

**Java Virtual Machine (JVM)**

JVM is an abstract computing machine, or virtual machine that enables a computer to run a java program. It is a platform-independent execution environment that converts Java bytecode into machine language and executes it.

It compiles the source code into a machine independent code, which is also called Java Byte Code. It is a specification that provides runtime environment in which java bytecode can be executed.

**Java Runtime Environment (JRE)**

The Java Runtime Environment (JRE), also known as Java Runtime, is part of the Java Development Kit (JDK), a set of programming tools for developing Java applications. The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

Java Runtime Environment (JRE) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library.

**Java Development Kit (JDK)**

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

Java Development Kit (JDK) is a superset of a JRE and contains tools for Java programmers, e.g. a javac compiler. The Java Development Kit is provided free of charge either by Oracle Corporation directly, or by the OpenJDK open source project, which is governed by Oracle.

JVM (Java Virtual Machine) architecture.

|  |
| --- |
| [http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s640/JVM-arc1.png](http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s1600/JVM-arc1.png) |
|  |

JVM has various sub components internally. You can see all of them from the above diagram.  
  
**1. Class loader sub system:** JVM's class loader sub system performs 3 tasks  
      a. It loads .class file into memory.  
      b. It verifies byte code instructions.  
      c. It allots memory required for the program.

**2. Run time data area:** This is the memory resource used by JVM and it is divided into 5 parts  
      **a. Method area:** Method area stores class code and method code.  
      **b. Heap:** Objects are created on heap.  
      **c. Java stacks:** Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.  
      **d. Program counter registers:** The program counter registers store memory address of the instruction to be executed by the micro processor.  
      **e. Native method stacks:** The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.

**3. Native method interface:** Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.

**4. Native method library:** holds the native libraries information.

**5. Execution engine:** Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler

**Java Exception Handling**

**What is exception?**

Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

**What is exception handling?**

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

“**Error**” is a critical condition that cannot be handled by the code of the program.

The significant **difference between error and exception** is that an **error** is caused due to lack of system resources, and an **exception** is caused because of your code.

**Types of Exception**

1. Checked Exception
2. Unchecked Exception
3. Error

**Checked exceptions –**

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

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

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation. e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

**Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

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

**Common scenarios where exceptions may occur**

**1) Scenario where ArithmeticException occurs**

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

int a=50/0;//ArithmeticException

**2) Scenario where NullPointerException occurs**

If any variable have null value, performing any operation by the variable occurs an NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

**3) Scenario where NumberFormatException occurs**

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

**4) Scenario where ArrayIndexOutOfBoundsException occurs**

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**Java Exception Handling Keywords**

There are 5 keywords used in java exception handling.

try

catch

finally

throw

throws

**Catching Exceptions**

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

Syntax

try {

// Protected code

}catch(ExceptionName e1) {

// Catch block

}

**Exception**

The exception classes are created to specify different kind of exception scenarios so that we can easily identify the root cause and handle the exception according to it’s type.

Some of the useful methods of Throwable class are;

* **public String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it’s constructor.
* **public synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
* **public String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
* **public void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

**try block**

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

**Nested try block**

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

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.

**catch block**

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try. A catch statement involves declaring the type of exception you are trying to catch.

**Example of try and catch block**

**public** **class** Testtrycatch2{

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

**try**{

**int** data=50/0;

   }**catch**(ArithmeticException e){System.out.println(e);}

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

}

}

**Multi catch block**

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

**public** **class** TestMultipleCatchBlock{

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

**try**{

**int** a[]=**new** **int**[5];

    a[5]=30/0;

   }

**catch**(ArithmeticException e){System.out.println("task1 is completed");}

**catch**(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

**catch**(Exception e){System.out.println("common task completed");}

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

 }

}

**The Finally Block**

The finally block follows a try block or a catch block. A finally block of code is always executed whether exception is handled or not. Finally block in java can be used to put "cleanup" code such as closing a file, closing connection and stream etc.

**The Throws/Throw Keywords**

If a method does not handle a checked exception, the method must declare it using the throws keyword. The **throws** keyword appears at the end of a method's signature.

import java.io.\*;

public class className {

public void deposit(double amount) throws RemoteException {

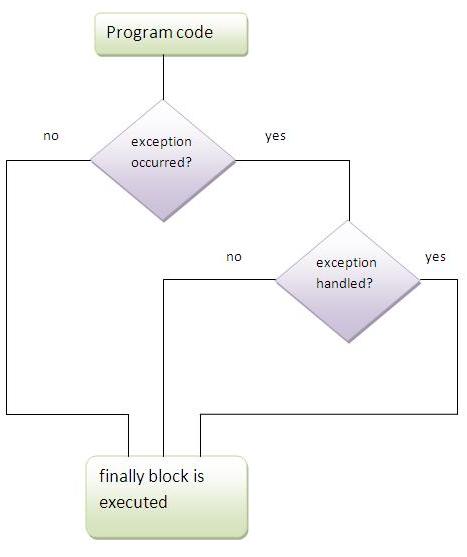
throw new RemoteException();

} }

we can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. throw keyword is used to throw exception to the runtime to handle it.

Catch(Exception ex)

throw ex

Try to understand the difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

**Advantages of Exception Handling**

Exception handling allows us to control the normal flow of the program by using exception handling in program.

It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

**Introduction to Debugging**

Debugging is the process of removing errors from programs. The program compiles and may even run, but one or more errors prevent the program from functioning properly on some or all inputs.

Ideally, programs are written correctly the first time and therefore never require debugging. Almost as ideal is to catch the errors in the program simply by thoroughly examining it, usually running it on various test cases in your mind and/or verifying invariants about what should be and must be true at this point. Evaluating the correctness of a program by examining the code is known as inspection.

Unfortunately, most of us have difficulty writing a correct program from scratch, or even removing our errors by inspection. This is particularly difficult in larger programs, or even virtually impossible unless programs are written with careful attention to modularity and abstraction. Therefore, programmers learn a variety of debugging techniques that involve actually executing the code on a computer.

**Print Statements**

A rudimentary debugging technique programmers are often introduced to early is debugging through the addition of print statements to a program. These statements are strategically placed to show the flow of control and the values of key variables. The output produced may make the problem obvious or be used to successively narrow down the problem location.

**Using an Interactive Debugger**

Java Development Kit, include a debugger called jdb. It allows us to do many useful things, such as examine variable values, see which methods are currently executing and who invoked them, and set breakpoints to make the program stop at various points.