**Programming Language**

A programming language is a language which is used to develop software.

Programming Language is of two types:

1. Structured Programming Language
2. Object Oriented Programming Language

**Structured Programming Language:** It is a high level programming language having its efficient features which replaced the assembly code when creating system programs. The creation of C Language shook the computer world. The creation of C is considered by many to have marked the beginning of the modern age of computer languages. During the late 1970s and early 1980s, C became the dominant computer programming language and it is still widely used today. Since C is a successful and useful language but programmers faced a problem with this language – its complexity. The increasing complexity of programs has driven the need for better ways to manage that complexity. This leads the creation of new language.

**Object Oriented Programming Language:** The easy way to master the management of complexity in the development of a software system is through the use of data abstraction. To solve this problem a new way to program is invented called object – oriented programming (OOP). OOP is a programming methodology that helps organize programs through the use of encapsulation, abstraction, polymorphism and inheritance. The result was the creation of C++. C++ extends C by adding object – oriented features, attributes and benefits. This is a crucial reason for the success of C++ as a language.

**THE CREATION OF JAVA**

In 1990, Sun Microsystems began a project called Green Project to develop software for consumer electronics. James Gosling, a veteran of classic network software design, was assigned to the new Green Project. James Gosling was appointed as a chief programmer in Sun Microsystems. Java was conceived by James Gosling, Patrick Naugton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called as “Oak”. As Oak matured, the World Wide Web was growing dramatically. Thus, in 1994, they completed work on product known as WebRunner, an early web browser written in Oak. WebRunner was later renamed HotJava. Finally, in 1995, Oak was renamed a Java.

**WHY NEED OF A NEW LANGAUGE WHEN C++ ALREADY EXISTS**

1. By the end of 1980s and early 1990s, object oriented programming with C++ took hold. But the trouble with C++ is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that the compilers are expensive and time consuming to create. An easier and most cost effective solution was needed. In an attempt to find such a solution, Gosling and others began work on portable, platform independent language that could be used to produce code that would run on variety of CPUs under different environments. This effort ultimately led to the creation of Java.
2. C++ language suffers from a criticism that it was not fully object – oriented because as per the rules of object oriented programming, codes must be enclosed within a class, but here main() is outside of the class. In C++, the main() function, if defined in the class, it becomes the member of that class. Every member of the class must be invoked using the object of the class only and this object gets created in main(). So the program never gets executed.

In 1990s, when Sun Microsystems was designing Java language, this was

taken as a challenge to design the language with main() function inside of the class. So to

resolve the problem with main() they have introduced a concept of “static members” i.e.

if any member of the class if declared as static, doesn’t require object of the class for

invoking that member. So if the main() method is declared as static, we don’t need object

to be created for the execution of the main() and it become the entry point of our

program.

**JAVA PROGRAM STRUCTURE**

class example

{

Collection of Members

static void main()

{

Create object of the class

Using the object invoke the members of the class

}

}

**JAVA DEVELOPMENT KIT (JDK)**

Java can be used to create three types of programs:

1. Applications
2. Applets
3. Servlets

An **application** is a program that runs on your computer, under the operating system of that computer. That is, an application created by java is more or less like one created using C or C++.

An **applet** is an application designed to be transmitted over the Internet and executed by java compatible Web browser. An applet is actually a tiny Java Program; dynamically downloaded across the network, just like an image, sound file or video clip. An applet is a program that can react to user input and dynamically change; not just run the same animation or sound over and over.

A **servlet** is a special Java program that runs inside a web server. It is a small pluggable extension to the server that enhances the server’s functionality.

The Java Development Kit (JDK) from Sun’s Java Software division contains the basic tools and libraries necessary for creating and executing Java applets and applications. It also contains a number of useful utilities for debugging and documenting Java source code.

**JAVA’S MAGIC: THE BYTECODE**

The key that allows java to solve both the security and portability problem just describes is that the output of a Java compiler is not executable code. Rather it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).

**JAVA VIRTUAL MACHINE (JVM)**

The JVM is an interpreter for bytecode. To execute Java bytecode, the VM uses a class loader to fetch java bytecode from a disk or a network. Each class file is fed to a bytecode verifier that ensures the class is formatted correctly and will not corrupt memory when it is executed. The bytecode verification phase adds to the time it takes to load a class, but it actually allows the program to run faster because the class verification is performed only once, not continuously as the program runs.

The execution unit of the VM carries out the instructions specified in the bytecode. The simplest execution unit is an interpreter, which is a program that reads the bytecode, interprets their meaning, and then performs the associated function. Interpreters are generally much slower than native code compilers because they continuously need to look up the meaning of each bytecode during execution. There is an elegant alternatively to interpreting code, called **Just-in-Time (JIT)** compilation.

**JUST-IN-TIME (JIT) COMPILER**

The JIT compiler converts the bytecode to native code instructions on the user’s machine immediately before execution. Traditional native code compilers run on the developer’s machine, are used by programmers, and produce non-portable executables. JIT compilers run on the user’s machine and are transparent to the user; the resulting native code instructions do not need to be ported because they are already at their destinations.

**THE JAVA BUZZWORDS**

The key considerations were summed up by the Java team in the following list of buzzwords.

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multi-threaded
7. Architecture Neural
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

**Simple**

Java was designed to be easy for the professional programmer to learn and use effectively because Java inherits the C/C++ syntax and many of the object-oriented features of C++.

**Secure**

When you use a Java compatible web browser you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

**Portability**

As java is interpreted to bytecode, so it can run on any platform. This makes Java as a platform independent language. This theory is called portability.

**Object oriented**

Java is pure high level object oriented language as data and functions are treated as a single entity and all the functions are done by object of the class.

**Robust**

Robust means free from to worry about many of the most common causes of programming errors. The reasons for program failures are:

1. Memory Management Mistakes
2. Mishandled Exceptional Conditions i.e. runtime errors

Memory management can be a difficult tedious task in traditional programming environment. For example, in C/C++, the programmer must manually allocate and de-allocate the memory. Java eliminates these problems by managing memory allocations and de-allocation for you. In fact, de-allocation is completely automatic because java provides garbage collection for unused objects.

Exceptional conditions in java are held by object oriented exceptional handling.

**Multi Threaded**

Java was designed to meet real world requirement of creating interactive network programs. To accomplish this, java supports multi-threaded programming, which allows you to write programs that do many thins simultaneously.

**Architectural Neutral**

One of the main problems facing programmers is that no guarantee exists if you write a program today, it will run tomorrow even on the same machine. The java designers made several hard decisions in the java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever”.

**Interpreted and High Performance**

Java enables the creation of cross platform programs by compiling into an intermediate representation called Java Bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. The java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a JIT compiler.

**Distributed**

Java was designed for the distributed environment of the internet because it handles TCP/IP protocols. Java received these interfaces in a package called Remote Method Invocation (RMI).

**Dynamic**

Java programs carry with them substantial amount of run time type information that is used to verify and resolve access to objects at run – time. This makes it possible to dynamically link code in safe and expedient manner.

**HOW TO INSTALL JAVA**

1. Paste the application file in a drive. (say C:\ drive).
2. Double click on icon
3. Click on next
4. Click on next
5. Click on OK
6. Open command prompt (Start > All Programs > Accessories > Run)
7. Change the directory in which you install the java. (C:\java\jdk\bin)
8. Type javac and press Enter.

**A FIRST SAMPLE PROGRAM**

1. Open command prompt
2. Change the directory to bin folder.
3. Type notepad first.java
4. Click on Yes and type the following code in it.

/\* this is the first java program \*/

class first

{

public static void main(String args[])

{

System.out.println(“Welcome to Java”);

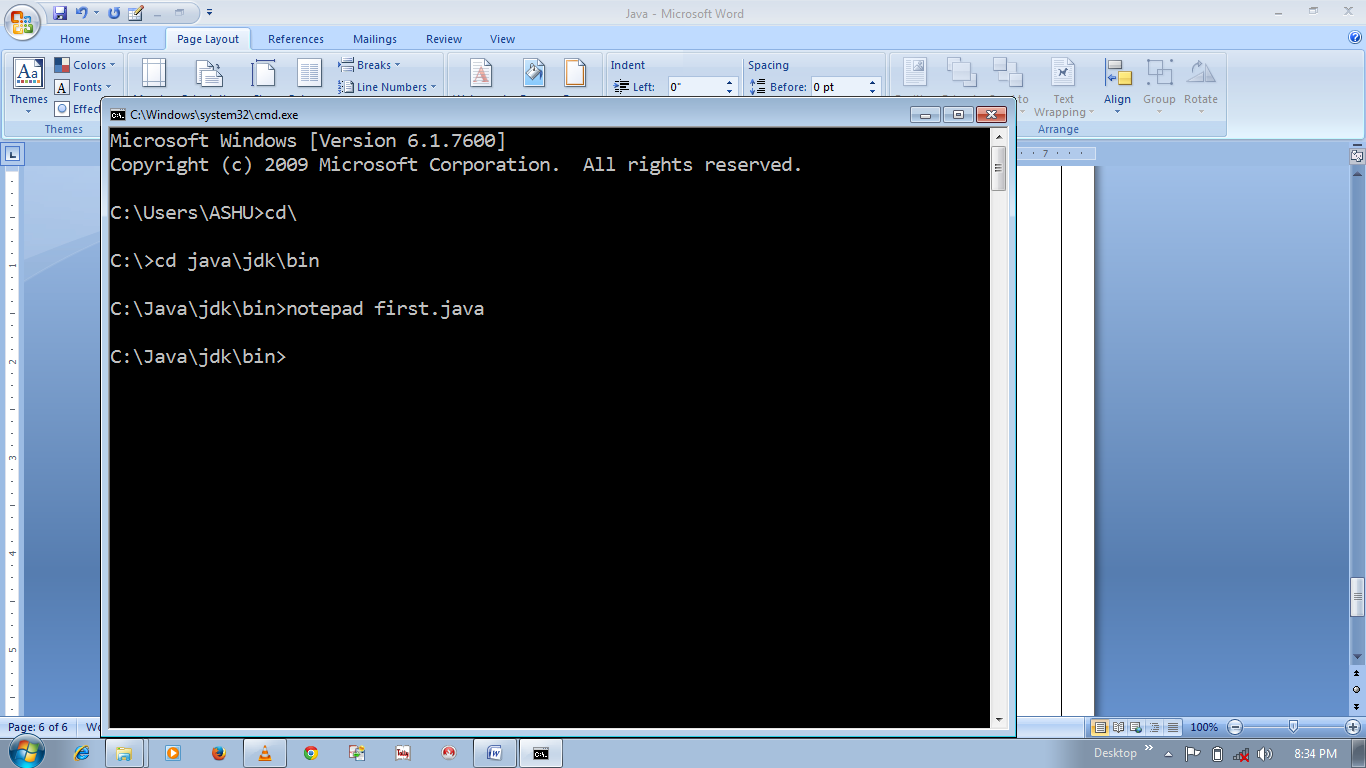
}

}

**Compiling the program**

The name of java compiler is javac.

Go to command prompt. A screen will appear like this.



Type javac first.java and press Enter.

**Running the program**

Type java first and press Enter.

In java, a source file is officially called a compilation unit. It is a text file that contains one or more class definitions. The java compiler requires that a source file use the .java filename extension.

In java all code must reside inside a class. By convention, the name of the class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches class name. The reason for this that java is case sensitive.

The line of code is shown here:

public static void main(String args[])

The **public** keyword is an access specifier, which allows the programmer to control the visibility of the class member. When the class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.

The keyword **static** allows main() to be called without having to instantiate a particular instance of the class. This is necessary since main() is called by the java interpreter before any object are made.

The keyword **void** simply tells the compiler that main() does not return a value.

Any information that you need to pass to a method is received by variable specified within the set of parenthesis that follow the name of the method. These variables are called parameters. In main(), there is only one parameter, **String args[]**, declares a parameter named **args**, which is an array of instances of the class **String**. Objects of type **String** store character strings. In this case, **args** receives any command line arguments present when the program is executed.

The line of code:

System.out.println

**System** is a pre-defined class that provides access to the system.

**out** is the output stream that is connected to the console**.**

**println** displays the string which is passed to it.

**OBJECT ORIENTED APPROACH**

In this approach, a program is a collection of members but enclosed under a special wrapper or container known as class, which gives the security for the contained that is defined in it.

A class is a user defined data type much like structure what we came across in procedural approach, but in procedural languages this structure can be defined only with variables, whereas a class can defined both with variables and structures.

**ARRAYS**

An array is a set of similar types of values, which are stored in a sequential order.

Java supports different styles of arrangements to an array:

1. One - Dimensional
2. Two – Dimensional
3. Jagged Arrays

An array index starts with zero (0) that means first item of an array will be stored on 0th position and the position of last item of an array will be total no. of items – 1.

In Java, arrays can be declared as fixed length or dynamic.

Fixed length array can store pre – defined number of items, while size of dynamic arrays increases as you add new items to the array.

**One Dimensional Array**

A one – dimensional array is essentially a list of like typed variables.

The general form of one –dimensional array is:

type var\_name[];

Here, **type** declares the base type of the array. The base type of array determines what type of data the array will hold.

e.g. int month\_day[];

Although this declaration establishes the fact that month\_day is an array variable, no array actually exists. In fact, the value of month\_day is set to null, which represents an array with no value. To link month\_day with an actual physical array of integers we must allocate them using the **new** operator. **new** is the special operator that allocates the memory.

The general form of **new** as it applies to one – dimensional array appears as follows:

type array\_var = new type [size];

Here **size** specifies the number of elements in an array. This is also known as dimension.

e.g. int month\_day = new int [5];

**// program to demonstrate one dimensional array**

class onedimarray

{

public static void main(String args[])

{

int month\_days[]=new int[5];

month\_days[0]=29;

month\_days[1]=35;

month\_days[2]=40;

month\_days[3]=45;

month\_days[4]=50;

for(int i=0;i<month\_days.length;i++)

{

System.out.println(month\_days[i]);

}

}

}

In the above program, we have used

month\_days.length;

Here, **length** is a property which is used to calculate the length of the array.

**// program to calculate the average of the marks**

class average

{

public static void main(String args[])

{

int marks[]=new int[5];

int sum=0,avg;

marks[0]=45;

marks[1]=57;

marks[2]=75;

marks[3]=67;

marks[4]=72;

for(int i=0;i<marks.length;i++)

{

sum=sum+marks[i];

}

avg=sum/marks.length;

System.out.println(“Sum is “+sum);

System.out.println(“Average is “+avg);

}

}

**STATIC AND NON-STATIC VARIABLES AND BLOCKS**

Static variables and static blocks will be the first part of the program being executed. It will get execute in the order in which they get defined.

After static block, non – static variables and non – static blocks will get executed.

First, non – static blocks will get executed. But it will execute if and only if the constructors are defined in the program. If the constructors will not get defined in the program then the static block will not get executed.

Non – static blocks will get executed before the execution of the constructor.

After the execution of non – static blocks, non – static variables will get executed.

class test

{

test()

{

System.out.println("Constructor of the class");

}

int x = m1();

int m1()

{

System.out.println("Non-Static Block m1()");

return 10;

}

{

System.out.println("Non-Static Block 1");

}

static

{

System.out.println("Static Block");

}

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

**{**

**System.out.println("In main()");**

**test t = new test();**

**System.out.println("x = " + t.x);**

**System.out.println("y = " + t.y);**

**}**

{

System.out.println("Non-Static Block 2");

}

int y = m2();

int m2()

{

System.out.println("Non-Static Block m2()");

return 20;

}

}

**Output:-**

Static Block

In main()

Non-Static Block m1()

Non-Static Block 1

Non-Static Block 2

Non-Static Block m2()

Constructor of the class

x = 10

y = 20

But if any non-static block is defined in main(), then it will execute according to the order in which it is defined.

class Example

{

static int a = m1();

static int m1()

{

System.out.println("Static Method");

return a;

}

static

{

System.out.println("Static Block");

}

int m2()

{

System.out.println("Non Static Method");

return 0;

}

{

System.out.println("Non Static Block");

}

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

**{**

**Example e = new Example();**

**e.m2();**

**{**

**System.out.println("Non Static Block in main()");**

**}**

**}**

}

Output:

Static Method

Static Block

Non Static Block

Non Static Method

Non Static Method in main()

As we can see from above output, **Non static Method in main()** is being executed in the last because it is defined as the last statement in the main() function.

**MAIN FUNCTION CONCEPT**

In case of inheritance, if main() function is defined in both base class and derived class, then only the derived class main() function will get executed.

JVM will search main() function from sub class to super class.

If JVM finds the main() function in sub class then it will not search it in super class.

If JVM does not finds the main() function in sub class then it will search main() function in super class, if finds then executes the main() function of super class.

If JVM does not finds main() function in both sub class and super class, then it will throws an exception **java.lang.NoSuchMethodError : main**

class Example

{

static int a = m1();

static int m1()

{

System.out.println("A:a");

return 10;

}

static

{

System.out.println("A class is loaded");

}

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

**{**

**System.out.println("A main()");**

**}**

}

class Example1 extends Example

{

static int b = m2();

static int m2()

{

System.out.println("B:b");

return 20;

}

static

{

System.out.println("B class is loaded");

}

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

**{**

**System.out.println("B main()");**

**System.out.println("B main a : " + a);**

**System.out.println("B main b : " + b);**

**}**

}

Output:

A:a

A class is loaded

B:b

B class is loaded

B main

B main a: 10

B main b: 20

**EXECUTION FLOW OF NON STATIC BLOCK IN INHERITANCE**

In case of non – static block and non – static variables, at first the statements before object creation in main() function will get executed. Then constructor call will take place. But constructor will get execute after the execution of parent class non – static block. After parent class non – static block will execute, then parent class constructor will execute. Then control comes to child class and non – static block of child class will be execute and then child class constructor will execute. Then the statements after object creation in main() function will execute.

class Example

{

int x = 10;

{

System.out.println("Example Non Static Block");

System.out.println("x = " + x);

}

Example()

{

System.out.println("Example Constructor");

x = 5;

}

}

class Example1 extends Example

{

int y = 20;

{

System.out.println("Example1 Non Static Block");

System.out.println("x = " + x);

System.out.println("y = " + y);

}

Example1()

{

System.out.println("Example1 Constructor");

y = 6;

}

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

**{**

**System.out.println("Example1 main");**

**Example1 e = new Example1();**

**System.out.println("x = " + e.x);**

**System.out.println("y = " + e.y);**

**}**

}

Output:

Example1 main

Example Non Static Block

x = 10

Example Constructor

Example1 Non Static Block

x = 5

y = 20

Example1 Constructor

x = 5

y = 6

**PROGRAM HAVING COMBINATION OF STATIC AND NON STATIC BLOCK**

class Example

{

static int a = m1();

static

{

System.out.println("Static Block of Example");

}

int x = m2();

{

System.out.println("Non Static Block of Example");

}

Example()

{

System.out.println("Constructor of class Example");

}

static int m1()

{

System.out.println("Static Function m1() of class Example");

return 10;

}

int m2()

{

System.out.println("Non Static Function m2() of class Example");

return 20;

}

void abc()

{

System.out.println("Non Static abc() of class Example");

}

void bbc()

{

System.out.println("Non Static bbc() of class Example");

}

}

class Example1 extends Example

{

static int b = m3();

static

{

System.out.println("Static Block of Class Example1");

}

int y = m4();

{

System.out.println("Non Static Block of class Example1");

}

Example1()

{

System.out.println("Constructor of class Example1");

}

static int m3()

{

System.out.println("Static Function m3() of class Example1");

return 30;

}

int m4()

{

System.out.println("Non Static Function m4() of class Example1");

return 40;

}

void abc()

{

System.out.println("Non Static abc() of class Example1");

}

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

**{**

**System.out.println("Example1 main");**

**Example1 e = new Example1();**

**e.abc();**

**e.bbc();**

**}**

}

Output:

Static Function m1() of class Example

Static block of Example

Static Function m3() of class Example1

Static Block of Class Example1

Example1 main

Non Static Function m2() of class Example

Non Static block of Example

Constructor of class Example

Non Static Function m4() of class Example1

Non Static block of class Example1

Constructor of class Example1

Non Static abc() of class Example1

Non Static bbc() of class Example

**CONSTRUCTOR EXECUTION FLOW**

If non – parameterized constructors are being called from the main() function, then at first non – parameterized parent class constructor and then child class constructor will be execute.

If parameterized constructor is being called from the main() function then parent class non – parameterized constructor and then parameterized child class constructor will be execute.

class Example

{

Example()

{

System.out.println("No arg constructor of class Example");

}

Example(int x)

{

System.out.println("Paremetrized Constructor of class Example");

}

}

class Example1 extends Example

{

Example1()

{

System.out.println("No arg constructor of class Example1");

}

Example1(int b)

{

System.out.println("Parametrized construcotr of class Example1");

}

public static void main(String args[])

{

Example1 e = new Example1();

Example1 e1 = new Example1(10);

}

}

Output:

No arg constructor of class Example

No arg constructor of class Example1

No arg constructor of class Example

Parameterized constructor of class Example1

**SUPER KEYWORD**

The super keyword is used to access super class members and constructors from subclass members and constructors. It has two syntaxes:

1. super() : is used to call super class constructors from subclass constructors
2. super. : is used to call superclass variables and methods from subclass members and constructors.

**Understanding super() working functionality**

super() is used to invoke super class constructor from sub class constructor to provide memory for super class fields in sub class object and further to initialize them with the initialization logic given by super class developer.

public class Object

{

public Object()

{

}

}

class Sample extends Example

{

int x = 50;

int y = 60;

Sample()

{ 3

super();

System.out.println(“Sample No arg”);

}

public static void main(String args[])

{

Sample s = new Sample();

1 2

}

}

10

11

class Example extends Object

{

int x = 10;

int y = 20; 7

Example() 4

{

super();

System.out.println(“Example No – arg”);

}

}

6

5

13

9

8

12

**ABOVE DIAGRAM SHOWS CONTROL FLOW OF SUB CLASS OBJECT CREATION**

**Who does place super() in all constructor?**

Compiler places super() call in all constructor at the time of compilation in case if no explicit super() or this() call is placed by developer.

**Rules to use super()**

It must be placed only in constructor as the first statement.

Else it leads to CE (compilation Error): “**call to super must be the first statement in constructor”.**

**Why it should be the first statement in the sub class constructor**

Because to send control to super class for identifying and providing memory for non – static variables before sub – class variables identification and initialization.

**Invoking super class constructor does it mean creating an object of super class**

No. super class object is not created when sub – class object is created, instead its non – static variables memory is provided in sub – class object.

**Super class parameterized constructor cannot be initialized from subclass non – parameterized and parameterized constructor.**

class Example1

{

Example1()

{

System.out.println("No arg Constructor of base class");

}

Example1(int a)

{

System.out.println("Parameterized Constructor of base class");

}

}

class Example extends Example1

{

Example()

{

System.out.println("Sub Class no arg Constructor");

}

Example(int a)

{

System.out.println("Sub Class Parameterized Constructor");

}

public static void main(String args[])

{

Example e = new Example();

Example e1 = new Example(10);

}

}

**Output:**

No arg constructor of the base class

Sub class no arg constructor

No arg constructor of the base class

Sub class Parameterized constructor

From the output we can easily conclude that super class parameterized constructor cannot be initialized from sub class non parameterized constructor.

**How can we initialize super class parameterized constructor from sub class?**

We must place super() call explicitly with argument in the sub class constructor. The argument type should be constructor parameter type.

class Example1

{

Example1()

{

System.out.println("No arg Constructor of base class");

}

Example1(int a)

{

System.out.println("Parameterized Constructor of base class");

}

}

class Example extends Example1

{

Example()

{

System.out.println("Sub Class no arg Constructor");

}

Example(int a)

{

super(10);

System.out.println("Sub Class Parameterized Constructor");

}

public static void main(String args[])

{

Example e = new Example();

Example e1 = new Example(10);

}

}

**Output:**

No arg constructor of the base class

Sub class no arg constructor

Parameterized constructor of the base class

Sub class Parameterized constructor

**When should developer place super() call explicitly?**

Developer should place super() call explicitly in the below two cases:

1. If super class does not no argument constructor.
2. If developer wants to initialized super class non – static variables with parameterized constructor.

**Rules to use super keyword**

1. super() must be placed only in constructors as the first statement.
2. super. cannot be referenced from static block

class Example

{

static int a = 10;

int x = 20;

static void m1()

{

System.out.println("Static function m1() of class Example");

}

void m2()

{

System.out.println("Non Static function m2() of class Example");

}

void m3()

{

System.out.println("Non Static function m3() of class Example");

}

}

class Example1 extends Example

{

static int a = 50;

int x = 60;

static void m1()

{

//super.m1(); //Error:Non Static Variable super Cannot be

//referenced from a static context

System.out.println("Static function m1() of class Example1");

}

void m2()

{

super.m2();

System.out.println("Non Static function m2() of class Example1");

}

void m4()

{

System.out.println("super.a : " + super.a);

System.out.println("a : " + a);

super.m1();

m1();

System.out.println("super.x : " + super.x);

System.out.println("x : " + x);

super.m2();

m2();

m3();

}

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

**{**

**Example1 e = new Example1();**

**e.m4();**

**}**

}

Output:

super.a : 10

a : 50

Static function m1() of class Example

Static function m1() of class Example1

super.x : 20

x : 60

Non Static function m2() of class Example

Non Static function m2() of class Example

Non Static function m2() of class Example1

Non Static function m3() of class Example

**Reference Type Conversions**

Like primitive types, we can also perform type conversion operation in between reference types. Assigning one class object to another class references variable is called references type conversion. To perform references type conversion two classes should be compatible. The classes becomes compatible only if they are developed with inheritance. Inheritance relation is also called IS-A relation.

**Rule to check in reference type conversion is:**

Source type IS A destination type.

class SourceDestCheck

{

public static void main(String[] args)

{

Object o = new SourceDestCheck();

System.out.println(o instanceof SourceDestCheck); // true

}

}

**In the above program we are testing which is source type and which is destination type.**

**Source type --> o**

**Destination type --> SourceDestCheck**

The class that is placed after extends keyword is called super class and the class that is placed before extends keyword is called sub class. **The classes created without inheritance relationship is called siblings.**

**Which are the classes called compatible?**

* Sub class is compatible with super class
* Super class is not compatible with sub class.
* Siblings are not compatible.

**What is the right assignment between referenced types?**

The rule we should check in referenced type conversion is Source type IS A Destination type or not. So we can assign subclass object reference to a super class referenced variable. But super class object reference cannot be assigned to a sub class referenced variable. Also sibling object reference cannot be assigned to another sibling class referenced variable.

**What type of referenced variable we must create to store all types of objects?**

java.lang.Object class type, because it is the super class of all types of classes.

Object

A

D

C

B

Object class is the super class of all user defined and pre - defined classes.

Class A “IS A” subclass of Object

Class B “IS A” subclass of A, Object

Class C “IS A” subclass of B, A, Object

Class D “IS A” subclass of Object, and is sibling of A, B, C

A class object reference in Object class reference variable

B class object reference in A or Object class reference variable

C class object reference in B or A or Object class reference variable

D class object reference in Object class reference

So, we can store

**class A{}**

**class B extends A{}**

**class C extends B{}**

**class D{}**

**class ReferenceTypeConversion**

**{**

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

**{**

**Object obj1 = new Object();**

**Object obj2 = new A();**

**Object obj3 = new B();**

**Object obj4 = new C();**

**Object obj5 = new D();**

**A a1 = new A();**

**A a2 = new B();**

**A a3 = new C();**

**// A a4 = new D(); Incompatible types**

**// B b1 = new A(); Incompatible types**

**B b2 = new B();**

**B b3 = new C();**

**// B b4 = new D(); Incompatible types**

**}**

**}**

**Types of referenced type conversion**

Java supports two types of reference type conversions:

* Up casting / automatic conversion
* Down casting / casting

**Up casting:** It is the implicit reference type conversion.

The process of storing sub class object reference into super class referenced variable is called up casting.

e.g.

A a = new B();

**Note:** It is not possible to store super class object reference in sub class referenced variable. It leads to **incompatible types** compile time error.

e.g.

A a = new A(); // Valid

B b = a; // Invalid

B b = new A(); // Invalid

**Down Casting:** It is the explicit reference type conversion, casting.

Retrieving sub class object reference from super class reference variable and storing it in the same sub class object reference is called down casting.

e.g.

A a = new A(); // Valid

B b = (B) a; // Valid

Down casting (explicit reference type conversion) is same as explicit primitive type conversion.

e.g.

float a = 10.5f;

int c = a;

System.out.println(c); // Error: Possibly loss of precision

float a = 10.5f;

int c = (int)a;

System.out.println(c); // No Error

**Rule in using cast operator**

The cast operator type and source type should have inheritance relation else it leads to compile time error **inconvertible types.**

class A{}

class D{}

class InconvertibleTypeConversion

{

public static void main(String[] args)

{

A a = new A();

D d = (D) a;

}

}

**java.lang.ClassCastException**

In casting the object coming from source variable, if it is not compatible with cast operator type JVM throws exception (run – time error) **ClassCastException.**

Storing super class reference variable to sub class reference variable can be performed by down casting. The compile will compile successfully because compiler will only checks for IS A relation between source type variable and cast operator type. But JVM checks for the object reference of reference variable. If object reference of reference variable is different from object reference of class in which reference variable is assigning, then JVM throws **ClassCastException.**

class A{}

class B extends A{}

class ClassCastExceptionDemo

{

public static void main(String[] args)

{

A a1 = new A();

A a2 = new B();

B b1 = new B();

B b2 = (B) a1; // RError: ClassCastExceptionError : A

cannot be cast to B

B b3 = (B) a2; // No Error

}

}

In the above example, the statement

**B b2 = (B) a1;**

throws an exception **ClassCastExceptionError : A cannot be cast to B** because **a1** is the reference variable having object reference of **class A**. But in the above statement we are trying to store **a1** in **class B**. That means here object reference of **a1** is different from the object reference of class in which we are trying to store it. So JVM throws an exception.

The statement,

**B b3 = (B) a2;**

does not gives any error because object reference of **a1** is of **class B** and we are storing it in the **class B** only. So, the casting is possible.

**How can we solve ClassCastException?**

To solve ClassCastException we should use **instanceof** operator.

It returns Boolean value by checking source type object with given class.

**Syntax:**

Referenced variable instanceof classname

Here, referenced variable is the source variable and class name is the cast operator type name. It returns true, if the referenced variable contains object IS A class type. Else returns false.

In the statement,

B b3 = (B) a2;

**Source variable 🡪 a2**

**Cast operator 🡪 B**

class InstanceOfDemo

{

public static void main(String[] args)

{

A a1 = new A();

A a2 = new B();

B b1 = new B();

System.out.println("a1 instanceof B:" +(a1 instanceof B));//false

System.out.println("a2 instanceof B:" +(a2 instanceof B));//true

if(a1 instanceof B)

{

B b2 = (B) a1;

}

B b3 = (B) a2;

}

}

**Rule of instanceof operator**

Source variable type and class name type should have IS A relation else compiler throws compile time error **inconvertible types**. If the object coming from referenced variable is same class or sub class object then instanceof operator returns true. If it is super class or sibling type object instanceof operator returns false.

class InstanceOfDemo1

{

public static void main(String[] args)

{

A a = new B();

System.out.println(a instanceof Object); // true

System.out.println(a instanceof B); // true

System.out.println(a instanceof C); // false

System.out.println(a instanceof D); // Inconvertible types

Object obj = new B();

System.out.println(a instanceof Object); // true

System.out.println(a instanceof A); // true

System.out.println(a instanceof B); // true

System.out.println(a instanceof C); // false

System.out.println(obj instanceof B); // true

}

}

The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

class CastParent

{

int a = 10;

}

class CastChild1 extends CastParent

{

int a = 30;

public static void main(String[] args)

{

CastParent c1 = new CastParent();

CastChild1 c2 = new CastChild1();

CastParent c3 = c2;

System.out.println("c1 = " + c1);

System.out.println("c2 = " + c2);

System.out.println("c3 = " + c3);

System.out.println("c1.a = " + c1.a); // 10

System.out.println("c2.a = " + c2.a); // 30

System.out.println("c3.a = " + c3.a); // 10

}

}

Output:

c1 = CastParent@19821f

c2 = CastChild1@addbf1

c3 = CastChild1@addbf1

c1.a = 10

c2.a = 30

c3.a = 10

From the above output we can analyze that the **address of c2 and c3 are same**. But when we are trying to access the value of **a,** from **c2** and **c3** then **c2 access the value from child class** but **c3 access the value from parent class.**

It is so because c2 is the reference variable of class CastChild and c3 is the reference variable of class Castparent and it is the type of reference variable which determines what members can be accessed.

**In the above example, if we change the value of c3.a, then it will not affect the value of c2.a, because c3 access the variable of parent class and c2 access the variable of child child.**

c3.a = 40;

System.out.println("c3.a = " + c3.a); // 40

System.out.println("c1.a = " + c1.a); // 10

System.out.println("c2.a = " + c2.a); // 30

**This rule will be applicable on both up casting and down casting.**

CastChild1 c4 = (CastChild1)c3;

System.out.println("c4.a = " + c4.a); // 30

c4.a = 20;

System.out.println("c4.a = " + c4.a); // 20

System.out.println("c3.a = " + c3.a); // 10

System.out.println("c1.a = " + c1.a); // 10

**Dynamic Method Dispatch**

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

abstract class Dimension

{

double dim1;

double dim2;

Dimension(double a,double b)

{

dim1 = a;

dim2 = b;

}

abstract double area();

}

class Rectangle extends Dimension

{

Rectangle(double a,double b)

{

super(a,b);

}

double area()

{

System.out.print("Area of rectangle : ");

return dim1\*dim2;

}

}

class Triangle extends Dimension

{

Triangle(double a,double b)

{

super(a,b);

}

double area()

{

System.out.print("Area of triangle : ");

return (dim1\*dim2)/2;

}

}

class DynamicMethodDispatch

{

static void assign(Dimension d)

{

System.out.println(d.area());

}

public static void main(String[] args)

{

assign(new Rectangle(10,10));

assign(new Triangle(10,50));

}

}

**What will be the output of the following program?**

class A

{

static int a = 10;

int x = 20;

}

class B extends A

{

static int a = 30;

int x = 40;

}

class Example

{

public static void main(String args[])

{

B b1 = new B();

A a1 = new B();

System.out.println("b1.a = " + b1.a); // 30

System.out.println("b1.x = " + b1.x); // 40

System.out.println("a1.a = " + a1.a); // 10

System.out.println("a1.x = " + a1.x); // 20

System.out.println("B.a = " + B.a); // 30

System.out.println("A.a = " + A.a); // 10

b1.a = 5;

a1.a = 6;

System.out.println("b1.a = " + b1.a); // 5

System.out.println("b1.x = " + b1.x); // 40

System.out.println("a1.a = " + a1.a); // 6

System.out.println("a1.x = " + a1.x); // 20

System.out.println("B.a = " + B.a); // 5

System.out.println("A.a = " + A.a); // 6

}

}

**What will be the output of the following program?**

class A

{

static int a = 10;

int x = 20;

}

class B extends A

{

static int a = 30;

int x = 40;

}

class Example

{

public static void main(String args[])

{

B b = new B();

A a = b;

System.out.println("b.a = " + b.a); // 30

System.out.println("a.a = " + a.a); // 10

System.out.println("b.x = " + b.x); // 40

System.out.println("a.x = " + a.x); // 20

}

}

**What will be the output of the following program?**

class A

{

int x = 10; //70

int y = 20; //80

void m1()

{

System.out.println("A x : " + x);

System.out.println("A y : " + y);

}

}

class B extends A

{

int x = 30; //50

int y = 40; //60

void m2()

{

System.out.println("B x : " + x);

System.out.println("B y : " + y);

System.out.println("super.x : " + super.x);

System.out.println("super.y : " + super.y);

}

void m3()

{

x = 50;

y = 60;

super.x = 70;

super.y = 80;

}

}

class Example

{

public static void main(String args[])

{

B b1 = new B();

A a1 = b1;

B b2 = new B();

A a2 = b2;

b1.m3();

System.out.println("b1.x : " + b1.x); // 50

System.out.println("b1.y : " + b1.y); // 60

System.out.println("a1.x : " + a1.x); // 70

System.out.println("a1.y : " + a1.y); // 80

System.out.println("b2.x : " + b2.x); // 30

System.out.println("b2.y : " + b2.y); // 40

System.out.println("a2.x : " + a2.x); // 10

System.out.println("a2.y : " + a2.y); // 20

System.out.println();

b1.m1();

b2.m1();

System.out.println();

b1.m2();

b2.m2();

}

}

Output:

A x : 70

A y : 80

A x : 10

A y : 20

B x : 50

B y : 60

super.x : 70

super.y : 80

B x : 30

B y : 40

super.x : 10

super.y : 20