Java is a simple and yet powerful object oriented programming language and it is in many respects similar to C++. Java originated at Sun Microsystems, Inc. in 1991. It was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. It was developed to provide a platform-independent programming language. This site gives you an **Introduction to Java Programming** accompanied with many **java examples**. Its a complete course in java programming for beginners to advanced java.

Platform independent

Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform *specific machine*, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

Java Virtual Machine

**What is the Java Virtual Machine? What  is its role?**

Java was designed with a concept of ‘write once and run everywhere’. Java Virtual Machine plays the central role in this concept. The JVM is the environment in which Java programs execute. It is a software that is implemented on top of real hardware and operating system. When the source code (.java files) is compiled, it is translated into byte codes and then placed into (.class) files. The JVM executes these bytecodes. So Java byte codes can be thought of as the machine language of the JVM. A JVM can either interpret the bytecode one instruction at a time or the bytecode can be compiled further for the real microprocessor using what is called ajust-in-time compiler**.** The JVM must be implemented on a particular platform before compiled programs can run on that platform.

**Object Oriented Programming**

Since Java is an object oriented programming language it has following features:

* Reusability of Code
* Emphasis on data rather than procedure
* Data is hidden and cannot be accessed by external functions
* Objects can communicate with each other through functions
* New data and functions can be easily added **Java has powerful features. The following are some of them:-**  
  Simple  
  Reusable  
  Portable (Platform Independent)  
  Distributed  
  Robust  
  Secure  
  High Performance  
  Dynamic  
  Threaded  
  Interpreted

**Object Oriented Programming** is a method of implementation in which programs are organized as cooperative collection of objects, each of which represents an instance of a class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

**OOP Concepts**

Four principles of Object Oriented Programming are  
Abstraction  
Encapsulation  
Inheritance  
Polymorphism

**Abstraction**

Abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

**Encapsulation**  
Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior ; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Encapsulation

\* Hides the implementation details of a class.  
\* Forces the user to use an interface to access data  
\* Makes the code more maintainable.

**Inheritance**

Inheritance is the process by which one object acquires the properties of another object.

**Polymorphism**

Polymorphism is the existence of the classes or methods in different forms or single name denoting different  
implementations.

Java is Distributed

With extensive set of routines to handle TCP/IP protocols like HTTP and FTP java can open and access the objects across net via URLs.

Java is Multithreaded

One of the powerful aspects of the Java language is that it allows multiple threads of execution to run concurrently within the same program A single Java program can have many different threads executing independently and continuously. Multiple Java applets can run on the browser at the same time sharing the CPU time.

Java is Secure

Java was designed to allow secure execution of code across network. To make Java secure many of the features of C and C++ were eliminated. Java does not use Pointers. Java programs cannot access arbitrary addresses in memory.

**Java source code**

A Java program is a collection of one or more java classes. A Java source file can contain more than one class definition and has a .java extension. Each class definition in a source file is compiled into a separate class file. The name of this compiled file is comprised of the name of the class with .class as an extension. Before we proceed further in this section, I would recommend you to go through the ‘Basic Language Elements’.

Below is a java sample code for the traditional Hello World program. Basically, the idea behind this Hello World program is to learn how to create a program, compile and run it. To create your java source code you can use any editor( Text pad/Edit plus are my favorites) or you can use an IDE like Eclipse.

|  |
| --- |
| public class HelloWorld {  public static void main(String[] args) {  System.out.println("Hello World");  }//End of main  }//End of HelloWorld Class |

Output  
Hello World

ABOUT THE PROGRAM

I created a class named “HelloWorld” containing a simple main function within it. The keyword class specifies that we are defining a class. The name of a public class is spelled exactly as the name of the file (Case Sensitive). All java programs begin execution with the method named main(). main method that gets executed has the following signature : public static void main(String args[]).Declaring this method as public means that it is accessible from outside the class so that the JVM can find it when it looks for the program to start it. It is necessary that the method is declared with return type void (i.e. no arguments are returned from the method). The main method contains a String argument array that can contain the command line arguments. The brackets { and } mark the beginning and ending of the class. The program contains a line ‘System.out.println(“Hello World”);’ that tells the computer to print out on one line of text namely ‘Hello World’. The semi-colon ‘;’ ends the line of code. The double slashes ‘//’ are used for comments that can be used to describe what a source code is doing. Everything to the right of the slashes on the same line does not get compiled, as they are simply the comments in a program.

Java Main method Declarations

class MainExample1 {public static void main(String[] args) {}}  
class MainExample2 {public static void main(String []args) {}}  
class MainExample3 {public static void main(String args[]) {}}

All the 3 valid main method’s shown above accepts a single String array argument.

**Compiling and Running an Application**

To compile and run the program you need the JDK distributed by Sun Microsystems. The JDK contains documentation, examples, installation instructions, class libraries and packages, and tools. Download an editor like Textpad/EditPlus to type your code. You must save your source code with a .java extension. The name of the file must be the name of the public class contained in the file.

Steps for Saving, compiling and Running a Java

Step 1:Save the program With .java Extension.  
Step 2:Compile the file from DOS prompt by typing javac <filename>.  
Step 3:Successful Compilation, results in creation of .class containing byte code  
Step 4:Execute the file by typing java <filename without extension>

**Java Development Kit**

The Java Developer’s Kit is distributed by Sun Microsystems. The JDK contains documentation, examples, installation instructions, class libraries and packages, and tools

**javadoc**

The javadoc tool provided by Sun is used to produce documentation for an application or program,

**Jar Files**

A jar file is used to group together related class files into a single file for more compact storage, distribution, and transmission.

**PATH and CLASSPATH**

The following are the general programming errors, which I think every beginning java programmer would come across. Here is a solution on how to solve the problems when running on a Microsoft Windows Machine.

**1. ‘javac’ is not recognized as an internal or external command, operable program or batch file**

When you get this error, you should conclude that your operating system cannot find the compiler (javac). To solve this error you need to set the PATH variable.

How to set the PATH Variable?

Firstly the PATH variable is set so that we can compile and execute programs from any directory without having to type the full path of the command. To set the PATH of jdk on your system (Windows XP), add the full path of the jdk<version>\bin directory to the PATH variable. Set the PATH as follows on a Windows machine:

a. Click Start > Right Click “My Computer” and click on “Properties”  
b. Click Advanced > Environment Variables.  
c. Add the location of bin folder of JDK installation for PATH in User Variables and System Variables. A typical value for PATH is:

C:\jdk<version>\bin (jdk<version is nothing but the name of the directory where jdk is installed)

If there are already some entries in the PATH variable then you must add a semicolon and then add the above value (Version being replaced with the version of JDK). The new path takes effect in each new command prompt window you open after setting the PATH variable.

**2. Exception in thread “main” java.lang.NoClassDefFoundError: HelloWorld**

If you receive this error, java cannot find your compiled byte code file, HelloWorld.class.If both your class files and source code are in the same working directory and if you try running your program from the current working directory than, your program must get executed without any problems as, java tries to find your .class file is your current directory. If your class files are present in some other directory other than that of the java files we must set the CLASSPATH pointing to the directory that contain your compiled class files.CLASSPATH can be set as follows on a Windows machine:

a. Click Start > Right Click “My Computer” and click on “Properties”  
b. Click Advanced > Environment Variables.

Add the location of classes’ folder containing all your java classes in User Variables.

If there are already some entries in the CLASSPATH variable then you must add a semicolon and then add the new value . The new class path takes effect in each new command prompt window you open after setting the CLASSPATH variable.

**Java 1.5**

The Java 1.5 released in September 2004.

Goals

Less code complexity  
Better readability  
More compile-time type safety  
Some new functionality (generics, scanner)

New Features

Enhanced for loop  
Enumerated types  
Autoboxing & unboxing  
Generic types  
Scanner  
Variable number of arguments (varargs)  
Static imports  
Annotations

**Java Operators**

They are used to manipulate primitive data types. **Java operators** can be classified as unary, binary, or ternary—meaning taking one, two, or three arguments, respectively. A unary operator may appear  
before (prefix) its argument or after (postfix) its argument. A binary or ternary operator appears between its arguments.

Operators in java fall into 8 different categories:

Java operators fall into eight different categories: **assignment, arithmetic, relational, logical, bitwise, compound assignment, conditional, and type**.

|  |
| --- |
| **Assignment Operators** =  **Arithmetic Operators** - + \* / % ++ --  **Relational Operators** > < >= <= == !=  **Logical Operators** && || & | ! ^  **Bit wise Operator** & | ^ >> >>>  **Compound Assignment Operators** += -= \*= /= %=  <<= >>= >>>=  **Conditional Operator** ?: |

Java has eight different operator types: assignment, arithmetic, relational, logical, bitwise, compound assignment, conditional, and type.

**Assignment operators**

The java assignment operator statement has the following syntax:

**<***variable***> = <***expression***>**

If the value already exists in the variable it is overwritten by the assignment operator (=).

|  |
| --- |
| public class AssignmentOperatorsDemo {  public AssignmentOperatorsDemo() {  // Assigning Primitive Values  int j, k;  j = 10; // j gets the value 10.  j = 5; // j gets the value 5. Previous value is overwritten.  k = j; // k gets the value 5.  System.out.println("j is : " + j);  System.out.println("k is : " + k);  // Assigning References  Integer i1 = new Integer("1");  Integer i2 = new Integer("2");  System.out.println("i1 is : " + i1);  System.out.println("i2 is : " + i2);  i1 = i2;  System.out.println("i1 is : " + i1);  System.out.println("i2 is : " + i2);  // Multiple Assignments  k = j = 10; // (k = (j = 10))  System.out.println("j is : " + j);  System.out.println("k is : " + k);  }  public static void main(String args[]) {  new AssignmentOperatorsDemo();  }  } |

**Arithmetic operators**

Java provides eight Arithmetic operators. They are for addition, subtraction, multiplication, division, modulo (or remainder), increment (or add 1), decrement (or subtract 1), and negation. An example program is shown below that demonstrates the different **arithmetic operators** in java.

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a String object, the other operand is implicitly converted to its string representation and string concatenation is performed.

*String message = 100 + "Messages"; //"100 Messages"*

|  |
| --- |
| public class ArithmeticOperatorsDemo {  public ArithmeticOperatorsDemo() {  int x, y = 10, z = 5;  x = y + z;  System.out.println("+ operator resulted in " + x);  x = y - z;  System.out.println("- operator resulted in " + x);  x = y \* z;  System.out.println("\* operator resulted in " + x);  x = y / z;  System.out.println("/ operator resulted in " + x);  x = y % z;  System.out.println("% operator resulted in " + x);  x = y++;  System.out.println("Postfix ++ operator resulted in " + x);  x = ++z;  System.out.println("Prefix ++ operator resulted in " + x);  x = -y;  System.out.println("Unary operator resulted in " + x);  // Some examples of special Cases  int tooBig = Integer.MAX\_VALUE + 1; // -2147483648 which is  // Integer.MIN\_VALUE.  int tooSmall = Integer.MIN\_VALUE - 1; // 2147483647 which is  // Integer.MAX\_VALUE.  System.out.println("tooBig becomes " + tooBig);  System.out.println("tooSmall becomes " + tooSmall);  System.out.println(4.0 / 0.0); // Prints: Infinity  System.out.println(-4.0 / 0.0); // Prints: -Infinity  System.out.println(0.0 / 0.0); // Prints: NaN  double d1 = 12 / 8; // result: 1 by integer division. d1 gets the value  // 1.0.  double d2 = 12.0F / 8; // result: 1.5  System.out.println("d1 is " + d1);  System.out.println("d2 iss " + d2);  }  public static void main(String args[]) {  new ArithmeticOperatorsDemo();  }  } |

**Relational operators**

Relational operators in Java are used to compare 2 or more objects. Java provides six relational operators:

greater than (>), less than (<), greater than or equal (>=), less than or equal (<=), equal (==), and not equal (!=).

All relational operators are binary operators, and their operands are numeric expressions.

Binary numeric promotion is applied to the operands of these operators. The evaluation results in a boolean value. Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators. An example program is shown below that demonstrates the different relational operators in java.

|  |
| --- |
| public class RelationalOperatorsDemo {  public RelationalOperatorsDemo( ) {  int x = 10, y = 5;  System.out.println("x > y : "+(x > y));  System.out.println("x < y : "+(x < y));  System.out.println("x >= y : "+(x >= y));  System.out.println("x <= y : "+(x <= y));  System.out.println("x == y : "+(x == y));  System.out.println("x != y : "+(x != y));  }  public static void main(String args[]){  new RelationalOperatorsDemo();  }  } |

**Logical operators**

Logical operators return a true or false value based on the state of the Variables. There are six logical, or boolean, operators. They are AND, conditional AND, OR, conditional OR, exclusive OR, and NOT. Each argument to a logical operator must be a boolean data type, and the result is always a boolean data type. An example program is shown below that demonstrates the different Logical operators in java.

|  |
| --- |
| public class LogicalOperatorsDemo {  public LogicalOperatorsDemo() {  boolean x = true;  boolean y = false;  System.out.println("x & y : " + (x & y));  System.out.println("x && y : " + (x && y));  System.out.println("x | y : " + (x | y));  System.out.println("x || y: " + (x || y));  System.out.println("x ^ y : " + (x ^ y));  System.out.println("!x : " + (!x));  }  public static void main(String args[]) {  new LogicalOperatorsDemo();  }  } |

Given that x and y represent boolean expressions, the boolean logical operators are defined in the Table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **!x** | **x & y**  **x && y** | **x | y**  **x || y** | **x ^ y** |
| true | true | false | true | true | false |
| true | false | false | false | true | true |
| false | true | true | false | true | true |
| false | false | true | false | false | false |

**Selection Statements**

**The If Statement**

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program. You can either have a single statementor a block of code within an if statement. Note that the conditional expression must be a Boolean expression.

The simple if statement has the following syntax:

*if (<conditional expression>)  
<statement action>*

Below is an example that demonstrates conditional execution based on if statement condition.

|  |
| --- |
| public class IfStatementDemo {  public static void main(String[] args) {  int a = 10, b = 20;  if (a > b)  System.out.println("a > b");  if (a < b)  System.out.println("b > a");  }  } |

Output

b > a

**The If-else Statement**

The if/else statement is an extension of the if statement. If the statements in the if statement fails, thestatements in the else block are executed. You can either have a single statement or a block of code within if-else blocks. Note that the conditional expression must be a Boolean expression.

The if-else statement has the following syntax:

*if (<conditional expression>)  
<statement action>  
else  
<statement action>*

Below is an example that demonstrates conditional execution based on if else statement condition.

|  |
| --- |
| public class IfElseStatementDemo {  public static void main(String[] args) {  int a = 10, b = 20;  if (a > b) {  System.out.println("a > b");  } else {  System.out.println("b > a");  }  }  } |

Output

b > a

**Switch Case Statement**The switch case statement, also called a case statement is a multi-way branch with several choices. A switch is easier to implement than a series of if/else statements. The switch statementbegins with a keyword, followed by an expression that equates to a no long integral value. Following the controlling expression is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label. The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed. Java includes a default label to use in cases where there are no matches. We can have a nested switch within a case block of an outer switch.

Its general form is as follows:

*switch (<non-long integral expression>) {  
case label1: <statement1>  
case label2: <statement2>  
…  
case labeln: <statementn>  
default: <statement>  
} // end switch*

When executing a switch statement, the program falls through to the next case. Therefore, if you want to exit in the middle of the switch statement code block, you must insert a break statement, which causes the programto continue executing after the current code block.

Below is a java example that demonstrates conditional execution based on nested if else statement condition to find the greatest of 3 numbers.

|  |
| --- |
| public class SwitchCaseStatementDemo {  public static void main(String[] args) {  int a = 10, b = 20, c = 30;  int status = -1;  if (a > b && a > c) {  status = 1;  } else if (b > c) {  status = 2;  } else {  status = 3;  }  switch (status) {  case 1:  System.out.println("a is the greatest");  break;  case 2:  System.out.println("b is the greatest");  break;  case 3:  System.out.println("c is the greatest");  break;  default:  System.out.println("Cannot be determined");  }  }  } |

Output

c is the greatest

# Iteration Statements

**While Statement**

The while statement is a looping construct control statement that executes a block of code while a condition is true. You can either have a single statement or a block of code within the while loop. The loop will never be executed if the testing expression evaluates to false. The loop condition must be a boolean expression.

The syntax of the while loop is

**while (<loop condition>)  
<statements>**

Below is an example that demonstrates the looping construct namely while loop used to print numbers from 1 to 10.

|  |
| --- |
| public class WhileLoopDemo {  public static void main(String[] args) {  int count = 1;  System.out.println("Printing Numbers from 1 to 10");  while (count <= 10) {  System.out.println(count++);  }  }  } |

Output

Printing Numbers from 1 to 10  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

**Do-while Loop Statement**

The do-while loop is similar to the while loop, except that the test is performed at the end of the loop instead of at the beginning. This ensures that the loop will be executed at least once. A do-while loop begins with the keyword do, followed by the statements that make up the body of the loop. Finally, the keyword while and the test expression completes the do-while loop. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop. You can either have a single statement or a block of code within the do-while loop.

The syntax of the do-while loop is

**do  
<loop body>  
while (<loop condition>);**

Below is an example that demonstrates the looping construct namely do-while loop used to print numbers from 1 to 10.

|  |
| --- |
| public class DoWhileLoopDemo {  public static void main(String[] args) {  int count = 1;  System.out.println("Printing Numbers from 1 to 10");  do {  System.out.println(count++);  } while (count <= 10);  }  } |

Output

Printing Numbers from 1 to 10  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

**or Loops**

The for loop is a looping construct which can execute a set of instructions a specified number of times. It’s a counter controlled loop.

The syntax of the loop is as follows:

**for (<initialization>; <loop condition>; <increment expression>)  
<loop body>**

The first part of a for statement is a starting initialization, which executes once before the loop begins. The <initialization> section can also be a comma-separated list of expression statements. The second part of a for statement is a test expression. As long as the expression is true, the loop will continue. If this expression is evaluated as false the first time, the loop will never be executed. The third part of the for statement is the body of the loop. These are the instructions that are repeated each time the program executes the loop. The final part of the for statement is an increment expression that automatically executes after each repetition of the loop body. Typically, this statement changes the value of the counter, which is then tested to see if the loop should continue.  
All the sections in the for-header are optional. Any one of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the <loop condition> signifies that the loop condition is true. The (;;) form of for loop is commonly used to construct an infinite loop.

Below is an example that demonstrates the looping construct namely for loop used to print numbers from 1 to 10.

|  |
| --- |
| public class ForLoopDemo {  public static void main(String[] args) {  System.out.println("Printing Numbers from 1 to 10");  for (int count = 1; count <= 10; count++) {  System.out.println(count);  }  }  } |

Output

Printing Numbers from 1 to 10  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10

# Java Access Specifiers

The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible.

Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified.

**Access Modifiers**

**1. private  
2. protected  
3. default  
4. public**

public access modifier

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member’s class.

default access modifier

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

**Below is a program to demonstrate the use of public, private, protected and default access modifiers while accessing fields and methods.**The output of each of these java files depict the Java access specifiers.

The first class is SubclassInSamePackage.java which is present in pckage1 package. This java file contains theBase class and a subclass within the enclosing class that belongs to the same class as shown below.

|  |
| --- |
| package pckage1;  class BaseClass {  public int x = 10;  private int y = 10;  protected int z = 10;  int a = 10; //Implicit Default Access Modifier  public int getX() {  return x;  }  public void setX(int x) {  this.x = x;  }  private int getY() {  return y;  }  private void setY(int y) {  this.y = y;  }  protected int getZ() {  return z;  }  protected void setZ(int z) {  this.z = z;  }  int getA() {  return a;  }  void setA(int a) {  this.a = a;  }  }  public class SubclassInSamePackage extends BaseClass {  public static void main(String args[]) {  BaseClass rr = new BaseClass();  rr.z = 0;  SubclassInSamePackage subClassObj = new SubclassInSamePackage();  //Access Modifiers - Public  System.out.println("Value of x is : " + subClassObj.x);  subClassObj.setX(20);  System.out.println("Value of x is : " + subClassObj.x);  //Access Modifiers - Public  // If we remove the comments it would result in a compilaton  // error as the fields and methods being accessed are private  /\* System.out.println("Value of y is : "+subClassObj.y);  subClassObj.setY(20);  System.out.println("Value of y is : "+subClassObj.y);\*/  //Access Modifiers - Protected  System.out.println("Value of z is : " + subClassObj.z);  subClassObj.setZ(30);  System.out.println("Value of z is : " + subClassObj.z);  //Access Modifiers - Default  System.out.println("Value of x is : " + subClassObj.a);  subClassObj.setA(20);  System.out.println("Value of x is : " + subClassObj.a);  }  } |

Output

Value of x is : 10  
Value of x is : 20  
Value of z is : 10  
Value of z is : 30  
Value of x is : 10  
Value of x is : 20

**Introduction to Java Classes**

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object.

Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class’s corresponding object are stored in the object’s instance variables. Methods define the operations that can be performed in javaprogramming.

A **class** has the following general syntax:

<class modifiers>class<class name>  
<extends clause> <implements clause>{

// Dealing with Classes (Class body)  
<field declarations (Static and Non-Static)>  
<method declarations (Static and Non-Static)>  
<Inner class declarations>  
<nested interface declarations>  
<constructor declarations>  
<Static initializer blocks>  
}

Below is an example showing the Objects and Classes of the Cube class that defines 3 fields namely length, breadth and height. Also the class contains a member function getVolume().

|  |
| --- |
| public class Cube {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  } |

*How do you reference a data member/function?*

This is accomplished by stating the name of the object reference, followed by a period (dot), followed by thename of the member inside the object.  
( objectReference.member ). You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this: objectName.methodName(arg1, arg2, arg3).

*For example:*

cubeObject.length = 4;  
cubeObject.breadth = 4;  
cubeObject.height = 4;  
cubeObject.getvolume()

**Class Variables – Static Fields**

We use class variables also know as Static fields when we want to share characteristics across all objects within a class. When you declare a field to be static, only a single instance of the associated variable is created common to all the objects of that class. Hence when one object changes the value of a class variable, it affects all objects of the class. We can access a class variable by using the name of the class, and not necessarily using a reference to an individual object within the class. Static variables can be accessed even though no objects of that class exist. It is declared using static keyword.

**Class Methods – Static Methods**

Class methods, similar to Class variables can be invoked without having an instance of the class. Class methodsare often used to provide global functions for Java programs. For example, methods in the java.lang.Mathpackage are class methods. You cannot call non-static methods from inside a static method.

**Instance Variables**

Instance variables stores the state of the object. Each class would have its own copy of the variable. Every object has a state that is determined by the values stored in the object. An object is said to have changed its state when one or more data values stored in the object have been modified. When an object responds to a message, it will usually perform an action, change its state etc. An object that has the ability to store values is often said to have persistence.

*Consider this simple Java program showing the use of static fields and static methods*

|  |
| --- |
| // Class and Object initialization showing the Object Oriented concepts in Java  class Cube {  int length = 10;  int breadth = 10;  int height = 10;  public static int numOfCubes = 0; // static variable  public static int getNoOfCubes() { //static method  return numOfCubes;  }  public Cube() {  numOfCubes++; //  }  }  public class CubeStaticTest {  public static void main(String args[]) {  System.out.println("Number of Cube objects = " + Cube.numOfCubes);  System.out.println("Number of Cube objects = "  + Cube.getNoOfCubes());  }  } |

Output

Number of Cube objects = 0  
Number of Cube objects = 0

**Final Variable, Methods and Classes**

In Java we can mark fields, methods and classes as final. Once marked as final, these items cannot be changed.

Variables defined in an interface are implicitly final. You can’t change value of a final variable (is a constant). A final class can’t be extended i.e., final class may not be subclassed. This is done for security reasons with basic classes like String and Integer. It also allows the compiler to make some optimizations, and makes thread safety a little easier to achieve. A final method can’t be overridden when its class is inherited. Any attempt to override or hide a final method will result in a compiler error.

**Introduction to Java Objects**

The Object Class is the super class for all classes in Java.

Some of the object class methods are  
  
equals  
toString()  
wait()  
notify()  
notifyAll()  
hashcode()  
clone()

An **object** is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior.

An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.

Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines thecharacteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the **new operator**. This process is called instantiating an object or creating an object instance.

When you create a new object, you use the new operator to instantiate the object. The new operator returns the location of the object which you assign o a reference type.

Below is an example showing the creation of Cube objects by using the new operator.

|  |
| --- |
| public class Cube {  int length = 10;  int breadth = 10;  int height = 10;  public int getVolume() {  return (length \* breadth \* height);  }  public static void main(String[] args) {  Cube cubeObj; // Creates a Cube Reference  cubeObj = new Cube(); // Creates an Object of Cube  System.out.println("Volume of Cube is : " + cubeObj.getVolume());  }  } |

**Method Overloading**

Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call

Below is an example of a class demonstrating Method Overloading

|  |
| --- |
| public class MethodOverloadDemo {  void sumOfParams() { // First Version  System.out.println("No parameters");  }  void sumOfParams(int a) { // Second Version  System.out.println("One parameter: " + a);  }  int sumOfParams(int a, int b) { // Third Version  System.out.println("Two parameters: " + a + " , " + b);  return a + b;  }  double sumOfParams(double a, double b) { // Fourth Version  System.out.println("Two double parameters: " + a + " , " + b);  return a + b;  }  public static void main(String args[]) {  MethodOverloadDemo moDemo = new MethodOverloadDemo();  int intResult;  double doubleResult;  moDemo.sumOfParams();  System.out.println();  moDemo.sumOfParams(2);  System.out.println();  intResult = moDemo.sumOfParams(10, 20);  System.out.println("Sum is " + intResult);  System.out.println();  doubleResult = moDemo.sumOfParams(1.1, 2.2);  System.out.println("Sum is " + doubleResult);  System.out.println();  }  } |

Output

No parameters

One parameter: 2

Two parameters: 10 , 20  
Sum is 30

Two double parameters: 1.1 , 2.2  
Sum is 3.3000000000000003

# Java Constructors

A **java constructor** has the same name as the name of the class to which it belongs. Constructor’s syntax does not include a return type, since constructors never return a value.

Constructors may include parameters of various types. When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition.

Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

The only action taken by the implicit default constructor is to call the superclass constructor using the super() call. Constructor arguments provide you with a way to provide parameters for the initialization of an object.

Below is an example of a cube class containing 2 constructors. (one default and one parameterized constructor).

|  |
| --- |
| public class Cube1 {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube1() {  length = 10;  breadth = 10;  height = 10;  }  Cube1(int l, int b, int h) {  length = l;  breadth = b;  height = h;  }  public static void main(String[] args) {  Cube1 cubeObj1, cubeObj2;  cubeObj1 = new Cube1();  cubeObj2 = new Cube1(10, 20, 30);  System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());  System.out.println("Volume of Cube1 is : " + cubeObj2.getVolume());  }  } |

**Note:** If a class defines an explicit constructor, it no longer has a default constructor to set the state of the objects.  
If such a class requires a default constructor, its implementation must be provided. Any attempt to call the default constructor will be a compile time error if an explicit default constructor is not provided in such a case.

**Java Overloaded Constructors**

Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as theclass, />their signatures are differentiated by their parameter lists. The above example shows that the Cube1 constructor is overloaded one being the default constructor and the other being a parameterized constructor.

It is possible to use this() construct, to implement local chaining of constructors in a class. The this() call in a constructorinvokes the an other constructor with the corresponding parameter list within the same class. Calling the default constructor to create a Cube object results in the second and third parameterized constructors being called as well. Java requires that any this() call must occur as the first statement in a constructor.

Below is an example of a cube class containing 3 constructors which demostrates the this() method in Constructors context

|  |
| --- |
| public class Cube2 {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube2() {  this(10, 10);  System.out.println("Finished with Default Constructor");  }  Cube2(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having 2 params");  }  Cube2(int l, int b, int h) {  length = l;  breadth = b;  height = h;  System.out.println("Finished with Parameterized Constructor having 3 params");  }  public static void main(String[] args) {  Cube2 cubeObj1, cubeObj2;  cubeObj1 = new Cube2();  cubeObj2 = new Cube2(10, 20, 30);  System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());  System.out.println("Volume of Cube2 is : " + cubeObj2.getVolume());  }  }  public class Cube2 {  int length;  int breadth;  int height;  public int getVolume() {  return (length \* breadth \* height);  }  Cube2() {  this(10, 10);  System.out.println("Finished with Default Constructor");  }  Cube2(int l, int b) {  this(l, b, 10);  System.out.println("Finished with Parameterized Constructor having 2 params");  }  Cube2(int l, int b, int h) {  length = l;  breadth = b;  height = h;  System.out.println("Finished with Parameterized Constructor having 3 params");  }  public static void main(String[] args) {  Cube2 cubeObj1, cubeObj2;  cubeObj1 = new Cube2();  cubeObj2 = new Cube2(10, 20, 30);  System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());  System.out.println("Volume of Cube2 is : " + cubeObj2.getVolume());  }  } |

Output

Finished with Parameterized Constructor having 3 params  
Finished with Parameterized Constructor having 2 params  
Finished with Default Constructor  
Finished with Parameterized Constructor having 3 params  
Volume of Cube1 is : 1000  
Volume of Cube2 is : 6000

# Java Inheritance

defines an is-a relationship between a superclass and its subclasses. This means that an object of a subclass can be used wherever an object of the superclass can be used. Class **Inheritance in java**mechanism is used to build new classes from existing classes. The inheritance relationship is transitive: if class x extends class y, then a class z, which extends class x, will also inherit from class y.

For example a car class can inherit some properties from a General vehicle class. Here we find that the baseclass is the vehicle class and the subclass is the more specific car class. A subclass must use the extends clause to derive from a super class which must be written in the header of the subclass definition. The subclass inherits members of the superclass and hence promotes code reuse. The subclass itself can add its own new behavior and properties. The java.lang.Object class is always at the top of any Class inheritance hierarchy.

|  |
| --- |
| class Box {  double width;  double height;  double depth;  Box() {  }  Box(double w, double h, double d) {  width = w;  height = h;  depth = d;  }  void getVolume() {  System.out.println("Volume is : " + width \* height \* depth);  }  }  public class MatchBox extends Box {  double weight;  MatchBox() {  }  MatchBox(double w, double h, double d, double m) {  super(w, h, d);  weight = m;  }  public static void main(String args[]) {  MatchBox mb1 = new MatchBox(10, 10, 10, 10);  mb1.getVolume();  System.out.println("width of MatchBox 1 is " + mb1.width);  System.out.println("height of MatchBox 1 is " + mb1.height);  System.out.println("depth of MatchBox 1 is " + mb1.depth);  System.out.println("weight of MatchBox 1 is " + mb1.weight);  }  } |

Output

Volume is : 1000.0  
width of MatchBox 1 is 10.0  
height of MatchBox 1 is 10.0  
depth of MatchBox 1 is 10.0  
weight of MatchBox 1 is 10.0

*What is not possible using java class Inheritance?*

1. Private members of the superclass are not inherited by the subclass and can only be indirectly accessed.  
2. Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same packageas the superclass.  
3. Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.  
4. A subclass can extend only one superclass

|  |
| --- |
| class Vehicle {  // Instance fields  int noOfTyres; // no of tyres  private boolean accessories; // check if accessorees present or not  protected String brand; // Brand of the car  // Static fields  private static int counter; // No of Vehicle objects created  // Constructor  Vehicle() {  System.out.println("Constructor of the Super class called");  noOfTyres = 5;  accessories = true;  brand = "X";  counter++;  }  // Instance methods  public void switchOn() {  accessories = true;  }  public void switchOff() {  accessories = false;  }  public boolean isPresent() {  return accessories;  }  private void getBrand() {  System.out.println("Vehicle Brand: " + brand);  }  // Static methods  public static void getNoOfVehicles() {  System.out.println("Number of Vehicles: " + counter);  }  }  class Car extends Vehicle {  private int carNo = 10;  public void printCarInfo() {  System.out.println("Car number: " + carNo);  System.out.println("No of Tyres: " + noOfTyres); // Inherited.  // System.out.println("accessories: " + accessories); // Not Inherited.  System.out.println("accessories: " + isPresent()); // Inherited.  // System.out.println("Brand: " + getBrand()); // Not Inherited.  System.out.println("Brand: " + brand); // Inherited.  // System.out.println("Counter: " + counter); // Not Inherited.  getNoOfVehicles(); // Inherited.  }  }  public class VehicleDetails { // (3)  public static void main(String[] args) {  new Car().printCarInfo();  }  } |

Output

Constructor of the Super class called  
Car number: 10  
No of Tyres: 5  
accessories: true  
Brand: X  
Number of Vehicles: 1

**this and super keywords**

The two keywords, this and super to help you explicitly name the field or method that you want. Using this and super you have full control on whether to call a method or field present in the same class or to call from the immediate superclass. This keyword is used as a reference to the current object which is an instance of the current class. The keyword super also references the current object, but as an instance of the current class’s super class.

The this reference to the current object is useful in situations where a local variable hides, or shadows, a field with the same name. If a method needs to pass the current object to another method, it can do so using the this reference. Note that the this reference cannot be used in a static context, as static code is not executed in the context of any object.

|  |
| --- |
| class Counter {  int i = 0;  Counter increment() {  i++;  return this;  }  void print() {  System.out.println("i = " + i);  }  }  public class CounterDemo extends Counter {  public static void main(String[] args) {  Counter x = new Counter();  x.increment().increment().increment().print();  }  } |

Output

Volume is : 1000.0  
width of MatchBox 1 is 10.0  
height of MatchBox 1 is 10.0  
depth of MatchBox 1 is 10.0  
weight of MatchBox 1 is 10.0

**Abstract Class in java**

**Java Abstract classes** are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

A class abstract Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

|  |
| --- |
| abstract class Vehicle {  int numofGears;  String color;  abstract boolean hasDiskBrake();  abstract int getNoofGears();  } |

Example of a shape class as an abstract class

|  |
| --- |
| abstract class Shape {  public String color;  public Shape() {  }  public void setColor(String c) {  color = c;  }  public String getColor() {  return color;  }  abstract public double area();  } |

We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

|  |
| --- |
| public class Point extends Shape {  static int x, y;  public Point() {  x = 0;  y = 0;  }  public double area() {  return 0;  }  public double perimeter() {  return 0;  }  public static void print() {  System.out.println("point: " + x + "," + y);  }  public static void main(String args[]) {  Point p = new Point();  p.print();  }  } |

Output

point: 0, 0

Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstractmethods of the Shape class must be implemented by the Point class.

The subclass must define an implementation for every abstract method of the abstract superclass, or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class.

A big Disadvantage of using abstract classes is not able to use multiple inheritance. In the sense, when a class extends an abstract class, it can’t extend any other class.

**Java Interface**

[[](http://americanportal.com/)](http://americanportal.com/)

In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

**Example 1**: Below is an example of a Shape interface

|  |
| --- |
| interface Shape {  public double area();  public double volume();  } |

Below is a Point class that implements the Shape interface.

|  |
| --- |
| public class Point implements Shape {  static int x, y;  public Point() {  x = 0;  y = 0;  }  public double area() {  return 0;  }  public double volume() {  return 0;  }  public static void print() {  System.out.println("point: " + x + "," + y);  }  public static void main(String args[]) {  Point p = new Point();  p.print();  }  } |

Similarly, other shape objects can be created by interface programming by implementing generic Shape Interface.

Example 2: Below is a java interfaces program showing the power of interface programming in java

Listing below shows 2 interfaces and 4 classes one being an abstract class.  
Note: The method *toString* in class *A1* is an overridden version of the method defined in the class named **Object**. The classes *B1* and *C1* satisfy the interface contract. But since the class**D1** does not define all the methods of the implemented interface *I2*, the class D1 is declared abstract.  
Also,  
i1.methodI2() produces a compilation error as the method is not declared in *I1* or any of its super interfaces if present. Hence a downcast of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a downcast.

When we invoke the toString() method which is a method of an Object, there does not seem to be any problem as every interface or class extends Object and any class can override the default toString() to suit your application needs. ((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are “siblings”. You can’t cast siblings into one another.

When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.

|  |
| --- |
| interface I1 {  void methodI1(); // public static by default  }  interface I2 extends I1 {  void methodI2(); // public static by default  }  class A1 {  public String methodA1() {  String strA1 = "I am in methodC1 of class A1";  return strA1;  }  public String toString() {  return "toString() method of class A1";  }  }  class B1 extends A1 implements I2 {  public void methodI1() {  System.out.println("I am in methodI1 of class B1");  }  public void methodI2() {  System.out.println("I am in methodI2 of class B1");  }  }  class C1 implements I2 {  public void methodI1() {  System.out.println("I am in methodI1 of class C1");  }  public void methodI2() {  System.out.println("I am in methodI2 of class C1");  }  }  // Note that the class is declared as abstract as it does not  // satisfy the interface contract  abstract class D1 implements I2 {  public void methodI1() {  }  // This class does not implement methodI2() hence declared abstract.  }  public class InterFaceEx {  public static void main(String[] args) {  I1 i1 = new B1();  i1.methodI1(); // OK as methodI1 is present in B1  // i1.methodI2(); Compilation error as methodI2 not present in I1  // Casting to convert the type of the reference from type I1 to type I2  ((I2) i1).methodI2();  I2 i2 = new B1();  i2.methodI1(); // OK  i2.methodI2(); // OK  // Does not Compile as methodA1() not present in interface reference I1  // String var = i1.methodA1();  // Hence I1 requires a cast to invoke methodA1  String var2 = ((A1) i1).methodA1();  System.out.println("var2 : " + var2);  String var3 = ((B1) i1).methodA1();  System.out.println("var3 : " + var3);  String var4 = i1.toString();  System.out.println("var4 : " + var4);  String var5 = i2.toString();  System.out.println("var5 : " + var5);  I1 i3 = new C1();  String var6 = i3.toString();  System.out.println("var6 : " + var6); // It prints the Object toString() method  Object o1 = new B1();  // o1.methodI1(); does not compile as Object class does not define  // methodI1()  // To solve the probelm we need to downcast o1 reference. We can do it  // in the following 4 ways  ((I1) o1).methodI1(); // 1  ((I2) o1).methodI1(); // 2  ((B1) o1).methodI1(); // 3  /\*  \*  \* B1 does not have any relationship with C1 except they are "siblings".  \*  \* Well, you can't cast siblings into one another.  \*  \*/  // ((C1)o1).methodI1(); Produces a ClassCastException  }  } |

Output

I am in methodI1 of class B1  
I am in methodI2 of class B1  
I am in methodI1 of class B1  
I am in methodI2 of class B1  
var2 : I am in methodC1 of class A1  
var3 : I am in methodC1 of class A1  
var4 : toString() method of class A1  
var5 : toString() method of class A1  
var6 : C1@190d11  
I am in methodI1 of class B1  
I am in methodI1 of class B1  
I am in methodI1 of class B1

## Interface vs Abstract Class

1. Abstract class is a class which contain one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concretemethods. A Java Interface can contain only method declarations and public static final constants and doesn’t contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present.

2. Abstract class definition begins with the keyword “abstract” keyword followed by Class definition. An Interface definition begins with the keyword “interface”.

3. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses

4. All variables in an Interface are by default – public static final while an abstract class can have instance variables.

5. An interface is also used in situations when a class needs to extend an other class apart from the abstractclass. In such situations its not possible to have multiple inheritance of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritance.

6. An Interface can only have public members whereas an abstract class can contain private as well as protected members.

7. A class implementing an interface must implement all of the methods defined in the interface, while a classextending an abstract class need not implement any of the methods defined in the abstract class.

8. The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those method in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

9. Interfaces are slow as it requires extra indirection to to find corresponding method in in the actual class. Abstract classes are fast

10.Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Automobile class might  
implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

Note: There is no difference between a fully abstract class (all methods declared as abstract and all fields are public static final) and an interface.

Note: If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they  
share is a set of method signatures, then tend towards an interface.

# Understanding Java Exceptions

ction. Exception handlers can be written to catch a specific exception such as Number Format exception, or an entire group of exceptions by using a generic exception handlers. Any exceptions not specifically handled within a Java program are caught by the Java run time environment

An exception is a subclass of the Exception/Error class, both of which are subclasses of the Throwable class. Java exceptions are raised with the throw keyword and handled within a catch block.

A Program Showing How the JVM throws an Exception at runtime

|  |
| --- |
| public class DivideException {  public static void main(String[] args) {  division(100,4); // Line 1  division(100,0); // Line 2  System.out.println("Exit main().");  }  public static void division(int totalSum, int totalNumber) {  System.out.println("Computing Division.");  int average = totalSum/totalNumber;  System.out.println("Average : "+ average);  }  } |

[**Download**](http://www.javabeginner.com/images/java-exceptions-1.zip) DivideException.java

An ArithmeticException is thrown at runtime when Line 11 is executed because integer division by 0 is an illegal operation. The “Exit main()” message is never reached in the main method

Output

Computing Division.  
java.lang.ArithmeticException: / by zero  
Average : 25  
Computing Division.  
at DivideException.division(DivideException.java:11)  
at DivideException.main(DivideException.java:5)  
Exception in thread “main”

### Exceptions in Java

Throwable Class

[[](http://americanportal.com/)](http://americanportal.com/)

The Throwable class provides a String variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred. All classes of throwables define a one-parameter constructor that takes a string as the detail message.

The class Throwable provides getMessage() function to retrieve an exception. It has a printStackTrace() method to print the stack trace to the standard error stream. Lastly It also has a toString() method to print a short description of the exception. For more information on what is printed when the following messages are invoked, please refer the java docs.

Syntax

String getMessage()

void printStackTrace()

String toString()

Class Exception

The class Exception represents exceptions that a program faces due to abnormal or special conditions during execution. Exceptions can be of 2 types: Checked (Compile time Exceptions)/ Unchecked (Run time Exceptions).

Class RuntimeException

Runtime exceptions represent programming errors that manifest at runtime. For example ArrayIndexOutOfBounds, NullPointerException and so on are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class. These are basically business logic programming errors.

Class Error

Errors are irrecoverable condtions that can never be caught. Example: Memory leak, LinkageError etc. Errors are direct subclass of Throwable class.

## Checked and Unchecked Exceptions

**Checked exceptions** are subclass’s of Exception excluding class RuntimeException and its subclasses. Checked Exceptions forces programmers to deal with the exception that may be thrown. Example: Arithmetic exception. When a checked exception occurs in a method, the method must either catch the exception and take the appropriate action, or pass the exception on to its caller

**Unchecked exceptions** are RuntimeException and any of its subclasses. Class Error and its subclasses also are unchecked. Unchecked exceptions , however, the compiler doesn’t force the programmers to either catch the exception or declare it in a throws clause. In fact, the programmers may not even know that the exception could be thrown. Example: ArrayIndexOutOfBounds Exception. They are either irrecoverable (Errors) and the program should not attempt to deal with them, or they are logical programming errors. (Runtime Exceptions). Checked exceptions must be caught at compile time. Runtime exceptions do not need to be. Errors often cannot be.

Exception Statement Syntax

Exceptions are handled using a try-catch-finally construct, which has the Syntax

try {  
<code>  
} catch (<exception type1> <parameter1>) { // 0 or more  
<statements>  
}  
} finally { // finally block  
<statements>  
}

try Block  
The java code that you think may produce an exception is placed within a try block for a  
suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the  
matching catch block to handle the error. Again if the matching catch handler is not found execution  
proceeds with the finally block and the default exception handler throws an exception.. If an exception is  
generated within the try block, the remaining statements in the try block are not executed.

catch Block  
Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed  
(Though the catch block throws an exception).

finally Block  
A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc. If the finally clock executes a control transfer statement such as a return or a break statement, then this control  
statement determines how the execution will proceed regardless of any return or control statement present in the try or catch.

The following program illustrates the scenario.

|  |
| --- |
| try {  <code>  } catch (<exception type1> <parameter1>) { // 0 or more  <statements>  }  } finally { // finally block  <statements>  } |

Output

Computing Division.  
Exception : / by zero  
Finally Block Executes. Exception Occurred  
result : -1

Below is a program showing the Normal Execution of the Program.

Please note that no NullPointerException is generated as was expected by most people

|  |
| --- |
| public class DivideException2 {  public static void main(String[] args) {  int result = division(100,0); // Line 2  System.out.println("result : "+result);  }  public static int division(int totalSum, int totalNumber) {  int quotient = -1;  System.out.println("Computing Division.");  try{  quotient = totalSum/totalNumber;  }  catch(Exception e){  System.out.println("Exception : "+ e.getMessage());  }  finally{  if(quotient != -1){  System.out.println("Finally Block Executes");  System.out.println("Result : "+ quotient);  }else{  System.out.println("Finally Block Executes. Exception Occurred");  return quotient;  }  }  return quotient;  }  } |

Output

null (And not NullPointerException)