: tags) {

System.out.println(tag);

}

**How do I create a Regular expression pattern into non-greedy in Java?**

The non-greedy regular expression constructs are like the greedy modifier with a question mark (?) immediately following the meta character.

For example, \* (asterisk) meta character that matches zero or more is a greedy pattern while \* (asterisk) followed with question mark \*? matches same zero or more characters however it is non-greedy.

See below are the examples of greedy construct and its equivalent non-greedy regular expression constructs in Java.

|  |  |  |
| --- | --- | --- |
| Construct | greedy | non-greedy |
| zero or more | \* | \*? |
| one or more | + | +? |
| zero or one | ? | ?? |

**Explain SimpleDateFormat class and its usage in Java.**

java.text.SimpleDateFormat concrete class is widely used in Java for parsing (text to date) and formatting of dates (date to text).

SimpleDateFormat enables choosing user defined patterns for date-time formatting. Also the date-time formatter initialized with a default format pattern could be created using either getTimeInstance, getDateInstance or getDateTime Instance methods in DateFormat class.

package com.javatutorials.date;

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

public class SimpleDateFmtExample {

public static void main(String[] args) {

String pattern = "MM/dd/yyyy";

SimpleDateFormat format = new SimpleDateFormat(pattern);

try {

Date date = format.parse("06/21/2016");

System.out.println(date);

DateFormat dateFmt = DateFormat.getDateInstance();

System.out.println(dateFmt.format(date));

} catch (ParseException e) {

e.printStackTrace();

}

// formatting

System.out.println(format.format(new Date()));

}

}

Note that the SimpleDateFormat objects are not synchronized. It is always a good practice to create separate format instances for each thread or use ThreadLocal class.

**How do I remove duplicate elements from a String Array in Java?**

Using Java Collection framework API, the duplicate elements in a string array can be removed easily by converting the string array to List, creating a HashSet from the List that removes the duplicates and converting set to String array back. See the example below.

String [] data = {"a", "c", "b", "d", "a", "c", "b", "d"};

System.out.println("Actual array: " + Arrays.toString(data));

List<String> strList = Arrays.asList(data);

Set<String> strSet = **new** HashSet<String>(strList);

String [] result = **new** String [strSet.size()];

strSet.toArray(result);

System.out.println("Array without duplicates: ");

**for** (String s: result) {

System.out.print(s + ", ");

}

The above program can be shortened further as shown below.

String [] data = {"a", "c", "b", "d", "a", "c", "b", "d"};

data = **new** HashSet<String>(Arrays.asList(data)). toArray(**new** String[**0**]);

**for** (String s: data) {

System.out.print(s + ", ");

}

**What is the maximum length of a String in Java?**

The maximum length of a String is Integer.MAX\_VALUE, which is 2^31 - 1. The return type of String length () method is int and an int could have only Integer.MAX\_VALUE as its largest value.

Also Integer.MAX\_VALUE is the maximum value that could be allocated for an array in Java.

How many String objects be created by the snippet and where it is stored in Java?

String str1 = "javapedia.net";

String str2 = "javapedia.net";

Only one string object will be created, and this object be stored in the string (constant) pool.

How do you convert a string object to character array in Java?

Using toCharArray () method a string object can be converted to a character array in Java.

**Is string a wrapper class in Java?**

No, String is a class, but not a wrapper class. Wrapper classes like Integer exists for each primitive type and they can be used to convert a primitive data value into an object or vice-versa.

**Can we compare String and String Buffer objects in Java?**

Although both String and String Buffer represent String objects, we cannot compare each other and if we try to compare them, we get an exception.

**Which class is final: String, String Buffer and String Builder?**

All the three classes are final.

**What is the initial capacity of a String Builder object in Java?**

It is the length of the initial string plus 16. For example, new StringBuilder ("javapedia"), the length of the string is 9 + 16 = 25, the initial capacity of the object.

**what happens if begin Index is equal to length of the String when calling substring (int beginIndex)?**

substring returns an empty string and does not throw IndexOutOfBoundException.

**Where String pool is located?**

Before Java 7, the JVM placed the Java String Pool in the PermGen space, which has a fixed size — it cannot be expanded at runtime and is not eligible for garbage collection.

From Java 7 onwards, the Java String Pool is stored in the heap space, which is garbage collected by the JVM. The advantage of this approach is the reduced risk of OutOfMemory error because unreferenced Strings will be removed from the pool, thereby releasing memory.

**What is the difference between appending a string to a String Builder and concatenating two strings with a + operator?**

String is immutable while String Builder is not.

When concatenating two String instances, a new object is created, and strings are copied. This may cause a huge garbage collector overhead if we need to create or modify a string in a loop. String Builder allows handling string manipulations much more efficiently.

**How string concatenation operator (+) works in Java?**

When 2 string objects are concatenated, internally Java creates a new StringBuilder object and appends both string objects and returns a new concatenated string by calling toString method on StringBuilder object.

For example, String website="javapedia"+".net"; will translate as below.

new StringBuilder("javapedia").append(".net").toString();

**When the concatenation of final string literal happens?**

JVM concatenates the final string literals during **compile time**.

For example, final String website = "javapedia." + "net";, the concatenation is handled by JVM during compile time itself.

**How do I initialize a String array?**

* 1. String myFruits[];

myFruits = **new** String [] {"Apple", "Pineapple", "Strawberry"};

* 1. String myFruits [] = **new** String [] {"Apple", "Pineapple", "Strawberry”};
  2. String [] myFruits = {"Apple", "Pineapple", "Strawberry"};

when passing as an argument to a method,

methodName(**new** String[] { "Apple", "Pineapple", "Strawberry" });

**Is arrays are considered as primitive data types?**

No, Arrays are objects in Java.

**How do I create a list from Array which is completely independent of the original array?**

The fixed length list returned by asList method is used as an argument to create a new ArrayList which is later garbage collected.

The new ArrayList is independent of the original array and the changes to the list does not reflect on the array and vice versa.

Adding/removing elements could be performed in this list.

String [] myFruits1 = { "Apple", "Pineapple", "Strawberry" };

List<String> myFruitsList = **new** ArrayList<String>(Arrays.asList(myFruits1));

Since Java 5, we have inbuilt API **Arrays.toString(arr)** and **Arrays.deepToString(arr)** to print the contents of the single dimensional array.

The difference between Arrays.toString and Arrays.deepToString is that Arrays.toString accepts primitive type arrays as well as object arrays where as deepToString accepts only object arrays.

The below example shows how to use the Arrays API to print the content of an array.

**int**[] intArray = **new** **int**[] { **1**, **2**, **3**, **4**, **5** };

System.out.println(Arrays.toString(intArray));

Byte[] byteArray = **new** Byte[] { **1**, **2**, **3**, **4**, **5** };

System.out.println(Arrays.deepToString(byteArray));

output:

[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5]

**How do you print the content of a multi-dimensional array in Java?**

java.util.Arrays.deepToString(Object[]) method returns a string representation of the content of the single (or) multi dimensioned array.

The below example illustrates how the deepToString method could be used to print the content of a multi-dimensional array.

// initializing an object array

Object[][] ob={{"Welcome "," to "}, {"javapedia", ".net"}};

System.out.println("The string content of array is:");

System.out.println(Arrays.deepToString(ob));

**Why is it a good practice to store sensitive information like password, SSN into a character Array rather than String?**

The String objects are immutable and are stored in String pool in memory until garbage collected. So Although a string object is processed and no longer required, for an indeterminate period of time the string object remains in the memory until garbage collected. Even this can not be controlled programmatically. By accessing the memory dump, the hackers could extract sensitive information from the string object hence String is insecure.

Character Array is a mutable object, and when it is no longer required, nullifying the reference guarantees that the object in memory cannot be accessed until garbage collected. Hence character array is preferred for storing sensitive information.

**What is the tradeoff between using an unordered array versus an ordered array?**

The major advantage of an ordered array is that the search times have time complexity of **O (log n),** compared to that of an unordered array, which is **O (n).**

The disadvantage of an ordered array is that the insertion operation has a time complexity of O(n) because the elements with higher values must be moved to make room for the new element. Instead, the insertion operation for an unordered array takes constant time of O (1).

**Difference between Arrays.sort and Arrays.parallelSort in Java 8.**

Arrays.parallelSort is introduced in Java 1.8 and uses multi-threading.

**Enum:**

**Why do we need enum?**

Enum is nothing but the lists of constants, so it is useful when required to define a list of constants.

**Define "enum".**

A Java Enum, introduced in Java 5, is a special kind of Java type used to define collections of constants. It is a special class type that holds constants, methods.

**public** **enum** DayOfTheWeek {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

**Can an enum be declared final?**

No. Compiler generates sub classes for each constants contained in the enum.

**Can an enum be declared abstract?**

No.

**Can an enum implement an interface?**

Yes.

interface IJob {

void jobRole();

}

**public** enum Job implements IJob {

JOB1(**50**) {

@Override

void whoAmI() {

**System**.**out**.println("I am a Software Tester.");

}

@Override

**public** void jobRole() {

**System**.**out**.println("My role is to Test the Software applications to assess quality.");

}

};

int payPerHour;

Job(int payPerHour) {

this.payPerHour = payPerHour;

}

abstract void whoAmI();

}

**Interface and Abstract Class**

**What is a Marker(Tag) Interface?**

An Interface which doesn't have any declaration inside (no methods) but still enforces a mechanism.

Marker interfaces are also referred as Null interfaces.

**Advantages of Inner Interface.**

* promotes Encapsulation.
* Organize interfaces and facilitate logical grouping of related interfaces.
* improves readability

**What are the access modifiers applicable for the nested interface declared inside a class?**

Inner interface (aka) nested interface can have any access modifier when declared inside a class.

Example of a nested interface from Java API.

java.util.Map has inner interface Entry.

**public** **static** **interface** **Map**.Entry<K,V>

**Inner Interface in Java.**

Inner interface also known as nested interface, refer to the interfaces that are declared inside another interface or class.

**Are interfaces also inherited from Object class?**

No, only classes in java are inherited from Object class. Interfaces in java are not inherited from Object class. But, classes which implement interfaces are inherited from Object class.

An interface implicitly declares one method for each public method in Object. This way the equals and Object class method is implicitly declared as a member in an interface.

**Can an Interface implement another interface?**

No. Interface doesn't provide implementation hence it is not possible.

However, an Interface can extend (inherit) another interface, also multiple interface

**public** **interface** **MyMainInterface** **extends** ParentA, ParentB {

}

**interface** **ParentA** {}

**interface** **ParentB** {}

**Can an interface inherit/extend multiple interface?**

Yes.

**package** com.tutorials.interfaceExample;

**public** **interface** **MyInterface** **extends** ParentInterface1, ParentInterface2 {

**void** **method**();

}

**class** **ImplementingClass** **implements** MyInterface {

**@Override**

**public** **void** **method**() {

System.out.println("HI");

}

}

**interface** **ParentInterface1** {

**void** **method**();

}

**interface** **ParentInterface2** {

**void** **method**();

}

**Can a class be defined inside an Interface?**

Yes.

**package** com.javatutorials.accessModifer.protectedPackage;

**public** **interface** **MyMainInterface** {

**public** **class** **MyClass** {

**public** **static** **void** **method**() {

System.out.println("Class method inside an interface.");

}

**void** **nonStaticmethod**() {

System.out.println("Instance method inside an interface.");

}

}

}

**class** **Mainclass** {

**public** **static** **void** **main**(String[] args) {

MyMainInterface.MyClass.method();

MyMainInterface.MyClass myClass = **new** MyMainInterface.MyClass();

myClass.nonStaticmethod();

}

}

**Syntax for inner interface.**

An interface inside an interface.

**interface** **outer\_interface\_name**{

**interface** **inner\_interface\_name**{ //static and public implicitly

}

}

An interface inside a class.

**class** **outer\_class\_name**{

**interface** **nested\_interface\_name**{ //implicitly static

}

}

**Advantage of having a class inside an interface.**

It limits the scope of the class to where it belongs. Class inside an interface is tightly coupled with the interface.

**What is the difference between an Abstract class and Interface?**

* Abstract classes may have some executable methods and methods left unimplemented. Interfaces contain no implementation code.
* A class can implement any number of interfaces, but subclass at most one abstract class.
* An abstract class can have non-abstract methods. All methods of an interface are abstract.
* An abstract class can have instance variables. An interface cannot.
* An abstract class can define constructor. An interface cannot.
* An abstract class can have any visibility: public, protected, private or none (package). An interface’s visibility must be public or none (package).
* An abstract class inherits from Object and includes methods such as clone () and equals ().

**Can a method inside a Interface be declared as final or static in Java?**

No. An interface method can have only public and abstract modifiers.

**Why interface can extend multiple interface whereas class can inherit only one class?**

Java doesn't allow multiple inheritance, so a class can extend only one Class. But an interface is a pure abstraction model and doesn't have inheritance hierarchy like classes.

**Give few examples of a marker interface?**

Some of the marker interfaces are **Serializable, Remote, Cloneable.**

**What is Externalizable?**

Externalizable is an Interface that extends Serializable Interface. It sends data into Streams in Compressed Format.

It has two methods, writeExternal(ObjectOuput out) and readExternal(ObjectInput in)

**Difference between Serializable and Externalizable.**

Serializable is a marker interface as it has defined no members while Externalizable is not as it has two methods.

**Does Externalizable extends Serializable interface?**

Yes.

**What is functional interface in Java 8.**

An interface with **only one abstract method** is known as functional interface. For example, Runnable and Callable interface are functional interface as it has only one method.

A functional interface can have default and static methods while it has only one abstract method.

**What are the rules for writing lambda expressions in Java 8?**

* A lambda expression can have **0, 1 or more parameters**.
* The **parameter's type** can be explicitly declared, or it can be inferred from the context implicitly.
* Multiple parameters are **enclosed in parentheses** and delimited by commas. Empty parentheses represent an empty set of parameters.
* With single parameter, if its type is inferred, it is not required to use parentheses.
* The lambda expression body can contain **0, 1 or more statements**.
* If the body of lambda expression has single statement, **curly parenthesis is optional,** and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

**What is SAM interface in Java 8?**

Functional Interface is also known as SAM Interface, it contains only one abstract method. SAM Interface stands for **Single Abstract Method** Interface.

**How will you call a default method of an interface in a class?**

Using super keyword along with interface name.

**interface** **A** {

**default** **void** **foo**() {

System.out.println("A.foo");

}

}

**public** **class** **B** **implements** A {

**@Override**

**public** **void** **foo**() {

System.out.println("B.foo");

}

**void** **aFoo**() {

A.super.foo();

}

**public** **static** **void** **main**(String[] args) {

B b = **new** B();

b.foo();

b.aFoo();

}

}

**What is Optional in Java8?**

Java 8 introducd Optional in java.util package that represents if a value is present or absent. The main advantage is No more too many null checks and NullPointerException. It avoids any runtime NullPointerExceptions and supports us in developing clean and neat Java APIs or Applications.

**Can we initialize Optional value as null in Java8?**

Yes, but it’s a bad practice. The main intention of Optional is to eliminate null referencing.

**Difference between Optional flatMap() and map() in Java8.**

Use map if the function returns the object you need or flatMap if the function returns an Optional.

**Can interface have instance fields?**

No. All the fields are implicitly public, static and final.

**Can interface have instance method?**

Yes, interface can have default methods from Java8 onwards.

**How JVM invoke special behavior in case of marker interface?**

Let consider Serializable interface as an example. Serialization is handled by the ObjectInputStream and ObjectOutputStream classes.

These classes will check your class whether it implements Serializable, Externalizable. If yes it will continue or else will throw NonSerializableException.

**How to write your own marker interface?**

We can create our own marker interface by creating an interface with no method. This marker interface has nothing to do with JVM, we add the special logic in market interface handler class by using instanceOf check.

**interface** **IMarker**{ }

**class** **MarkerImpl** **implements** IMarker{

//do some task

}

**class** **Main**{

**public** **static** **void** **main**(String[] args){

MarkerImpl ob = **new** MarkerImpl();

**if** (ob instanceOf IMarker){

// do some task

}

}

}

**Concrete class.**

Refers to the class with complete implementation i.e. no abstract methods and only concrete methods.

**Can I create an abstract variable?**

No, abstract keyword is applicable only for classes and methods.

**What is an abstract class?**

A class that has only partial implementation and has abstract methods.

**Can an abstract class be instantiated?**

No. Abstract classes cannot be instantiated since it is abstract and not concrete.

**What is the difference between an Interface and an Abstract class?**

* All methods in an interface are implicitly abstract. On the other hand, an abstract class may contain both abstract and non-abstract methods.
* A class may implement several Interfaces but can extend only one abstract class.
* For a class to implement an interface, it must implement all its declared methods. However, a class may not implement all declared methods of an abstract class. Though, in this case, the sub-class must also be declared as abstract.
* Abstract classes can implement interfaces without even providing the implementation of interface methods.
* Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
* Members of a Java interface are public by default. A member of an abstract class can either be private, protected or public.
* An interface is abstract and cannot be instantiated. An abstract class also cannot be instantiated but can be invoked if it contains a main method.

**Can an abstract method be declared static in Java?**

No. An abstract method cannot be static. You cannot override a static method, so making it abstract would not make any sense.

**Java 8, difference between abstract class and Interface.**

Both abstract class and interface are used for abstraction by hiding the internal implementation of the feature and only showing the functionality to the users.

**method types**: Interface can have only abstract methods. From Java 8, interface can have default and static methods also. Abstract class can have abstract and non-abstract methods.

**Final Variables**: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.

**Type of variables**: Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.

**Accessibility of Data Members**: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

**Implementation**: Abstract class can provide the implementation of interface while Interface cannot provide the implementation of abstract class.

**Inheritance vs Abstraction**: A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".

**Multiple implementation**: An interface can extend another Java interface only; an abstract class can extend another Java class and implement multiple Java interfaces.

**Can abstract class have Constructor?**

Yes, but its object cannot be created by calling the constructor instead it is invoked during constructor chaining.

**Can abstract class implement an interface in Java?**

Yes, an abstract class can implement the interface by using implements keyword. Since they are abstract, they **don’t need to implement all methods**.

**What is an abstract method in Java?**

An abstract method is a method without a body. You just declare method, without defining it and use abstract keyword in method declaration. All method declared inside Java Interface are by default abstract. Here is an example of an abstract method in Java.

**public** **void** **abstract** **anExampleAbstractMethod**();

**Can an abstract class have the main method?**

Yes, it is also a static method and it can execute the abstract class.

**public** **abstract** **class** **AbstractClassMain** {

**public** **static** **void** **main**(String[] args) {

System.out.println("Hello from abstract class");

}

}

**Can an abstract class have a final concrete method?**

Yes. An abstract class can have the final concrete method. See the below example.

**public** **class** **AbstractClassFinalConcreteMethod** {

**public** **static** **void** **main**(String[] args) {

Parent p = **new** Child();

p.parentMethod();

}

}

**abstract** **class** **Parent** {

**final** **void** **parentMethod**() {

System.out.println("Final Method in a abstract class");

}

}

**class** **Child** **extends** Parent {

}

Output:

Final Method in an abstract class

In the above example, the parent abstract class has a final concrete method 'parentMethod' which is allowed and called directly from the parent reference.