**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 things 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”);

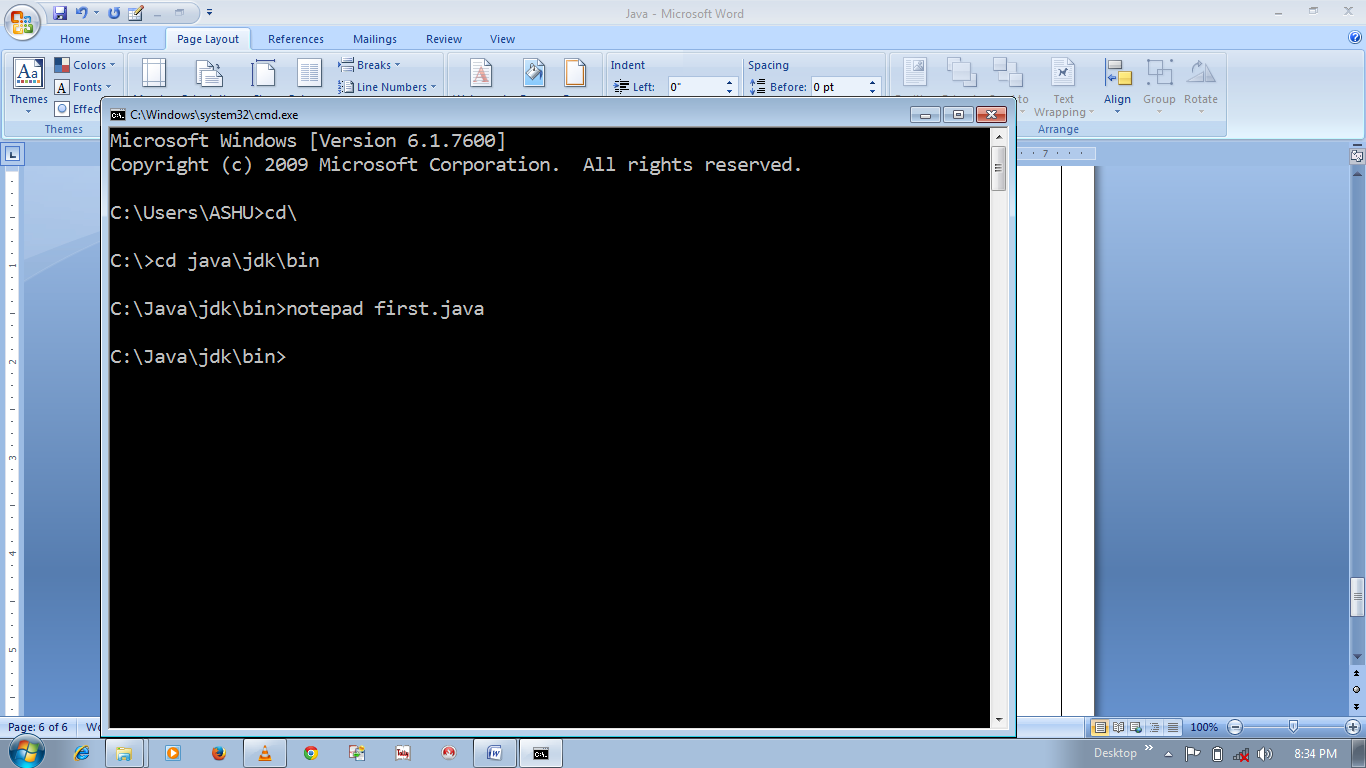
}

}

**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);

}

}

**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 **incompatible 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 compiler 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 **a2** 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));

}

}

**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 variables will get executed. Then non – static blocks 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 non - static block will not get executed.

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

After the execution of non – static variables, non – static blocks 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.

**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

**How many times a super class is loaded into JVM?**

A class is loaded into JVM only once. So a superclass is loaded into JVM only once when its first subclass is loaded into JVM. When we load its next subclass, superclass is not loaded again, only the current subclass is loaded.

**Method Hiding, Over-riding and Overloading**

* Redefining superclass static method in subclass with same prototype is called method hiding.
* Redefining superclass non – static method in subclass with same prototype is called method over – riding.
* Defining new method with existed method name but with different parameters, type, list, order is called method over – loading

**Program to demonstrate hiding, over – riding and over – loading**

class A

{

static void m1()

{

System.out.println("A m1(static)");

}

void m2()

{

System.out.println("A m2(non static)");

}

static void m3()

{

System.out.println("A m3(static)");

}

int m3(String s)

{

System.out.println("A m3(non static int)");

return 1;

}

}

class B extends A

{

static void m1()

{

System.out.println("B m1(static)");

}

void m2()

{

System.out.println("B m2(non static)");

}

void m3(float f,int x)

{

System.out.println("B m3(non static void)");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1();

b.m2();

b.m3();

b.m3("abc");

b.m3(45.3f,67);

A a = b;

a.m1();

a.m2();

a.m3();

a.m3("abc");

}

}

Output:

B m1(static)

B m2(non static)

A m3(static)

A m3(non static int)

B m3(non static void)

A m1(static)

B m2(non static)

A m3(static)

A m3(non static int)

**Method over – riding execution control flow**

In object oriented programming method, over – riding is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its super classes. The implementation in the subclass over – rides (replaces) the implementation in the superclass. So, overridden method is always executed from the object whose object is stored in referenced variable. Super class method is called overridden method, and subclass method is called overriding method.

**What is the output of the following program?**

class A

{

void m1()

{

System.out.println("A m1");

}

void m2()

{

System.out.println("A m2");

m1();

}

}

class B extends A

{

void m1()

{

System.out.println("B m1");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1();

b.m2();

A a = b;

a.m1();

a.m2();

}

}

Output:

B m1

A m2

B m1

B m1

A m2

B m1

**What is the output of the following program?**

class A

{

private void m1()

{

System.out.println("A m1");

}

void m2()

{

System.out.println("A m2");

m1();

}

}

class B extends A

{

void m1()

{

System.out.println("B m1");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1();

b.m2();

A a = b;

// a.m1(); error: m1 has private access in A

a.m2();

}

}

Output:

B m1

A m2

A m1

A m2

A m1

**In executing static method call we should not consider object type rather we must consider only referenced variable type.**

class A

{

static void m1()

{

System.out.println("A m1");

}

void m2()

{

System.out.println("A m2");

m1();

}

}

class B extends A

{

static void m1()

{

System.out.println("B m1");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1();

b.m2();

A a = b;

a.m1();

a.m2();

}

}

Output

B m1

A m2

A m1

A m1

A m2

A m1

**How can we execute superclass method if it is over ridden in subclass?**

Using super keyword.

class Sample

{

void m1()

{

System.out.println("Sample m1");

}

void m2()

{

System.out.println("Sample m2");

}

void m3()

{

System.out.println("Sample m3");

}

}

class Example extends Sample

{

void m1()

{

System.out.println("Example m1");

}

void m2()

{

super.m2();

System.out.println("Example m2");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2();

e.m3();

}

}

Output

Example m1

Sample m2

Example m2

Sample m3

**How can we execute superclass overridden static method from subclass?**

Using class name

class Sample

{

static void m1()

{

System.out.println("Sample m1");

}

}

class Example extends Sample

{

static void m1()

{

Sample.m1();

System.out.println("Example m1");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

}

}

Output

Sample m1

Example m1

**Can we override main method in subclass?**

We can override main method in subclass, and it is only executed from the subclass. To execute superclass main method, we must call it explicitly from subclass.

class Sample

{

public static void main(String args[])

{

System.out.println("Sample main");

}

}

class Example extends Sample

{

public static void main(String args[])

{

Sample.main(new String[0]);

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

}

}

Output

Sample main

Example main

**What is the output of the below program?**

class A

{

void m1()

{

System.out.println("A m1");

}

void m2()

{

System.out.println("A m2");

m1();

}

}

class B extends A

{

void m1()

{

System.out.println("B m1");

}

void m3()

{

System.out.println("B m3");

m1();

super.m2();

}

}

class C extends B

{

void m2()

{

System.out.println("C m2");

//m4();

}

}

class Example extends C

{

void m1()

{

System.out.println("Example m1");

}

void m2()

{

System.out.println("Example m2");

}

void m4()

{

System.out.println("Example m4");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2();

e.m3();

e.m4();

}

}

Output

Example m1

Example m2

B m3

Example m1

A m2

Example m1

Example m4

**What is the output of the following program?**

class A

{

static void m1()

{

System.out.println("A m1");

}

static void m2()

{

System.out.println("A m2");

m1();

}

}

class B extends A

{

static void m1()

{

System.out.println("B m1");

}

void m3()

{

System.out.println("B m3");

m1();

super.m2();

}

}

class C extends B

{

static void m2()

{

System.out.println("C m2");

//m4();

}

}

class Example extends C

{

static void m1()

{

System.out.println("Example m1");

}

static void m2()

{

System.out.println("Example m2");

}

static void m4()

{

System.out.println("Example m4");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2();

e.m3();

e.m4();

}

}

**Output**

Example m1

Example m2

B m3

B m1

A m2

A m1

Example m4

**What is the output of the following program?**

class A

{

static int a = 10;

int x = 20;

static void m1()

{

System.out.println("A m1");

}

void m2()

{

System.out.println("A m2");

}

void m3()

{

System.out.println("A m3");

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

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

m1();

m2();

}

}

class B extends A

{

static int a = 50;

int x = 60;

static void m1()

{

System.out.println("B m1");

}

void m2()

{

System.out.println("B m2");

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

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

}

void m4()

{

super.a = a - 10;

super.x = x - 10;

}

}

class Example

{

public static void main(String args[])

{

B b1 = new B();

B b2 = new B();

A a1 = new B();

b1.a = 15;

b1.x = 16;

b2.a = 18;

b2.x = 19;

b1.m4();

b2.m4();

b1.m3();

System.out.println();

b2.m3();

System.out.println();

System.out.println("b1.a : " + b1.a);

System.out.println("a1.a : " + a1.a);

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

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

}

}

**Output**

A m3

A a : 8

A x : 6

A m1

B m2

B a : 18

B x : 16

A m3

A a : 8

A x : 9

A m1

B m2

B a : 18

B x : 19

b1.a : 18

a1.a : 8

b1.x : 16

a1.x : 20

**Can we overload methods in same class?**

Yes, it is possible to overload methods in the same class or in superclass and subclass, because overloaded methods are different methods. But we cannot override methods in same class; it leads to CE “method is already defined” because overriding methods are same, methods with different implementations.

**Overloaded methods invocation control flow**

Compiler always checks for the called method definition in referenced variable type class with the given argument’s type parameters. So in searching and executing a method definition we must consider both referenced variable type and argument type.

1. Referenced variable type for deciding from which class method should be bind
2. Argument type for deciding which overloaded method should be bind.

For example,

B b = new B();  
 A a = new B();

b.m1(50); => b.m1(int);

in this method call, we should search m1() method definition in class B with int

parameter in compilation.

a.m1(50) => a.m1(int);  
 in this method call, we should search m1() method definition in class A with int

parameter, not in class B even though object is of class B.

JVM always executes the called method definition from the class which is linked by the compiler, if it is not overridden in current object class. If it is overridden method then only it is executed from current objects subclass.

This means:

1. In case b.m1(50) method class, we should execute m1(int) method from class B because referenced variable type and object type are both of same class B.
2. But in case of a.m1(50) method call, we should execute m1(int) method from class B if it is overriding in class B else it means it is overloading or static we should execute it from class A because referenced variable type is of class A.

**Note**: private methods are not overridden methods because they are not inherited to subclass. So private methods are executed from referenced variable type class.

Also static methods are not overridden methods because they are hidden methods as they do not need object for executing their logic, so static, methods are also executed from referenced variable type class.

It means

1. If m1(int) is static method and not overridden method, overloaded method is executed from class A.
2. If m1(int) is a non – static method and if it is overridden in class B, it is executed from class B not from class A.

**4 points to be considered to link and execute overloaded methods**

1. Overloading method executed based on given argument types.
2. Widening, Auto – boxing, var – arg: if the given argument type parameter is not found, then compiler search for widening type, auto – boxing type, var – arg type parameter method of the given argument.
3. Ambiguous error: When same argument type parameter method is not found, if the parameter of the two overloaded methods are matched with given argument type then we get ambiguous error.
4. Method overloading with inheritance: With the given argument type parameter method we should first search in current reference variable’s class, if not found then we should search it in next parent class, grand – parent class till root class object, if it is not found we must repeat the searching with widening, auto – boxing, var- arg. If this searching also failed then should throw CE: cannot find symbol.
5. **Overloading method executed based on given argument types**

class Example

{

void m1(int a)

{

System.out.println("m1 int arg");

}

void m1(char ch)

{

System.out.println("m1 char arg");

}

public static void main(String args[])

{

Example e = new Example();

e.m1(99);

e.m1('c');

e.m1((char)100);

e.m1((int)'d');

System.out.println();

int i1 = 97;

int i2 = 'a';

char ch1 = 98;

char ch2 = 'b';

e.m1(i1);

e.m1(i2);

e.m1(ch1);

e.m1(ch2);

System.out.println();

e.m1((char)i1);

e.m1((int)ch1);

System.out.println();

e.m1(i1 + i2);

e.m1(ch1 + ch2);

System.out.println();

e.m1(10 + 'a');

e.m1('a' + 'b');

e.m1((char)('a' + 'b'));

e.m1((char)'a' + 'b');

System.out.println();

e.m1(m2());

e.m1(m3());

}

static int m2()

{

return 'a';

}

static char m3()

{

return 97;

}

}

Output

m1 int arg

m1 char arg

m1 char arg

m1 int arg

m1 int arg

m1 int arg

m1 char arg

m1 char arg

m1 char arg

m1 int arg

m1 int arg

m1 int arg

m1 int arg

m1 int arg

m1 char rg

m1 int arg

m1 int arg

m1 char arg

**We can also overload method with reference types**

class A

{

void foo(Object o)

{

System.out.println("Object Parameter");

}

void foo(String s)

{

System.out.println("String parameter");

}

void foo(Integer i)

{

System.out.println("Integer Parameter");

}

}

class Example

{

public static void main(String args[])

{

A a = new A();

Object obj1 = new Object();

Object obj2 = "Ashu";

Object obj3 = new Integer(10);

a.foo(obj1);

a.foo(obj2);

a.foo(obj3);

System.out.println();

a.foo((String)obj2);

a.foo((Integer)obj3);

System.out.println();

//a.foo((String)obj1);

//a.foo((Integer)obj1);

//a.foo((String)obj3);

//a.foo((Integer)obj2);

}

}

Output

Object Parameter

String parameter

Integer Parameter

String parameter

Integer Parameter

1. **Searching mechanism of compiler for overloaded methods**
2. First searches with widening type of the given argument.
3. Second searches with Auto Boxing type of the given argument.
4. Third searches with Var – arg type of the given argument.
5. At last, compiler throws CE: cannot find symbol

So the order of searching for the overloading method definition is

Same type > Widening > Auto Boxing > CE

**What is the output of the following program?**

class Sample

{

void m1(int i)

{

System.out.println("Integer Argument");

}

void m1(byte b)

{

System.out.println("Byte Argument");

}

}

class Example extends Sample

{

public static void main(String args[])

{

byte b = 10;

short s = 15;

char ch = 'a';

int i = 20;

Sample s1 = new Sample();

s1.m1(b);

s1.m1(s);

s1.m1(ch);

s1.m1(i);

s1.m1(10);

s1.m1(15);

s1.m1('a');;

s1.m1(20);

s1.m1((byte)10);

s1.m1(15);

s1.m1('a');

s1.m1(20);

}

}

Output

Byte Argument

Integer Argument

Integer Argument

Integer Argument

Integer Arguemnt

Integer Arguemnt

Integer Argument

Integer Arguemnt

Byte Arguement

Integer Argument

Integer Argument

Integer Argument

**Reference Type Widening**

When we invoke an overloaded method by passing an Object, first compiler checks for same argument type parameter method. If same parameter method is not available, then it search for its immediate superclass parameter type method. It repeats this searching till it found java.lang.Object class parameter. If no method is found with this passed object mathed parameter, compiler throws CE: cannot find symbol.

The referenced type parameter method can be called by passing parameter either

1. Same object type
2. Its subclass object or
3. Null

class Sample

{

static void m1(Sample e)

{

System.out.println("m1 method");

}

}

class Example extends Sample

{

public static void main(String args[])

{

m1(new Example());

m1(new Sample());

//m1(new Test());

//m1("abc");

m1(null);

}

}

Output

m1 method

m1 method

m1 method

**What should be the method parameter to accept all types of objects as argument?**

java.lang.Object class

1. **What does happened if the passed parameter matched parameter is not found and if it is matched with widening parameters with different order?**

It leads to CE: ambiguous error

class Sample

{

void m1(int i,float f)

{

System.out.println("Integer Float");

}

void m1(float f,int i)

{

System.out.println("Float Integer");

}

}

class Example

{

public static void mian(String args[])

{

Sample s = new Sample();

s.m1(10,4.3f);

s.m1(3.4f,10);

//s.m1(10,10);

}

}

In the above program, **e.m1 (10, 10)** method call leads to CE: ambiguous error, because the parameters of both methods are matched with this method call arguments, so compiler cannot decide which method definition must be bind for this method call, hence it throws CE.

**Three important cases with referenced type overloaded methods**

**Case 1:** If we overloaded methods with superclass and subclass types as parameter, and if the passed argument is matched, subclass parameter method is executed. In this case compiler and JVM give first priority to subclass method parameter method.

class A

{

}

class B extends A

{

}

class Example

{

void m1(A a)

{

System.out.println("class A m1");

}

void m1(B b)

{

System.out.println("class B m1");

}

public static void main(String args[])

{

Example e = new Example();

e.m1(new A());

e.m1(new B());

//e.m1("abc");

e.m1(null);

A a1 = new A();

A a2 = new B();

B b1 = new B();

A a3 = null;

B b2 = null;

e.m1(a1);

e.m1(a2);

e.m1(b1);

e.m1(a3);

e.m1(b2);

}

}

Output

class A m1

class B m1

class B m1

class A m1

class A m1

class B m1

class A m1

class B m1

**Case 2:** When a method is overload with siblings, and if passed argument is matched with both parameters, compiler throws CE: ambiguous error.

class A

{}

class B extends A

{}

class T

{}

class Example

{

void m1(A a)

{

System.out.println("class A m1");

}

void m1(T t)

{

System.out.println("class T m1");

}

public static void main(String args[])

{

Example e = new Example();

e.m1(new A());

e.m1(new B());

e.m1(new T());

//e.m1(null);

e.m1((B)null);

e.m1((T)null);

}

}

Output

class A m1

class A m1

class T m1

class A m1

class T m1

**Note:** **e.m1(null) will give error because it is matched with both Example e and Test t.**

**Case 3:** When a method is overloaded with sibling parameter along with superclass and subclass parameters, if we pass null directly it leads to CE: ambiguous error.

class A

{}

class B extends A

{}

class T

{}

class Example

{

void m1(Object o)

{

System.out.println("Object argument");

}

void m1(A a)

{

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

}

void m1(B b)

{

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

}

void m1(T t)

{

System.out.println("T argument");

}

public static void main(String args[])

{

Example e = new Example();

e.m1("abc");

e.m1(new A());

e.m1(new B());

e.m1(new T());

e.m1(new Object());

//e.m1(null);

e.m1((T)(null));

}

}

Output

Object argument

A argument

B argument

T argument

Object argument

T argument

**Note: e.m1(null) returns an error: reference to m1 is ambiguous, both method m1(B) in Example and method m1(T) in Example match.**

In the above program, if m1(Test t) definition is not available then:

1. m1(new Test()) method call binds with m1(Object)
2. m1(null) method call does not leads to CE, and is bind with m1(B b).

**What is output of the below programs?**

**Case 1:**

class Sample

{

void m1(String s)

{

System.out.println("String argument");

}

void m1(Test t)

{

System.out.println("Test argument");

}

}

class Example

{

public static void main(String args[])

{

Sample s = new Sample();

s.m1("abc");

s.m1(new Test());

s.m1(null);

}

}

Output

s.m1(null) gives an error

Error: reference to m1 is ambiguous, both method m1(String) in Sample and method m1(Test) in Sample match.

**Case 2:**

class Sample

{

void m1(Object obj)

{

System.out.println("Object argument");

}

void m1(Test t)

{

System.out.println("Test argument");

}

}

class Example

{

public static void main(String args[])

{

Sample s = new Sample();

s.m1("abc");

s.m1(new Test());

s.m1(null);

}

}

Output

Object argument

Test argument

Test argument

In this case **s.m1(null)** does not returns an error, because subclass parameter type method is bind for this method call.

1. **Method Overloading with inheritance**

Methods can be overloaded in same class or in superclass. In superclass if we have method with superclass method name but with different parameters type or list and order then subclass method is called overloading method not overriding method.

**Methods cannot be overridden in the same class, they should be overridden in the subclasses.**

**Execution control flow with inheritance**

1. First search for the method with give argument type in subclass, if not found search it in superclass with same argument type. If not found then
2. It searches with Widening, if not found then search with auto – boxing, if not found, then search with the var – arg in subclass first, if not found in subclass then search within superclass. If not found then
3. Finally compiler throws CE: cannot find symbol

**What is the output of the following programs?**

class Sample

{

void add(int a,int b)

{

System.out.println("Sample Integer Integer");

}

void add(String a,float b)

{

System.out.println("Sample String Float");

}

int add(String a,String b)

{

System.out.println("Sample String String");

return 0;

}

}

class Test extends Sample

{

void add(int x,int y)

{

System.out.println("Test Integer Integer");

}

float add(float a,int b)

{

System.out.println("Test Float Integer");

return a+b;

}

String add(String a,double b)

{

System.out.println("Test String Double");

return a;

}

}

class Example

{

public static void main(String args[])

{

Test t = new Test();

t.add(10,20);

t.add("abc",20);

t.add("abc","xyz");

t.add("10",20.0);

}

}

Output

Test Integer Integer

Sample String Float

Sample String String

Test String Double

class A

{

void m1(int a)

{

System.out.println("A Integer argument");

}

}

class B extends A

{

void m1(float f)

{

System.out.println("B Float argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1(50);

b.m1('a');

b.m1(50L);

System.out.println();

A a = new B();

a.m1(50);

a.m1('a');

//a.m1(50L);

}

}

Output

A Integer argument

A Integer argument

B Float argument

A Integer argument

A Integer argument

Note: a.m1(50L) returns an error: actual argument long cannot be converted to int by method invocation conversion.

class A

{

void m1(float f)

{

System.out.println("A Float argument");

}

}

class B extends A

{

void m1(int a)

{

System.out.println("B Integer argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1(50);

b.m1('a');

b.m1(50L);

System.out.println();

A a = new B();

a.m1(50);

a.m1('a');

a.m1(50L);

}

}

Output

B Integer argument

B Integer argument

A Float argument

A Float argument

A Float argument

A Float argument

class A

{

void m1(int a)

{

System.out.println("A Integer argument");

}

}

class B extends A

{

void m1(float f)

{

System.out.println("B Float argument");

}

void m1(char ch)

{

System.out.println("B char argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1(50);

b.m1('a');

b.m1(50L);

System.out.println();

A a = new B();

a.m1(50);

a.m1('a');

//a.m1(50L);

}

}

Output

A Integer argument

B char argument

B Float argument

A Integer argument

A Integer argument

Note: a.m1(50L) returns an error: actual argument long cannot be converted to int by method invocation conversion.

class A

{

void m1(int a)

{

System.out.println("A Integer argument");

}

void m1(char ch)

{

System.out.println("A char argument");

}

}

class B extends A

{

void m1(float f)

{

System.out.println("B Float argument");

}

void m1(char ch)

{

System.out.println("B char argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1(50);

b.m1('a');

b.m1(50L);

System.out.println();

A a = new B();

a.m1(50);

a.m1('a');

//a.m1(50L);

}

}

Output

A Integer argument

B char argument

B Float argument

A Integer argument

B char argument

**Note:** a.m1(50L) returns an error: actual argument long cannot be converted to int and char by method invocation conversion.

**Note:** a.m1(‘a’) returns output **B char argument** because it is method overriding.

class A

{

void m1(float f)

{

System.out.println("A Float argument");

}

}

class B extends A

{

void m1(float f)

{

System.out.println("B Float argument");

}

void m1(int a)

{

System.out.println("B Integer argument");

}

void m1(long l)

{

System.out.println("B long argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1(50);

b.m1('a');

b.m1(50L);

System.out.println();

A a = new B();

a.m1(50);

a.m1('a');

a.m1(50L);

}

}

Output

B Integer argument

B Integer argument

B Long argument

B Float argument

B Float argument

B Float argument

class A

{

void m1(Object obj)

{

System.out.println("A Object argument");

}

}

class B extends A

{

void m1(String s)

{

System.out.println("B String argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1("a");

b.m1(10);

System.out.println();

A a = new B();

a.m1("a");

a.m1(10);

}

}

Output

B String argument

A Object argument

A Object argument

A Object argument

class A

{

void m1(String s)

{

System.out.println("A String argument");

}

}

class B extends A

{

void m1(Object obj)

{

System.out.println("B Object argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1("a");

b.m1(10);

System.out.println();

A a = new B();

a.m1("a");

//a.m1(10);

}

}

Output

A String argument

B Object argument

A String argument

**Note:** a.m1(10) returns an error: actual argument int cannot be converted to String by method invocation conversion.

class A

{

void m1(String s)

{

System.out.println("A String argument");

}

void m1(Integer io)

{

System.out.println("A Integer io");

}

}

class B extends A

{

void m1(Object obj)

{

System.out.println("B Object argument");

}

void m1(String s)

{

System.out.println("B String argument");

}

void m1(Integer io)

{

System.out.println("B Integer io");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1("a");

b.m1(10);

System.out.println();

A a = new B();

a.m1("a");

a.m1(10);

}

}

Output

B String argument

B Integer io

B String argument

B Integer io

class A

{

void m1(Object o)

{

System.out.println("A Object argument");

}

}

class B extends A

{

void m1(Object obj)

{

System.out.println("B Object argument");

}

void m1(String s)

{

System.out.println("B String argument");

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

b.m1("a");

b.m1(10);

System.out.println();

A a = new B();

a.m1("a");

a.m1(10);

}

}

Output

B String argument

B Object argument

B Object argument

B Object argument

class A

{

int x = m1();

int m1()

{

System.out.println("A m1");

return 50;

}

}

class B extends A

{

int y = m1();

int m1()

{

System.out.println("B m1");

return 60;

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

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

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

}

}

Output

B m1

B m1

x : 60

y : 60

class A

{

int x = m1();

static int m1()

{

System.out.println("A m1");

return 50;

}

}

class B extends A

{

int y = m1();

static int m1()

{

System.out.println("B m1");

return 60;

}

}

class Example

{

public static void main(String args[])

{

B b = new B();

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

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

}

}

Output

A m1

B m1

x : 50

y : 60

**Constructor Overloading**

Defining multiple constructors in a class with different parameters type or list or order is called constructor overloading. Like methods, **constructors can also be overloaded but they cannot be overridden because constructors are not inherited.**

class Example

{

Example()

{

System.out.println("No argument constructor");

}

Example(int a)

{

System.out.println("Integer argument constructor");

}

Example(String s)

{

System.out.println("String argument constructor");

}

public static void main(String args[])

{

Example e1 = new Example();

Example e2 = new Example(10);

Example e3 = new Example("abc");

}

}

Output

No argument constructor

Integer argument constructor

String argument constructor

**Why should we overload constructor?**

1. To define different object initialization logic
2. To execute the same initialization logic by taking input values in different types.

**How can we call superclass overloaded constructors from subclass constructors?**

By using super(). We must place super() call in subclass constructor by passing argument same as superclass overloaded constructor parameter type.

class Sample

{}

class Example extends Sample

{

Example()

{

System.out.println("Example no - arg constructor");

}

Example(String s)

{

super(10);

System.out.println("Example string argument constructor");

}

public static void main(String args[])

{

Example e1 = new Example();

Example e2 = new Example("abc");

}

}

**Error:**

**required: no arguments  
found: int**

**Understanding this()**

**How can we call overloaded constructors from other constructors of same class without creating other new object?**

We must use this() call by passing the overloaded constructor parameter argument type.

class Example extends Sample

{

Example()

{

System.out.println("Example no - arg constructor");

}

Example(int a)

{

this();

System.out.println("Example int - arg constructor");

}

public static void main(String args[])

{

Example e = new Example(10);

}

}

Output

Example no – arg constructor

Example int – arg constructor

**Note: In the above program, only one object is created per new keyword execution, even though two constructors are executed.**

**What is the output of the below program?**

class Example

{

Example()

{

this(10);

System.out.println("No - arg constructor");

}

Example(int a)

{

this("abc");

System.out.println("int - arg constructor");

}

Example(String str)

{

System.out.println("String argument constructor");

}

public static void main(String args[])

{

Example e1 = new Example();

Example e2 = new Example(10);

Example e3 = new Example("abc");

}

}

Output

String argument constructor

int - arg constructor

No - arg constructor

String argument constructor

int - arg constructor

String argument constructor

**this() execution control flow**

Due to this() call, control is sent to respective parameter constructor that is matched with the given argument. Control is sent to another constructor without creating object and also without executing the current constructor logic. Object is created from the constructor that has super() call, because to create subclass object superclass memory should be created and should be included in subclass object. After super() execution subclass object initialized and NSB’s are executed, and then constructors logic is executed in reverse constructors call stack without creating new objects.

**What is the output of the below program?**

class Example

{

int x = m1();

{

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

}

int m1()

{

System.out.println("m1:x");

return 10;

}

Example()

{

this(10);

x = 50;

System.out.println("No arg constructor");

}

Example(int a)

{

this("abc");

x = 60;

System.out.println("int arg constructor");

}

Example(String str)

{

x = 70;

System.out.println("String arg constructor");

}

public static void main(String args[])

{

Example e1 = new Example();

System.out.println("e1.x : " + e1.x);

Example e2 = new Example(10);

System.out.println("e2.x : " + e2.x);

Example e3 = new Example("abc");

System.out.println("e3.x : " + e3.x);

}

}

Output

m1:x

Non Static Block

String arg constructor

int arg constructor

No arg constructor

e1.x : 50

m1:x

Non Static Block

String arg constructor

int arg constructor

e2.x : 60

m1.x

Non Static Block

String arg constructor

e3.x : 70

**Important point: Non – Static variables and non – static blocks are executed only once per object creation.**

**Constructor chaining**

Calling one constructor from other constructor by using super() and this() is called constructor chaining.

Subclass constructor are chained with superclass constructors by using super() and subclass overloaded constructors are chained by using this().

**What is the output of the following program?**

class SuperClass

{

SuperClass()

{

this(10);

System.out.println("Superclass no - arg");

}

SuperClass(int a)

{

this("abc");

System.out.println("Superclass int - arg");

}

SuperClass(String s)

{

System.out.println("Superclass string - arg");

}

}

class SubClass extends SuperClass

{

SubClass()

{

this(10);

System.out.println("Subclass no - arg");

}

SubClass(int a)

{

this("abc");

System.out.println("Subclass int - arg");

}

SubClass(String s)

{

System.out.println("Subclass string - arg");

}

}

class Example

{

public static void main(String args[])

{

new SubClass();

System.out.println();

new SubClass(10);

System.out.println();

new SubClass("abc");

}

}

Output

Superclass string - arg

Superclass int - arg

Superclass no - arg

Subclass string - arg

Subclass int - arg

Subclass no - arg

Superclass string - arg

Superclass int - arg

Superclass no - arg

Subclass string - arg

Subclass int - arg

Superclass string - arg

Superclass int - arg

Superclass no - arg

Subclass string - arg

**Rules on this()**

1. Like super(), this() call also must be placed as first statement only in constructor, else it leads to CE: **“call to this must be first statement in a constructor”**.

class Example

{

Example()

{

System.out.println("No arg");

this(10);

}

Example(int a)

{

System.out.println("int arg");

}

public static void main(String args[])

{

new Example();

}

}

Error: call to this must be first statement in a constructor

1. this() and super() calls cannot be placed in the same constructor, because both must be placed as first statement in constructor.

class Example

{

Example()

{

this(10);

super();

System.out.println("No arg");

}

Example(int a)

{

System.out.println("int arg");

}

public static void main(String args[])

{

new Example();

}

}

Error: call to super must be first statement in a constructor

1. Also, multiple this() or super() cannot be placed in same constructor, violation leads to same CE.

class Example

{

Example()

{

this(10);

this(10);

System.out.println("No arg");

}

Example(int a)

{

super();

super();

System.out.println("int arg");

}

public static void main(String args[])

{

new Example();

}

}

Error:   
call to this must be first statement in a constructor  
call to super must be first statement in a constructor

1. In an object creation we are not allowed to call same constructor more than once, violation leads to CE: “**recursive constructor invocation**”. But we are allowed to call it again in another object creation.
2. So, this() cannot be placed in all constructors, at least one constructor must be left with super() call in order to call superclass constructor when subclass object is created. Else it leads to CE: “**recursive constructor invocation**”.

class Example

{

Example()

{

this(10);

System.out.println("No arg");

}

Example(int a)

{

this("abc");

System.out.println("int arg");

}

Example(String s)

{

this();

System.out.println("string arg");

}

public static void main(String args[])

{

new Example();

}

}

Error: recursive constructor invocation

1. Also in a constructor its own this call cannot be placed, violation leads to CE: “**recursive constructor invocation**”.

class Example

{

Example()

{

this();

System.out.println("No arg");

}

public static void main(String args[])

{

new Example();

}

}

Error: recursive constructor invocation

**FINAL METHODS AND FINAL CLASSES**

**Final Method**

A method which has final keyword in its definition is called final method. Rule is it cannot be overridden in subclass. But it is inherited to subclass, and we can invoke it to execute its logic.

**When a method should be declared as final?**

If we do not want allow subclass to override superclass method and to ensure that all subclasses uses the same superclass method logic then that method should be declared as final method.

**Rule:** Final method cannot be overridden in subclass, violation leads to CE.

class Sample

{

void m1()

{

System.out.println("Sample m1()");

}

final void m2()

{

System.out.println("Sample m2()");

}

}

class Example extends Sample

{

void m1()

{

System.out.println("Example m1()");

}

void m2()

{

System.out.println("Example m2()");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2();

}

}

Error: m2() in Example cannot override m2() in Sample

Overridden method is final

**What is difference between private and final methods?**

Private method cannot be inherited, whereas final method can be inherited but cannot be overridden.

**Can we overload final methods in subclass?**

Yes, we can overload final methods in subclass.

class Sample

{

void m1()

{

System.out.println("Sample m1()");

}

final void m2(int a)

{

System.out.println("Sample m2()");

}

}

class Example extends Sample

{

void m1()

{

System.out.println("Example m1()");

}

void m2(float b)

{

System.out.println("Example m2()");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2(10);

e.m2(5.1F);

}

}

Output

Example m1()

Sample m1()

Sample m2()

**Can we declare main() method as final?**

Yes, it is possible. But then we cannot override main() method.

class Sample

{

public static final void main(String args[])

{

System.out.println("Sample");

}

}

class Example extends Sample

{

public static void main(String args[])

{

System.out.println("Example");

}

}

Error: main(String[]) in Example cannot override main(String[]) in Sample

**Final Class**

A class which has final keyword in its definition is called final class and it cannot be extended.

**When a class should be declared as final?**

1. If we do not want to override all methods of our class.
2. If we do not want to extend our class functionality.

final class Sample

{

void m1()

{

System.out.println("Sample class m1()");

}

}

class Example extends Sample

{

public static void main(String args[])

{

System.out.println("Example");

}

}

Error: cannot inherit from final sample

**If a class is declared as final, are all its members final?**

No, final class members are not final until they are declared as final members explicitly by using final keyword.

**Can we instantiate final class?**

Yes, we can instantiate final class means we can create object of the class.

**Find out the compile time error in the below program.**

final class Sample

{

int x = 10;

final int y = 20;

void m1()

{

System.out.println("Sample class m1()");

}

}

class Example

{

public static void main(String args[])

{

Sample s = new Sample();

s.x = 50;

s.y = 60;

s.m1();

}

}

Error: cannot assign value to final variable y

**ABSTRACT METHOD AND ABSTRACT CLASSES**

A method that does not have body is called abstract method, and the class that is declared as abstract using abstract keyword is called abstract class. If a class contains abstract methods, it must be declared as abstract.

**Rule 1:** If method does not have not body, then it should be declared using abstract keyword else it will leads to CE.

class Example

{

void m1();

public static void main(String args[])

{

}

}

Error: Missing method body or declared abstract

**Rule 2:** If a class have abstract method, then it should be declared as abstract else it will leads to CE.

class Example

{

abstract void m1();

public static void main(String args[])

{

}

}

Error: Example is not abstract and does not override abstract m1() in Example

**Rule 3:** If class is declared as abstract then it cannot be instantiated, violation leads to CE.

abstract class Example

{

abstract void m1();

public static void main(String args[])

{

Example e = new Example();

}

}

Error: Example is abstract; cannot be instantiated.

**Why abstract class cannot be instantiated?**

Because it is not fully implemented class. So its abstract method cannot be executed.

**Who will provide implementation for abstract methods?**

Sub class developer provide implementation for abstract methods according to their business requirement. Basically in projects abstract methods are (methods prototypes) are defined by super class developers, and they are implemented (method body and logic) by subclass developer.

**Rule 4:** The subclass of an abstract class should override all abstract methods or it should declared as abstract, violation leads to CE.

abstract class Sample

{

abstract void m1();

abstract void m2();

}

class Example extends Sample

{

void m1()

{

System.out.println("m1()");

}

}

Error: Example is not abstract and does not override abstract m2() in Sample

**Solution to the problem**

1. Declare child class as abstract

abstract class Sample

{

abstract void m1();

abstract void m2();

}

abstract class Example extends Sample

{

void m1()

{

System.out.println("m1()");

}

}

1. Override all methods of parent class

abstract class Sample

{

abstract void m1();

abstract void m2();

}

class Example extends Sample

{

void m1()

{

System.out.println("m1()");

}

void m2()

{

System.out.println("m2()");

}

}

**What type of members we can define in abstract class?**

We can define all static and non – static members including constructors.

**Will abstract memory be created when subclass object be created?**

Yes, its non – static members get memory when its concrete subclass be created.

**How can we execute static and non – static concrete members of abstract class?**

Static members can be executed directly from its main method and its non – static members are executed by using its concrete subclass object.

**Program to show execution of static members**

abstract class Example

{

static int a = 10;

static

{

System.out.println("Example static block");

}

static void m2()

{

System.out.println("Static function m2()");

}

public static void main(String args[])

{

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

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

m2();

}

}

Output

Example static block

Example main

a : 10

Static function m2()

**Program to show execution of non – static members**

abstract class Sample

{

abstract void m1();

int a = 10;

{

System.out.println("Example non static block");

}

void m2()

{

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

}

Sample()

{

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

}

}

class Example extends Sample

{

void m1()

{

System.out.println("Sample abstract function m1() defined in

Example");

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

e.m2();

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

}

}

Output

Example non static block

Constructor of the class

Sample abstract function m1() defined in Example

Non Static function m2()

a : 10

**What is the output of the below program?**

abstract class Sample

{

abstract void m1();

static int a = 10;

int x = 20;

static

{

System.out.println("Sample static block");

}

{

System.out.println("Sample non static block");

}

Sample()

{

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

}

static void m2()

{

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

}

void m3()

{

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

}

public static void main(String args[])

{

System.out.println("Sample main");

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

m2();

}

}

class Example extends Sample

{

static int b = 30;

int y = 40;

static

{

System.out.println("Example static block");

}

{

System.out.println("Example non static block");

}

Example()

{

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

}

static void m4()

{

System.out.println("Static function m4() of class Example");

}

void m5()

{

System.out.println("Non Static function m5() of class Example");

}

void m1()

{

System.out.println("Class Sample abstract function m1() defined in class Example");

}

public static void main(String args[])

{

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

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

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

m2();

m4();

System.out.println();

Example e = new Example();

e.m1();

e.m3();

e.m5();

}

}

Output

Sample static block

Example static block

Example main

a : 10

b : 30

Static function m2() of class Sample

Static function m4() of class Example

Sample non static block

Constructor of Sample class

Example non static block

Constructor of Example class

Class Sample abstract function m1() defined in class Example

Non Static function m3() of class Sample

Non Static function m5() of class Example

**Can we declare abstract method as static?**

No, we are not allowed to declare abstract method as static. It leads to compile error “*illegal combination of modifiers: abstract and static”.*

abstract class Example

{

static abstract void m1();

}

Error: illegal combination of modifiers: abstract and static

**Can we declare abstract method as final?**

No, because it should be allowed to override in subclass. It leads to compile time error “*illegal combination of modifiers: abstract and final”.*

abstract class Example

{

final abstract void m1();

}

Error: illegal combination of modifiers: abstract and final

**Can we declare abstract method as private?**

No, because it should be inherited to subclass. It leads to compile time error “*illegal combination of modifiers: abstract and private”.*

abstract class Example

{

private abstract void m1();

}

Error: illegal combination of modifiers: abstract and static

**What are the legal modifiers allowed in combination with abstract modifier?**

Only two modifiers are allowed with abstract.

1. protected
2. public

**If we abstract modifier twice, then also it leads to CE: repeated modifier**

abstract class Example

{

abstract abstract void m1();

}

Error: repeated modifier

**INTERFACE**

Interface is a fully unimplemented class used to support multiple inheritance.

In developing, using and deriving a subclass from an interface we must follow below rules:

**Rule 1:** Interface cannot have method body, violation leads to CE.

interface A

{

void m1()

{

System.out.println("m1()");

}

}

Error: interface methods cannot have body

**Rule 2:** In interface we can have only static final variables means they must be initialized.

interface A

{

int a;

}

Error: = expected

**Rule 3:** We cannot declare interface members as private or protected members, violation leads to CE.

interface A

{

private void m1();

protected void m2();

}

Error: modifier private and protected not allowed here

**Rule 5:** Interfaces cannot be instantiated.

interface A

{}

class Example

{

public static void main(String args[])

{

A a = new A();

}

}

Error: A is abstract; cannot be instantiated

**Rule 6:** We can create reference variable of interface for storing its **subclass objects references to develop loosely coupled user application to get Runtime Polymorphism** for executing methods from its different classes.

interface A

{}

class Example implements A

{

public static void main(String args[])

{

A a;

a = new Example();

}

}

**Rule 7:** We cannot declare interface as final, it leads to CE. Because it should contain subclass.

final interface A

{}

Error: illegal combination of modifiers: interface and final

**Can we apply abstract keyword to interface?**

Yes, it is optional and at compilation time it is removed by compiler.

**Can we create empty interface?**

Yes, it is possible.

**How can we write logic in interface?**

Using the concept of inner class.

**Inheritance with interface**

Using **implements** keyword, we can derive a class from interface.

**Rule 8:** All the methods must be defined in subclass of interface using the keyword public, otherwise it will lead to CE. Because the default access modifier of interface is public whereas default access modifier of class is private.

interface A

{

void m1();

}

class Example implements A

{

void m1()

{

System.out.println("m1()");

}

}

Error: m1() in Example cannot implement m1() in A;

attempting to assign weaker access privileges.

**Solution is to define m1() in Example using public.**

interface A

{

void m1();

}

class Example implements A

{

public void m1()

{

System.out.println("m1()");

}

}

**Rule 9:** The class derived from interface should implement all abstract methods of interface, otherwise it will lead to CE.

interface A

{

void m1();

void m2();

}

class Example implements A

{

public void m1()

{

System.out.println("m1()");

}

}

Error: Example is not abstract and does not override abstract method m2() in A.

**Solution to the problem**

1. Define the subclass as abstract

interface A

{

void m1();

void m2();

}

abstract class Example implements A

{

public void m1()

{

System.out.println("m1()");

}

}

**Program demonstrates implementation of interface in inheritance**

interface Vehicle

{

void enegine();

void breaks();

}

abstract class Bus implements Vehicle

{

public void breaks()

{

System.out.println("Bus has two breaks");

}

}

class Volvo extends Bus

{

public void enegine()

{

System.out.println("Enegine capacity is 60 km/hr");

}

}

class RedBus extends Bus

{

public void enegine()

{

System.out.println("Enegine capacity is 40 km/hr");

}

}

class Example

{

static void assign(Vehicle v)

{

v.enegine();

v.breaks();

}

public static void main(String args[])

{

assign(new Volvo());

assign(new RedBus());

}

}

Output

Enegine capacity is 60 km/hr

Bus has two breaks

Enegine capacity is 40 km/hr

Bus has two breaks

**SIMILARITIES AND DIFFERENCES BERWEEN ABSTRACT CLASS AND INTERFACE**

**Main differences to be answered in interview**

Interface is a fully unimplemented class used for declaring operations of an object. Abstract class is a partially implemented class. It implements some of the operations of the object those are declared in its interface. These implemented operations are common for all next level subclasses. Remaining operations are implemented by the next level subclasses according to their requirement.

Interface allows us to develop multiple inheritance so we must start object design with interface, whereas abstract class does not support multiple inheritance so it always comes next to interface in object development graph.

**Similarities**

1. Both interface and abstract class cannot be instantiated, but abstract class can be instantiated indirectly via sub class.
2. Reference variable can be creates for both interface and abstract class.
3. Subclass should implement all abstract methods.
4. Both cannot be declared as final.
5. Both can have inner class.
6. Both have default access modifier is package, can be changed to public.

**Understanding enum**

**Definition and need of enum**

enum is one type of class which is final. It is created by using the keyword “enum”. It is used for defining set of named constants those represents a menu kind of items.

Before Java 5 these menu items are created by using class. class has a problem in accessing these menu items, i.e. it cannot return or print item name as it is declared instead it returns or prints its value.

class Months

{

static final int JAN = 1;

static final int FEB = 2;

}

class Example

{

public static void main(String args[])

{

System.out.println(Months.JAN);

System.out.println(Months.FEB);

}

}

Output

1

2

**Syntax**

enum <enumname>

**Note: For creating named constants use only names, do not use any datatype.**

enum Months

{

JAN,FEB;

}

class Example

{

public static void main(String args[])

{

System.out.println(Months.JAN);

System.out.println(Months.FEB);

}

}

Output

JAN

FEB

**//An enumeration of apple varieties**

enum Apple

{

Jonathan, GoldenDel, RedDel, Winesap, Cortland;

}

The identifiers Jonathan, GoldenDel, RedDel, Winesap, Cortland are called enumerations constants. Each is implicitly declared as public, static final member of Apple. Furthermore, their type is the type of enumeration in which they are declared, which is Apple in this case. Thus, in the language of Java, these constants are called **self – typed**, in which self refers to the enclosing enumeration.

However, even though enumerations define a class type, **we cannot instantiate enum using new keyword**.

We can create variable of enumeration type.

Apple ap;

Because **ap** is of type **Apple**, the only values that it can be assigned or contains are those defined by the enumerations.

ap = Apple.RedDel

Two enumeration constants can be compared for equality by using the == relational operator.

if (ap == RedDel) // ….

An enumeration value can also be used to control a **switch statement**. All the case statements must use constants from the same enum as that used by the switch expression.

switch(ap)

{

case Jonathan:

//....

case Winesap:

//....

}

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, Jonathan, not Apple.Jonathan, is used. This is because the type of enumeration in the switch expression has already implicitly specified the enum type of the **case** constants. There is no need to qualify the constants in the **case** statements with their enum type name. Attempting to do so will cause CE.

enum Apple

{

Jonhathan,GoldenDel,RedDel,Winesap,Cortland;

}

class Example

{

public static void main(String args[])

{

Apple ap;

ap = Apple.RedDel;

System.out.println("Value of ap : " + ap);

ap = Apple.Jonhathan;

if(ap == Apple.Jonhathan)

System.out.println("ap contains Jonhathan");

switch(ap)

{

case Jonhathan:

System.out.println("Jonhathan is yellow");

break;

case GoldenDel:

System.out.println("Golden Delicious is red");

break;

case RedDel:

System.out.println("Red Delicious is red");

break;

case Winesap:

System.out.println("Winesap is red");

break;

case Cortland:

System.out.println("Cortland is red");

break;

}

}

}

Output

Value of ap : RedDel

ap contains Jonhathan

Jonhathan is yellow

**The values() and valueOf() methods**

All enumerations automatically contains two pre – defined methods: values() and valueOf().

Their general forms are:

public static enum – type[] values()

public static enum – type valueOf(String str)

The values() method returns an array that contains a list of the enumeration constants. The valueOf() method returns the enumeration constant whose value corresponds to the String passed in **str**. In both cases, enum – type is the type of enumerations.

enum Apple

{

Jonhathan,GoldenDel,RedDel,Winesap,Cortland;

}

class Example

{

public static void main(String args[])

{

System.out.println("Here is the list of Apples");

Apple apples[] = Apple.values();

for(Apple ap : apples)

System.out.println(ap);

Apple a = Apple.valueOf("Jonhathan");

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

}

}

Output

Here is the list of Apples

Jonhathan

GoldenDel

RedDel

Winesap

Cortland

a contains : Jonhathan

**What is the data type of the named constants in enum?**

Its data type is current enum type. It is added by compiler automatically.

In the enum Months (first enum program), its named constants are converted as shown below:

public static final Months JAN;

public static final Months FEB;

**Compiler changed code for the enum Months [Months.class]**

final class Months extends java.lang.Enum

{

public static final Months JAN;

public static final Months FEB;

private static final Months[] $VALUES;

public static Months[] values()

{

return (Months[])$VALUES.clone();

}

public static Months valueof(String s)

{

return (Months)Enum.valueOf(Months,s);

}

private Months(String s,int i)

{

super(s,i);

}

static

{

JAN = new Months("JAN",0);

FEB = new Months("FEB",1);

$VALUES = new Months[]{JAN,FEB};

}

}

**Conclusion from above program**

1. enum is a final class.
2. It is subclass of java.lang.Enum.
3. It is an abstract class which is the default super class for enum type classes. Also it is implementing from Comparable and Serializable interfaces.
4. Since, every enum type is Comparable, so we can its objects to collection TreeSet object, also it can be stored in file as it is Serializable type.
5. All named constants created inside enum are referenced type the current type.
6. These enum type variables are initialized in static block with this enum class object by using (String, int) parameter constructor.  
   Here
   1. String parameter value is named constant name exactly as declared in its enum declaration
   2. int parameter value is its position in the list. The position number starts from “ZERO”.
7. In every enum type class compiler places private(String, int) parameter constructor with super(s, i) statement to call java.lang.Enum class constructor. The string value “named constant name” and its ordinal value “its position in the list”. These two values are stored in java.lang.Enum class non – static variables “name” and “ordinal” respectively, which are created in our enum class object separately for every enum variable’s object.
8. Two methods value() and valueOf() methods are added.
9. It is also inheriting methods from java.lang.Enum class. The important methods are
   1. public final String name()  
      **returns name of the current enum exactly as declared in enum declaration**
   2. public final int ordinal()  
      **returns current enum position**

**Why compiler places no – arg constructor with no – arg super class in normal classes and parameterized constructor with (String, int) arg super class in enum?**

For normal classes, their super class is java.lang.Object and it has no – arg constructor. So compiler places default constructor without parameters with no – arg super call.

But for enum, their super class is java.lang.Enum, it has String and int parameter constructor. So compiler places default constructor with (String, int) parameter with super call with String, int argument.

**enum type class object JVM architecture**

Every enum type class object has minimum 2 variables

1. name
2. ordinal

Every named constant variable defined inside enum type is an object’s referenced variable that holds current enum class object. So, in the above Months enum class JAN, FEB are objects.

enum Months

{

JAN,FEB;

}

class Example

{

public static void main(String args[])

{

Months months[] = Months.values();

for(Months month : months)

{

System.out.print(month.name());

System.out.print(" --- ");

System.out.println(month.ordinal());

}

}

}

Output

JAN --- 0

FEB --- 1

**How can we assign values to Menu items in enum?**

Syntax

Named\_constant(value)

Example

JAN(1), FEB(2)

**Rule:** To assign values to named constants, enum must have a parameterized constructor with the passed argument type. Else it leads to CE.

**Where these values 1, 2 are stored?**

We must create a non – static int type variable to store these values.

So, to store these named constants values we must follow below 3 rules:

1. We must create non – static variable in enum.
2. We must define parameterized constructor with argument type.
3. Named constants must end with ;( semicolon) to place normal variables, methods, constructors explicitly. It acts as separators for compiler to differentiate named constants and general members.

**Program to demonstrate to print assigned value of named constant.**

enum Months

{

JAN(1),FEB(2);

int num;

Months(int num)

{

this.num = num;

}

int getValue()

{

return num;

}

}

class Example

{

public static void main(String args[])

{

Months months[] = Months.values();

for(Months month : months)

{

System.out.println(month.getValue());

}

}

}

Output

1

2

**enum has below set of rules**

1. We cannot derive a subclass from enum, because it is final class.
2. We cannot instantiate enum using new and constructor, it leads to CE: enum types may not be instantiated. But it gets instantiated by compiler in .class file.
3. Named constant must be separated by ;( semicolon), but not by ,( comma).
4. We can place ;( semicolon) at the end of all named constants, here in enum it is acts as separator and also tells end of all named constants.
5. Inside enum we can define
   1. Named constants
   2. All static members – SV, SB, SM, Main Method
   3. All non – static members except abstract method – NSV, NSB, NSM, constructor  
      **abstract method is not allowed because we cannot declare enum as abstract as enum object should be created to initialize named constant.**
   4. Inner class, interface, enum

enum Color

{

RED(15),BLUE(25),GREEN;

static int a = 10;

int x = 20;

static void m1()

{

System.out.println("Static m1()");

}

void m2()

{

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

}

static

{

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

}

{

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

}

Color()

{

System.out.println("No arg Constructor");

this.x = 50;

}

Color(int x)

{

System.out.println("int arg Constructor");

this.x = x;

}

public static void main(String args[])

{

System.out.println("Color main");

}

class A

{}

}

class Example

{

public static void main(String args[])

{

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

//accessing named constants

System.out.println(Color.RED);

System.out.println(Color.BLUE);

//accessing static members

System.out.println(Color.a);

Color.m1();

//accessing non - static members

System.out.println(Color.RED.x);

System.out.println(Color.BLUE.x);

System.out.println(Color.GREEN.x);

Color.RED.m2();

Color.BLUE.m2();

}

}

Output

Example main

Non Static Block

int arg Constructor

Non Static Block

int arg Constructor

Non Static Block

No arg Constructor

Static Block

RED

BLUE

10

Static m1()

15

25

50

Non Static m2()

Non Static m2()

1. Named constants must be the first statement in enum and they must be separated with comma.
2. We can also create empty enum.
3. To define enum only with non – static members, it must starts with ;( semicolon) to tell compiler that enum is without named constant(s).

enum Color

{

;

int a = 10;

}

1. We cannot declare explicit constructor as protected or public, because enum constructor is by default private.

enum Color

{

;

private Color()

{}

}

No error

enum Color

{

;

protected Color()

{}

}

Error: modifier protected not allowed here

enum Color

{

;

public Color()

{}

}

Error: modifier public not allowed here

1. If we define no – arg constructor as the first statement in enum and if we are not using access modifier explicitly, then ;( semicolon) is optional.

enum Color

{

Color()

{}

}

1. But ;( semicolon) is mandatory, if we define constructors with parameters.

enum Color

{

Color(int a)

{}

}

Error: { expected

enum Color

{

;

Color(int a)

{}

}

No Error

1. ;( semicolon) is mandatory if we define no – arg constructor along with normal members

enum Color

{

Color()

{}

int a = 10;

}

Error: { expected

enum Color

{

;

Color()

{}

int a = 10;

}

No Error

1. Named constants must be declared according to that enum’s available constructors.
   1. If we do not define constructor, it has default constructor. So named constants must be declared without arguments.

enum Color

{

JAN,FEB,MAR(3);

}

Error: constructor Color in enum Color cannot be applied to given types

required: no arguments

found: int

* 1. If we define constructor with int parameter, then it does not have default constructor, so all named constants declared with only int arguments.

enum Color

{

JAN,FEB,MAR(3);

Color(int a)

{}

}

Error: constructor Color in enum Color cannot be applied to given types

required: int

found: no arguments

* 1. If we define constructors with no – arg, int and String parameters, then all named constants must be declared either without argument, or with int argument or with String argument.

1. Only strictfp modifier is allowed for enum.
2. We can define enum inside a class but we cannot define inside a method.

class Example

{

enum Color

{}

}

No Error

**Note: private, protected, public, static, strictfp are allowed for inner enum  
 final and abstract modifiers are not allowed for inner enum**

class Example

{

void m1()

{

enum Color

{}

}

}

Error: enum types must not be local

**Can we overload constructor in enum?**

We can overload constructors in enum to initialize named constant with objects with different types of values.

**INNER CLASSES**

The class defined inside another class or interface is called inner class.

We can also create an interface in another class or interface.

class Example

{

class Sample

{}

}

class Example

{

interface Sample

{}

}

interface Example

{

class Sample

{}

}

**From which Java version inner class concept is available?**

From Java 1.1 version inner class concept is introduced for supporting AWT, and Swing components event handling.

**Need of inner class**

Inner class is used for creating an object logically inside another object with clear separation of properties region.

**Types of inner classes**

We have 4 types of inner classes:

1. Nested class (static inner class)
2. Inner class (non – static inner class)
3. Method local class (local inner class)
4. Anonymous class (argument inner class)

**Syntax to create all above 4 types of inner classes**

Nested class (static inner class)

class Example

{

static class Sample

{}

}

Inner class (non – static inner class)

class Example

{

class Sample

{}

}

Method local class (local inner class)

class Example

{

void m1()

{

class Sample

{}

}

}

Anonymous class (argument inner class)

class Example

{

void m2()

{

new Thread()

{};

}

}

Compiler generates .class file separately for every inner class, so that the inner class logic is completely separated from outer class and stored in another .class file.

**Find out .class file names for below inner classes**

class Example

{

static class A

{}

class B

{}

void m1()

{

class C

{}

new Thread()

{};

class D

{}

}

void m2()

{

class C

{}

class E

{}

new Thread()

{};

}

}

Output

Example.class

Example$A.class

Example$B.class

Example$1C.class

Example$1.class

Example$1D.class

Example$2C.class

Example$1E.class

Example$2.class

**Static inner class**

The inner class defined at class level with static keyword is called static inner class.

**Modifiers allowed**

private, protected, public, final, abstract, strictfp

**Types of members allowed**

All 8 types of static and non – static members are allowed

1. Static variables
2. Static block
3. Static method
4. Main method
5. Non – static variables
6. Non – static block
7. Non – static method
8. Constructor

**Accessing outer class members from inner class**

We can access outer class all members from its static inner class including private members. In accessing outer class members we just consider static inner class as a static method of outer class. So, we can access static members directly by their name from static inner class. But we cannot access non – static members directly by their name. We must access them only by creating outer class object.

class Example

{

static int a = 10;

int b = 20;

private int c = 30;

static class Sample

{

public static void main(String args[])

{

System.out.println(a);

System.out.println(b); // Error

System.out.println(c); // Error

Example e = new Example();

System.out.println(e.a);

System.out.println(e.b);

System.out.println(e.c);

}

}

}

Error: non – static variable b and c cannot be accessed from a static block

**Can we call outer class members using inner class referenced variable?**

No, it leads to CE: cannot find symbol

class Example

{

int x = 10;

static class Sample

{

public static void main(String args[])

{

Sample s = new Sample();

System.out.println(s.x);

}

}

}

Error: cannot find symbol

**Why cannot we call non – static members from static inner class?**

Static inner class members do not get memory in outer class object. So, we must create outer class object explicitly to access it non – static members. But in case of static members call compiler places outer class name in accessing static members’ definition.

**Accessing inner class members from outer class**

Including private members we can access all members of static inner class from outer class as shown below:

1. Static members by using inner class name
2. Non – static members by using inner class object

class Example

{

static class Sample

{

static int a = 10;

int b = 20;

private int c = 30;

}

public static void main(String args[])

{

Sample s = new Sample();

System.out.println(Sample.a);

System.out.println(s.b);

System.out.println(s.c);

}

}

Output

10

20

30

**Accessing inner class members from outside of outer class**

The syntax for accessing inner class from outside outer class is

**Outerclassname.innerclassname**

**Rule:** Like outer class private members, inner class private members are also cannot be accessed from outside of outer class.

class Example

{

static class A

{

static int a = 10;

int b = 20;

private int c = 30;

}

}

class Sample

{

public static void main(String args[])

{

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

Example.A a1 = new Example.A();

System.out.println("b : " + a1.b);

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

}

}

Output

10

20

Note: a1.c will raise an error “**c has private access in A”.**

**Inner class Execution**

By just executing outer class, static inner class members are not executed automatically.

class Example

{

static class A

{

public static void main(String args[])

{

System.out.println("Inner class main method");

}

}

public static void main(String args[])

{

System.out.println("Outer class main method");

}

}

Output

Outer class main method

We must execute static inner class separately from command prompt as:

**java Example$A**

**Note:**

1. When we load outer class, inner class is not get loaded.
2. When we load inner class, outer class is not get loaded.

class Example

{

static

{

System.out.println("Outer class get loaded");

}

static class A

{

static

{

System.out.println("Inner class get loaded");

}

public static void main(String args[])

{

System.out.println("Inner class main method");

}

}

public static void main(String args[])

{

System.out.println("Outer class main method");

}

}

**Executing outer class**

java Example

**Output**

Outer class get loaded  
Outer class main method

**Executing inner class**

java Example$A

**Output**

Inner class get loaded

Inner class main method

1. If we access outer class members from inner class or inner class members from outer class, then both the classes will gets loaded into JVM.

class Example

{

static

{

System.out.println("Outer class get loaded");

}

Example()

{

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

}

static class A

{

static

{

System.out.println("Inner class get loaded");

}

A()

{

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

}

static void m1()

{

System.out.println("Static method of inner class");

}

void m2()

{

System.out.println("Non static method of inner class");

}

public static void main(String args[])

{

System.out.println("Inner class main method");

m3();

Example e = new Example();

e.m4();

}

}

static void m3()

{

System.out.println("Static method of outer class");

}

void m4()

{

System.out.println("Non static method of outer class");

}

public static void main(String args[])

{

System.out.println("Outer class main method");

A.m1();

A a = new A();

a.m2();

}

}

**Executing outer class**

java Example

**Output**

Outer class get loaded

Outer class main method

Inner class get loaded

Static method of inner class

Constructor of the inner class

Non Static method of inner class

**Executing inner class**

java Example$A

**Output**

Inner class get loaded

Inner class main method

Outer class get loaded

Static method of outer class

Constructor of outer class

Non static method of outer class

**Can we define outer class members in inner class again?**

Yes, it is possible.

**Then how can we differentiate both members in inner class?**

By using their class name or object name.

class Example

{

static int a = 10;

int b = 20;

static class A

{

static int a = 30;

int b = 40;

void m1()

{

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

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

Example e = new Example();

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

System.out.println("Example b : " + e.b);

}

}

void m2()

{

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

System.out.println("Example b : " + b);

A a1 = new A();

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

System.out.println("A b : " + a1.b);

}

public static void main(String args[])

{

Example e = new Example();

e.m2();

A a1 = new A();

a1.m1();

}

}

Output

Example a : 10

Example b : 20

A a : 30

A b : 40

A a : 30

A b : 40

Example a : 10

Example b : 20

class Sample

{

public static void main(String args[])

{

Example e = new Example();

Example.A a1 = new Example.A();

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

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

System.out.println("Example b : " + e.b);

System.out.println("A b : " + a1.b);

}

}

Output

Example a : 10

A a : 30

Example b : 20

A b : 40

**Non – Static Inner Class**

The inner class that is defined at class level without static keyword is called non – static inner class. We must develop non – static inner class to create a separate object in outer class object, so that when outer class is instantiated, this inner class members are provided memory as part of every outer class instance.

**After compiling outer class with non – static inner class, how many .class files are generated?**

Two .class files get generated. One for outer class and another one for inner class (**same as static inner class).**

**Allowed modifiers for non – static inner class**

private, protected, public, final, abstract, strictfp

**Types of modifiers allowed inside non – static inner class**

Only non – static members are allowed. They are:

1. Non – static variable
2. Non – static block
3. Non – static method
4. Constructor

We should not declare static members in non – static inner class. It leads to CE.

class Example

{

class A

{

static int a = 10;

int b = 20;

}

}

Error: Illegal static declaration in inner class Example.A

Modifier static is only allowed in constant variable declarations

**Accessing outer class members from inner class**

We can access outer class all members from its non – static inner class including private members by their name directly.

class Example

{

static int a = 10;

int b = 20;

private int c = 30;

class A

{

void m1()

{

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

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

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

}

}

}

No Error on compilation

**Accessing non – static inner class members from outer class and outside outer class**

**How can we execute non – static inner class members?**

We must call non – static inner class members from outer class main method or outside outer class main method by using its object.

**Syntax to create non – static inner class object**

There are two syntaxes:

1. **Old Syntax**innerclassname ref\_var = new innerclassconstructor();
2. **New Syntax**

Outerclassname.innerclassname ref\_var = new outerclassconstructor().new innerclassconstructor();

We must use first syntax to access non – static inner class members from non – static members of its outer class.

We must use second syntax to access non – static inner class members from static members of its outer class and also from outside outer class.

**Why we must create outer class object to create non – static inner class object?**

Because this inner class is a non – static member of outer class. To access non – static member of the class from static context we must have to create class object.

class Example

{

static int a = 10;

int b = 20;

private int c = 30;

class A

{

void m1()

{

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

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

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

}

}

public static void main(String args[])

{

Example.A a1 = new Example().new A();

a1.m1();

}

}

Output

a : 10

b : 20

c : 30

class Example

{

static int a = 10;

int b = 20;

private int c = 30;

class A

{

void m1()

{

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

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

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

}

}

void m1()

{

A a = new A(); // compiler changed code : A a = this.A();

a.m1();

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

}

}

Output

a : 10

b : 20

c : 30

class Sample

{

public static void main(String args[])

{

Example.A a1 = new Example().new A();

a1.m1();

}

}

Output

a : 10

b : 20

c : 30

class Sample

{

public static void main(String args[])

{

Example e = new Example();

Example.A a1 = e.new A();

a1.m1();

}

}

Output

a : 10

b : 20

c : 30

**Can we define main method in non – static inner class?**

No, it leads to CE.

**Can we execute non – static inner class members by using java command as java Example$A**

No, it leads to exception: main method not found

So to run non – static inner class, we must execute its outer class.

**Can we define outer class members in inner class again?**

Yes, it is possible.

**Then how can we differentiate both members in inner class?**

By using, **outerclassname.this**

class Example

{

int x = 10;

class A

{

int x = 20;

void m1()

{

System.out.println("m1() x : " + x);

System.out.println("m1() this.x : " + this.x);

System.out.println("m1() Example.this.x : " + Example.this.x);

}

void m2()

{

int x = 30;

System.out.println("m2() x : " + x);

System.out.println("m2() this.x : " + this.x);

System.out.println("m2() Example.this.x : " + Example.this.x);

}

}

void m3()

{

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

A a = new A();

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

}

public static void main(String args[])

{

Example e = new Example();

Example.A a1 = e.new A();

a1.m1();

a1.m2();

e.m3();

}

}

Output

m1() x : 20

m1() this.x : 20

m1() Example.this.x : 10

m2() x : 30

m2() this.x : 20

m2() Example.this.x : 10

Example x : 10

A x : 20

**Note: Non – Static inner class bytecodes are stored in every outer class instance.**

**Method Local Inner Class**

The inner class defined inside a method of outer class is called method inner class.

**Syntax**

class A

{

void m1()

{

class B

{}

}

}

**After compiling outer class with a local inner class, how many .class files will generate?**

Two .class files will generate.

One for outer class and another one for inner class.

A.class

A$1B.class

Where 1 is the index number representing B is the first local inner class created in class A

**Allowed modifiers**

Only final, abstract and strictfp.

Accessibility modifiers (public, private and protected) are not allowed.

**Types of members allowed in local class**

Only non – static members are allowed. They are:

1. Non – static variable
2. Non – static block
3. Non – static method
4. Constructor

**Rule:** We should not declare static members in local inner class.

It leads to CE: “inner class cannot have static declarations”.

class Example

{

void m1()

{

class A

{

}

}

}

Error: Illegal static declaration in inner class A.

Modifier static is only allowed in constant variable declarations

**Accessing outer class members from local inner class**

1. If local class is defined in static method, outer class static members are accessible directly and non – static methods are accessible only through outer class object.
2. If local class is defined in non – static method, outer class all static and non – static members are accessible directly by their names and they ate executed from the current object of that method.
3. Its methods parameters and local variables are accessible from local inner class only if they are final. If we try to access its methods non – final variables, it leads to CE: “local variable is accessed from within inner class; needs to be declared final”.

class Example

{

int a = 10;

static int b = 20;

static void m1(final int e,int f)

{

int c = 30;

final int d = 40;

class A

{

void m1()

{

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

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

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

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

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

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

}

}

}

void m2(final int i,int j)

{

int g = 30;

final int h = 40;

class B

{

void m2()

{

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

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

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

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

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

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

}

}

}

public static void main(String args[])

{

m1(1,1);

Example e = new Example();

e.m2(1,1);

}

}

Error:

non - static variable a cannot be referenced from a static context

local variable c is accessed from within inner class; needs to be declared final

local variable f is accessed from within inner class; needs to be declared final

local variable g is accessed from within inner class; needs to be declared final

local variable j is accessed from within inner class; needs to be declared final

**How can we access and execute method local inner class members?**

Method local inner class members are allowed to access only inside its enclosing method after its definition. This rule is same like local variable rule that is local variable is accessible only inside the method after its creation statement.

**Rule:** If we access method local inner class before its definition or from outside methods, it leads CE: “**cannot find symbol**”.

class Example

{

static void m1()

{

A a1 = new A();

class A

{

int x = 10;

}

System.out.println(a1.x);

}

public static void main(String args[])

{

m1();

}

}

Error: cannot find symbol

**Solution**

class Example

{

static void m1()

{

class A

{

int x = 10;

}

A a1 = new A();

System.out.println(a1.x);

}

public static void main(String args[])

{

m1();

}

}

Output: 10

**How can we access method local inner class members from outside the class?**

We must follow either of the following two approaches:

1. We must return local inner class object from its enclosing method **or**
2. We must send local inner class object as argument to outer class.

In maximum cases in project, we return local inner class object from its enclosing method.

**Procedure for returning or sending local inner class object**

We must send super class name as return type and parameter of the method.

**Why should we use super class name?**

Because we cannot use the local inner class name before its declaration or outside of its enclosing method.

class Example

{

A m1()

{

class A

{

void m2()

{

System.out.println("A m2()");

}

}

A a = new A();

a.m3(a);

return a;

}

void m3(A a)

{

a.m2();

}

}

Error: cannot find symbol at arrow marked places

**What is the default super class of all inner classes, is it java.lang.Object?**

Yes, for every outer class, and inner class java.lang.Object is the super class.

**So, can we use it as return type and parameter for sending local inner class object?**

Yes, but we cannot call methods of this local inner class, because compiler searches method definition in Object not in local inner class.

class Example

{

Object m1()

{

class A

{

void m2()

{

System.out.println("A m2()");

}

}

A a = new A();

m3(a);

return a;

}

void m3(Object obj)

{

obj.m2();

}

}

Error: cannot find symbol at arrow marked place

**Solution:**

We must define a special super class with m3() method declaration. So, this super class should be an interface type.

1. If we have to access method local inner class from the class in which it is defined, then we must send local inner class object as argument to the outer class method.

interface B

{

void m2();

}

class Example

{

void m1()

{

class A implements B

{

public void m2()

{

System.out.println("A m2()");

}

}

A a = new A();

m3(a);

}

void m3(B b)

{

b.m2();

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

}

}

Output

A m2()

1. If we have to access method local inner class from outside the class in which it is defined, then we must return local inner class object from its enclosing method.

interface B

{

void m2();

}

class Example

{

B m1()

{

class A implements B

{

public void m2()

{

System.out.println("A m2()");

}

}

A a = new A();

return a;

}

public static void main(String args[])

{

Example e = new Example();

e.m1();

}

}

class Sample

{

public static void main(String args[])

{

Example e = new Example();

B b = e.m1();

b.m2(); // object of class A is assigned to B

}

}

Output

A m2()

**How can we differentiate outer class method variable name from inner class method variable if both have same name?**

We cannot differentiate outer class method variable from inner class method variable if both have same name.

class Example

{

void m1()

{

final int x = 2;

class A

{

void m2()

{

System.out.println("In A m2() x : " + x);

int x = 4;

System.out.println("In A m2() x : " + x);

}

}

A a = new A();

a.m2();

}

public static void main(String args[])

{

Example a = new Example();

a.m1();

}

}

Output

2

4

**What is the output of the following program?**

class Example

{

int x = 1;

void m1()

{

final int x = 2;

class A

{

int x = 3;

void m2()

{

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

int x = 4;

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

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

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

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

}

}

A a = new A();

a.m2();

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

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

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

}

public static void main(String args[])

{

Example e = new Example();

}

}

Output

x : 3

x : 4

this.x : 3

A.this.x : 3

Example.this.x : 1

x : 2

a.x : 3

this.x : 1

**Anonymous inner class**

Anonymous class is one type of inner class. It is a nameless subclass of a class/interface. Like other inner classes it is not individual class, it is a subclass of some other existed class or interface. Using anonymous inner class we can do three things at a time:

1. Inner class creation as a subclass of outer class.
2. Overriding outer class method
3. Creating and sending its object as argument or return type to another method.

**Syntax**

new outer\_class\_name()

{

//overriding outerclass methods

}

**Example**

new Example()

{}

Note: it is not Example class object creation

new Example();

Note: This is the Example class subject creation