**Exception Handling**

Exception handling in java is one of the powerful mechanisms to handle runtime errors caused by exceptions. Exception handling plays an important role in software development.

Exception handling in java helps in minimizing exceptions and helps in recovering from exceptions. It is one of the powerful mechanisms to handle runtime exceptions and makes it bug-free. Exception handling helps in maintaining the flow of the program.

Definition:

An exception handling is defined as an abnormal condition that may happen at runtime and disturb the normal flow of the program.

**For handling exceptions, there are 2 possible approaches**

### **1. JVM**

If an exception is not handled explicitly, then JVM takes the responsibility of handling the exception.

Once the exception is handled, JVM will halt the program and no more execution of code will take place

**Example**:

import java.util.\*;

class Main {

public static void main (String[] args) {

System.out.println(5/0);

System.out.println("End of program!");

}

}

**Runtime Error:**

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

at Main.main(File.java:5)

**2**. **Developer**

Developers can explicitly write the implementation for handling the exception. Once an exception is handled, the normal execution of code will continue.

### **Advantage of Exception Handling**

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;

10.statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial). Some more advantages:

* Separating Error-Handling Code from “Regular” Code
* Propagating Errors Up the Call Stack
* Grouping and Differentiating Error Types

#### **Disadvantages of excepting handling in java**

* Experiencing unnecessary overhead
* Not understanding how the application really works
* Filling your logs with noisy events
* Inability to focus on what actually matters

### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## **Java Exception Handling Example**

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

**JavaExceptionExample.java**

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

**Output:**

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

rest of the code...

## **Java Exceptions**

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

## **Java try and catch**

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

### **Example**

public class Main {

public static void main(String[ ] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

}

}

}

**Output:**

Something went wrong.

## **Finally**

The finally statement lets you execute code, after try...catch, regardless of the result:

**Example**

public class Main {

public static void main(String[] args) {

try {

int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]);

} catch (Exception e) {

System.out.println("Something went wrong.");

} finally {

System.out.println("The 'try catch' is finished.");

}

}

}

**output:**

Something went wrong.

## **The throw keyword**

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception type**. There are many exception types available in Java: ArithmeticException, FileNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc:

**Example:**

Throw an exception if age is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

public class Main {

static void checkAge(int age) {

if (age < 18) {

throw new ArithmeticException("Access denied - You must be at least 18 years old.");

}

else {

System.out.println("Access granted - You are old enough!");

}

}

public static void main(String[] args) {

checkAge(15); // Set age to 15 (which is below 18...)

}

}

**Output:**

Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.  
        at Main.checkAge(Main.java:4)  
        at Main.main(Main.java:12)

## **Hierarchy of Java Exception classes**

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



**Collections in Java**

The **Collections in Java** provides an architecture to store and manipulate the group of objects, interfaces and classes. This java collection is a framework. This framework has several useful functions that have tons of useful functions, making a programmer task super easy.

This framework provides many interfaces (Queue, Set, List, Deque) and classes ( PriorityQueue, HashSet, ArrayList, Vector, LinkedList, LinkedHashSet).

## **Framework in java**

**Java frameworks** are the prewritten code used by developers to create applications in the java language.

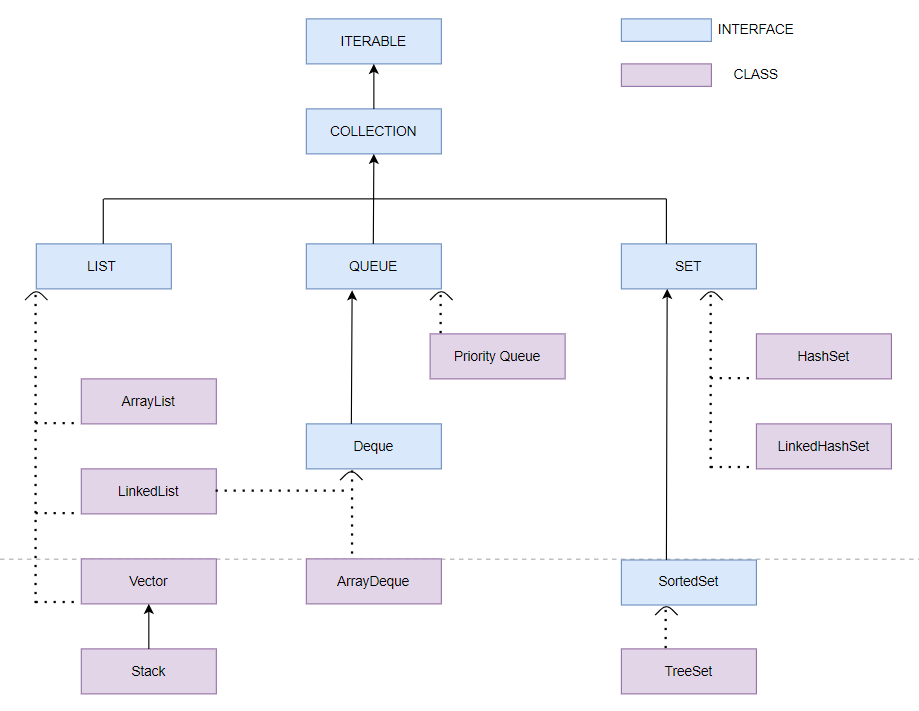
## **Collection framework**

The Collection framework is a unified architecture for storing and manipulating a group of objects.

The collection framework was designed to meet several goals, such as −

* The framework had to be high-performance and adapt a collection easy method.
* The implementations for the fundamental collections were to be highly efficient.
* The framework had to allow different types of collections to work in a similar manner.
* The framework had to extend and/or adapt a collection easily.

## **Collection Framework Hierarchy**



## **Need for the Collection Framework**

Suppose, A variable is created to store data and a 10 value is assigned (Example, int a =10). Now the programmer wants to store another data of the same datatype. So, the programmer needs to create another variable and assign a new value (Example, int b= 20).

If the programmer wants to store 100 values then the disadvantage of this is the programmer has to create multiple variables with a unique name and it is very time-consuming also.

In this case array concept is introduced. Programmer declare an array with specific size and store elements.

**For example,**

int  arr[] = new int[100]; // 100 is size of array   
arr[0] = 10;  
arr[1] = 20;  
arr[2] = 30;  
.  
.  
.  
arr[100] = 90;

This is the way of store multiple values of the same datatype.

But there are certain limitations

1. **Array**  
   Array stores the values of the same datatype i.e., Array is homogeneous but it can overcome by creating an array of object classes but this is not a good option.

Public class MultipleValues

{

Public static void main( string[] args)

{

objects a[]- new objects [5];

a[0]=10;

a[1]=10.45;

a[2]='A';

a[3]="name";

a[4]= true;

For( int i=0;i<a.leanght;i++)

{

system.out.println(a[1]);

}

}

}

The main limitation is an array has a fixed size (not growable) i.e.,

In the above example array is created with a size of five which means the array store only five data values.

If the size of the array is five and the user store only four values then memory is wasted.

To overcome this limitation, the Collection Framework was used.

## **Advantages of collections framework**

* Not necessary to learn multiple ad hoc collection APIs.
* It provides a standard interface for collections and also provides algorithms to manipulate them.
* It reduces the programming efforts by providing useful data structures and algorithms.
* Can establish a common language to pass collections back and forth that provides compatibility between unrelated APIs.
* The collection is resizable and can grow.