JAVA FUNDAMENTALS

Java:

Java is a programming language, platform & technology.

Java is called as programming language because by using Java we can write programs.

Platform:

It can be a software or hardware environment in which program runs.

C compiler converts unicodeinto bit code whereas Java compiler converts unicodeinto byte code.

JVM:

JVM stands for Java Virtual Machine. It contains interpreter which converts byte code into bit code.

Both compiler & interpreter are called translation softwares.

Differences between compiler & interpreter:

Compiler converts the whole program at a time whereas interpreter converts line by line.

Compiler will produce the file whereas interpreter does not produce the file.

C is a platform dependent whereas Java is a platform independent.

Java is called as platform independent because programs written in Java language can be executed on any platform.

Java Virtual Machine(JVM) is not a platform independent because windows jvm for windows only, linuxjvm for linux only, solariesjvm for solaries only, .. etc.,

C library included in C program at compile time whereas Java library included in Java program at runtime.

Java library is called as Java API(Application Programming Interface) because Java library is an interface between application & programming language.

Java API is a part of JRE(Java Runtime Environment).

JRE contains JVM(Java Virtual Machine) & Java API(Application Programming Interface).JRE is a part of JDK(Java Development Kit).

JDK is called as Java software. Latest version of JDK is 15.

In 1995 Java was developed by James Gosling, Patrick Naughton, Ed Frank, Chris Warth& Mike Sheridan at Sun Microsystems(now owned by Oracle Corporation).

Java slogan is "Write Once, Run Anywhere(WORA)"

Identifiers:

Identifier is a word and it is used to identify variables, methods, classes, interfaces, packages, .. etc.,

Identifier can be a variable name, method name, class name, interface name, package name, .. etc.,

Rules to declare an identifier:

1) It can be formed by using alphabets(A to Z & a to z), digits(0 to 9), underscore symbol(_) and dollar symbol($).
2) It must begins with alphabet, underscore symbol(_) or dollar symbol($).
3) The length of the identifier is not limited.
4) It should not contain special symbol other than underscore & dollar symbols.
5) It should not contain white space characters(Space bar, tab & enter keys).

Keywords:

A set of words reserved by language itself and those words are called keywords.

Examples:

int, char, float, double, if, else, while, for, do, private, protected, public, static, final, void, assert, enum, class, interface, package, ..etc.,

All keywords must be written in lowercase letters only.

There are at present total 50 keywords including strictfp, assert &enum.

strictfp keyword added in JDK 1.2 version in 1998.

assert keyword added in JDK 1.4 version in 2002.

enum keyword added in JDK 1.5 version in 2004.

Note1:const &goto keywords presently not in use.

Note2:Keyword cannot be used as an identifier.

Literals:

A literal is a source code representation of a fixed value.

In Java, literals are divided into 6 categories:

1) Integer Literals:

Examples: 5, 9, 13, 467, 0, -2, -98, -987

2) Floating Point Literals:

Examples: 2.46, 0.08, -2.46, -999.3566

3) Character Literals:

Examples: 'a', 'x', 'A', 'Z', 'c'

4) String Literals:

Examples: "a", "hi", "hello", "welcome"

5) Boolean Literals:

Examples: true, false

6) Object Literal:

Example: null

Note1: true, false & null are not keywords.

Note2:true, false & null are also cannot be used as an identifier.

Data Types:

A data type that determines what value variable can hold and what are theoperations can performed on variables.

In Java, data types are divided into 2 categories:

1) Primitive Data Types

2) Reference Data Types

1) Primitive Data Types:

Primitive data types are predefined data types and these are named by keywords.

These are divided into 4 sub categories:

1) Integers:byte, short, int, long

2) Floating Point Numbers:float, double

3) Characters:char

4) Boolean:boolean

2) Reference Data Types:

Arrays, Strings, Classes, Interfaces, .. etc.,

Variables:

A variable is a container that contains data.

Declaration:

Syntax: DataType Variable;

Example: int x;

Assignment:

Syntax: Variable=Literal;

Example: x=10;

Initialization:

Syntax: DataType Variable=Literal;

Example: int x=10;

If the number is out of int range & within long range(long literal) then it must be suffixed with l or L.

Every float literal must be suffixed with f or F;

ASCII Code:

Digits starts with 48, Capital letters starts with 65 & Small letters starts with 97.

Type Conversions:

boolean type cannot be converted to other types & other types cannot be converted to boolean type.

The following 19 conversions are done by system implicitly. These conversions are called widening conversions:

byte to short, int, long, float, double => 5

short to int, long, float, double => 4

int to long, float, double => 3

long to float, double => 2

float to double => 1

char to int, long, float, double => 4

==========

Total =>19

==========

The following 23 conversions are must be done by programmer explicitly otherwise compile time error occurs. These conversions are called narrowing conversions.

byte to char	=> 1
short to byte, char	=> 2
int to byte, short, char	=> 3
long to byte, short, int, char	=> 4
float to byte, short, int, long, char	=> 5
double to byte, short, int, long, float, char	=> 6
char to byte, short	=> 2

==========

Total =>23

==========

By

Mr. Venkatesh Mansani

Naresh i Technologies

JAVA FUNDAMENTALS

Operations on data:

1) (byte, short, int, char) + (byte, short, int, char) => int
2) (byte, short, int, long, char) + long => long
3) (byte, short, int, long, float, char) + float => float
4) (byte, short, int, long, float, double, char) + double => double

Note: Operations cannot be performed on booleantypes.

Declaration rules to a source file(.java file):

1) A source file can have only one public class.
2) A source file can have any number of non public classes.
3) If the source file contains public class then file name must match with public class name.
4) If the source file does not contain public class then no naming restrictions to a public class name.

Java Program Structure:

```
class Demo
{
    public static void main(String args[])
    {
        System.out.println("Welcome to Java");
    }
}
```

In the above example String & System classes are the part of java.lang package.

java.lang package is a default package and it is implicitly imported in every java program.

Compilation:

C:\>javac Demo.java

The Java compiler generates .class file for every class in a source file.

Execution:

C:\>java Demo

A class that contains main() method only can be used to execute the program.

javac& java are called jdk tools. All jdk tools are the part of bin folder in JDK.

JDK stands for Java Development Kit. It is called as Java Software. Latest version of JDK is 15.

Download & Install JDK:

To download JDK, type the following url in address bar in a browser window.

https://www.oracle.com/in/java/technologies/javase-downloads.html

Click on JDK Download hyper link, scoll down, click on jdk-15.0.1_windows-x64_bin.exe hyper link for windows 64 bit operating system.

Accept license agreement check box & click on download button.

Note: JDK 15 not available for windows 32 bit operating system.

After downloading double click on downloaded file to install.

Path Setting:

Right click on **This PC** icon, click on P**roperties** menu item, click on **Advanced System Settings**, click on **Environment Variables** button, click on **New** button under user variables type **Variable Name** is path & type **Variable Value** is C:\Program Files\Java\jdk-15\bin; Click on **Ok** button.

If the path variable already exists then select the path variable, click on edit button, append the path C:\Program Files\Java\jdk-15\bin;

Note: Before setting the path check jdk folder name in Java folder.

JAVA FUNDAMENTALS

Operaters: An operator is a special symbol that operates on data.

Operators are divided into 3 categories:

1) Unary Operators
2) Binary Operators
3) Ternary Operators

Unary Operators:

An operator that operates on only one operand is called as unary operator.

Examples: ++a, a++, --a, a--, +a, -a, !a, .. etc.,

Binary Operators:

An operator that operates on two operands is called as binary operator.

Examples:a+b, a-b, a*b, a/b, a%b, a<b, a>b, a<=b, a>=b, a==b, a!=b, .. etc.,

Ternary Operators:

An operator that operates on three operands is called as ternary operator.

Example: Conditional operator(? :)

Operators are divided into many categories based on their operations:

1) Arithmetic Operators
2) Relational Operators
3) Logical Operators
4) Increment & Decrement Operators
5) Bitwise Operators
6) Assignment Operators
7) Conditional Operators
8) Other Operators

1) Arithmetic Operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Division

Division operator returns quotient whereas modulo division operator returns remainder.

2) Relational Operators:

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equals to
>=	Greater than or equals to
==	Equals to
!=	Not equals to

The above all operators returns boolean value.

3) Logical Operators:

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND operator returns true if both the expressions returns true, otherwise returns false.

Logical OR operator returns false if both the expressions returns false, otherwise returns true.

Logical NOT operators reverse the logical state.

4) Increment & Decrement Operators:

Operator	Meaning
++	Increment
--	Decrement

Pre IncrementExample:

```
class Demo
{
    public static void main(String args[])
    {
        int a=5;
        int b=++a;
        System.out.println(a);
        System.out.println(b);
    }
}
```

In the above example, value of a incremented by 1 then assigned to b because it is a pre increment operator.

Post Increment Example:

```
class Demo
{
    public static void main(String args[])
    {
```

```
        int a=5;
        int b=a++;
        System.out.println(a);
        System.out.println(b);
    }
}
```

In the above example, value of a assigned to b then a value incremented by 1 because it is a post increment operator.

5) Bitwise Operators:

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left Shift
>>	Right Shift
~	tilde

Note: All bitwise operators operate on binary data.

6) Assignment Operators:

Operator	Meaning
=	Normal Assignment
a+=b	a=a+b
a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b

a%=b	a=a%b
a&=b	a=a&b
a|=b	a=a|b
a^=b	a=a^b
a<<=b	a=a<<b
a>>=b	a=a>>b

7) Conditional Operator(?:):

It is used to express the condition.

Example:

```
class Demo
{
    public static void main(String args[])
    {
      int a=5, b=3;
      int c=(a>b)?a:b;
      System.out.println(c);
    }
}
```

In the conditional operator statement, if the condition returns true then a value stored in c otherwise b value stored in c.

8) Other Operators:

Operator	Meaning
[]	Array Operator
()	Type Cast Operator

+	Unary Plus
-	Unary Minus
instanceof	Instance Of Operator
.	Member Selection Operator
()	Method Call Operator .. etc.,

By

Mr. Venkatesh Mansani

Naresh i Technologies

STATEMENTS IN JAVA

Java Statements are divided into 3 categories:

1) Selection Statements
2) Iteration Statements(Loops)
3) Jump Statements

1) Selection Statements:

i) if Statement

ii) if else Statement

iii) if else if …. else Statement

iv) Nested if Statement

v) switch Statement

2) Iteration Statements(Loops):

i) while loop

ii) do while loop

iii) for loop

iv) Enhanced for loop (or) for each loop

v) Nested loops

3) Jump Statements:

i) break Statement

ii) break LABEL Statement

iii) continue Statement

iv) continue LABEL Statement

v) return Statement

i) if Statement:

It is used to express the condition. If block is executed if the condition is true.

Example:

```
class Demo
{
	public static void main(String args[])
	{
		int a=5;
		if(a>0)
		{
			System.out.println("Positive Number");
		}
	 }
}
```

ii) if else Statement:

It is also used to express the condition. If block is executed if the condition is true otherwise else block is executed.

Example:

```
class Demo
{
	public static void main(String args[])
	{
		int a=5;
		if(a>0)
```

```
        {
            System.out.println("Positive Number");
        }
        else
        {
            System.out.println("Negative Number");
        }
    }
}
```

iii) if else if …. else Statement:

It is used to execute one block of code among multiple blocks of code. Else block is executed if all the conditions are false.

Example:

```
class Demo
{
    public static void main(String args[])
    {
        int a=5;
        if(a>0)
        {
            System.out.println("Positive Number");
        }
        else if(a<0)
        {
```

```
                System.out.println("Negative Number");
        }
        else
        {
                System.out.println("Zero");
        }
    }
}
```

iv) Nested if Statement:

If statement in another if statement is called as nested if statement.

Example:

```
class Demo
{
    public static void main(String args[])
    {
        int a=5;
        if(a>0)
        {
            if(a%2==0)
            System.out.println("Even Number");
            else
            System.out.println("Odd Number");
        }
        else
```

```
        {
            if(a<0)
            System.out.println("Negative Number");
            else
            System.out.println("Zero");
        }
    }
}
```

i) while loop:

The body of while loop is repeatedly executed until the condition becomes false.

Example:

```
class Demo
{
    public static void main(String args[])
    {
        int i=1;
        while(i<=10)
        {
            System.out.println(i);
            i++;
        }
    }
}
```

ii) do while loop:

The body of do while loop is repeatedly executed until the condition becomes false.

Do while loop is executed at least once

Example:

```
class Demo
{
	public static void main(String args[])
	{
		int i=1;
		do
		{
			System.out.println(i);
			i++;
		}while(i<=10);
	}
}
```

iii) for loop:

The body of for loop is repeatedly executed until the condition becomes false.

It supports to write initialization expression, test expression and updation expression in a one line.

Example:

```
class Demo
{
	public static void main(String args[])
	{
```

```
        for(int i=1;i<=10;i++)
        {
            System.out.println(i);
        }
    }
}
```

i) break Statement:

It terminates the nearest enclosing loop or switch statement.

Example:

```
class Demo
{
    public static void main(String args[])
    {
      for(int i=1;i<=10;i++)
      {
            if(i==5)
              break;
            System.out.println(i);
      }
    }
}
```

ii) break LABEL Statement:

It terminates the specified LABEL loop. Here break is a keyword & LABEL is an identifier.

Example:

```
class Demo
{
	public static void main(String args[])
	{
	  FIRST: for(int i=1;i<=3;i++)
	  {
		SECOND: for(int j=1;j<=10;j++)
		{
		   if(j==5)
		     break FIRST;
		   System.out.println(j);
		}
	  }
	}
}
```

iii) continue Statement: It passes the control to the next iteration of a loop.

Example:

```
class Demo
{
	public static void main(String args[])
	{
	  for(int i=1;i<=10;i++)
	  {
```

```
            if(i==5)
              continue;
            System.out.println(i);
        }
    }
}
```

iv) continue LABEL Statement:

It passes the control to the next iteration of specified LABEL loop.

Example:

```
class Demo
{
    public static void main(String args[])
    {
      FIRST: for(int i=1;i<=3;i++)
      {
          SECOND: for(int j=1;j<=10;j++)
          {
              if(j==5)
                continue FIRST;
              System.out.println(j);
          }
      }
    }
}
```

OBJECT ORIENTED PROGRAMMING (OOP)

Java is an object oriented programming language (OOPL).

OOPL:

A language that supports all the principles of an object oriented programming is known as an object oriented programming language.

Object Oriented Principles(OOPs):

1) Encapsulation
2) Abstraction
3) Inheritance
4) Polymorphism

Object Oriented = Object Based + Inheritance + Runtime Polymorphism

Object Oriented Languages:

C++, Java, VC++, C#.Net, Visual Basic.Net, Simula, Ada, Small Talk, Python, .. etc.,

Object Based Languages:

Java Script, VB Script, Visual Basic, .. etc.,

In order to use the above principles in a Java programming we need the following language constructs:

1) Class
2) Object

Class:

A class is a collection of variables & methods.

The syntax of a class:

```
class ClassName
{
        DataType Variable1;
```

```
        DataType Variable2;
        ================
        ReturnType MethodName1(arg1, arg2, … )
        {
                ==================
        }
}
```

Class will not occupy memory where as file occupies memory.

Object:

Object is an instance of a class.

The syntax to create an object:

ClassName ObjectReference=new Constructor();

new:

It is called as dynamic memory allocation operator and it allocates the memory to instance variables at run time.

Object occupies memory and object reference also occupies memory.

Object contains data whereas an object reference contains hash code.

Anonymous Object:

Syntax: new Constructor();

Example: new Demo();

It is also called as unreferenced object.

Object reference can be created without object also. That object reference contains null.

null is an object literal.

Variables:

A variable is a container that contains data.

There are 3 types of variables in Java:

1) Instance Variables
2) Class Variables
3) Local Variables

1) Instance Variables:

A variable that is defined as a member of a class is known as an instance variable.

Memory allocated to instance variables whenever an object is created.

Instance variables are stored in heap area.

2) Class Variables:

A variable that is defined as a static member of a class is known as class variable.

Memory allocated to class variables whenever class is loaded.

Class variables are stored in method area.

3) Local Variables:

A variable that is defined inside a method is known as local variable.

Memory allocated to local variables whenever method is called.

Local variables are stored in stack area.

Note1: Local variables cannot be static in Java.

Note2: No global variables in Java (Outside the class).

Execution Priority:

1) Class Variables
2) Static Blocks
3) Main Method

Class can also be called as reference data type & Object reference can also be called as reference variable.

Primitive type variable contains data whereas reference type variable contains hash code.

Separate copy of instance variable exists for every object whereas only one copy of class variable exists for all objects.

Example:

```
class Demo
{
	int x=10;
	static int y=20;
	public static void main(String args[])
	{
		int z=30;
		Demo d1=new Demo();
		Demo d2=new Demo();
	}
}
```

There are two ways to access an instance variable:

1) By using object
2) By using object reference

Use object to access an instance variable if it is required only one time.

Use object reference to access an instance variable if it is required more than one time.

There are four ways to access class variable:

1) Directly
2) By using class name
3) By using object
4) By using object reference

3rd & 4th ways are not recommended to use.

Access class variable directly if it is present in the same class.

Use class name to access class variable if it is present in another class.

There is a only one way to access local variable: i.e. directly.

Instance Variable Example:

```
class Demo
{
	int x=10;
	public static void main(String args[])
	{
		System.out.println(new Demo().x);
		Demo d=new Demo();
		System.out.println(d.x);
	}
}
```

Class Variable Example:

```
class Demo
{
	static int x=10;
	public static void main(String args[])
```

```
    {
        System.out.println(x);
        System.out.println(Demo.x);
        System.out.println(new Demo().x);
        Demo d=new Demo();
        System.out.println(d.x);
    }
}
```

Local Variable Example:

```
class Demo
{
    public static void main(String args[])
    {
        int x=10;
        System.out.println(x);
    }
}
```

Note: Local variables must be initialized before access otherwise compile time error occurs because they do not get default values.

```
class Demo
{
    int x=10;
    int y=20;
    public static void main(String args[])
    {
```

```
            int z=10;
            for(int i=1;i<=10;i++)
            {
                  ============
            }
      }
}
```

In the above example x is an instance variable, y is a class variable, z is a local variable, args is a method parameter & i is a block variable

Instance variables & Class variables scope is based on access modifiers.

Method parameters & local variables scope is limited to method only.

Block variables scope is limited to block only.

Note: Method parameters & block variables are also called as local variables.

Use instance variable if the value is different for objects.

Use class variable if the value is same for all objects.

Use local variable to perform the task.

By

Mr. Venatesh Mansani

Naresh i Technologies

OBJECT ORIENTED PROGRAMMING (OOP)

Arrays:

An array is a collection of similar data elements.

Array index always begins with 0 and ends with size-1.

An array itself an object in Java. Array reference is also called as an object reference or reference variable.

Declaration:

Syntax: DataType ArrayReference[]=new DataType[size];

Example: int a[]=new int[5];

(or)

int []a=new int[5];

(or)

int[] a=new int[5];

Assignment:

Syntax: ArrayReference[index]=Literal;

Examples: a[0]=10;

a[1]=98;

Initialization:

Syntax: DataType ArrayReference[]={Literal1, Literal2, …….}

Example: int[] a={13, 83, 81, 45, 83};

Accessing array elements by using for loop:

Example

```
class Demo
{
	public static void main(String args[])
	{
		int[] a={43, 31, 88, 13, 54};
		for(int i=1;i<=10;i++)
		{
			System.out.println(a[i]);
		}
	}
}
```

Accessing array elements by using enhanced for loop or for each loop:

Example:

```
class Demo
{
	public static void main(String args[])
	{
		int[] a={43, 31, 88, 13, 54};
		for(int b : a)
		{
			System.out.println(b);
```

```
            }
        }
}
```

Methods:

A group of statements into a single logical unit is called as method.

Methods are used to perform the task.

Task code must be written in a method only.

Methods are divided into four categories:

1) Methods with arguments and with return value.

2) Methods with arguments and without return value.

3) Methods without arguments and with return value.

4) Methods without arguments and without return value.

Advantages of methods:

1) Modularity

2) Reusability

1) Methods with arguments and with return value:

Syntax:

```
ReturnType MethodName(arg1, arg2, ........)
{
    ============
}
```

ReturnType can be primitive type or reference type. Arguments also can be primitive type or reference type

Instance Method:

A method that is defined as a member of a class is known as an instance method.

There are two ways to access an instance method:

1) By using object

2) By using object reference

Use object to access an instance method if it is required only one time.

Use object reference to access an instance method if it is required more than one time.

Example:

```
class Demo
{
	int add(int a, int b)
	{
		int c=a+b;
		return c;
	}
	public static void main(String args[])
	{
		int x=new Demo().add(4, 3);
		System.out.println(x);
		Demo d=new Demo();
		int y=d.add(5, 3);
		System.out.println(y);
	}
```

```
}
```

2) Methods with arguments and without return value:

Syntax:

```
ReturnType MethodName(arg1, arg2, ........)
{
        ============
}
```

ReturnType is an empty data type(void). Arguments also can be primitive type or reference type

Example:

```
class Demo
{
        void add(int a, int b)
        {
                int c=a+b;
                System.out.println(c);
        }
        public static void main(String args[])
        {
                Demo d=new Demo();
                d.add(5, 3);
        }
}
```

3) Methods without arguments and with return value:

Syntax:

```
ReturnType MethodName()
{
        ============
}
```

ReturnType can be primitive type or reference type.

Example:

```
class Demo
{
        int get()
        {
                int a=5;
                return a;
        }
        public static void main(String args[])
        {
                Demo d=new Demo();
                int x=d.get();
                System.out.println(x);
        }
}
```

4) Methods without arguments and without return value:

Syntax:

```
ReturnType MethodName()
{
    ============
}
```

ReturnType is an empty data type(void).

Example:

```
class Demo
{
    void display()
    {
        System.out.println("Welcome");
    }
    public static void main(String args[])
    {
        Demo d=new Demo();
        d.display();
    }
}
```

Method Overloading:

If two or more methods with the same name and with different parameters list then it is said to be method overloading.

The parameters can be different in no. of arguments, data types or order of an arguments.

Return type can be same or different in method overloading.

Examples:

1) void add(int a, int b){}

   int add(int a, int b, int c){}

2) void add(int a, int b){}

   void add(float a, float b){}

3) void add(int a, float b){}

   void add(float a, int b){}

Method Overriding:

If two or more methods with the same and with the same parameters list then it is said to be method overriding.

Example:

void add(int a, int b){}

void add(int x, int y){}

Note: Methods cannot be overridden in the same class because of ambiguity to call.

"static" keyword:

It is called as modifier because it modifies the behaviour of a variable, method, class & block.

By using static keyword we can create class variables, class methods, nested top level classes (It is a one type of inner class) & static initialization blocks.

Class Methods:

A method that is defined as a static member of a class is called as class method.

There are four ways to access class method:

1) Directly

2) By using Class Name

3) By using Object

4) By using Object Reference

3rd & 4th way are not recommended to use.

Access class method directly if it is present in the same class.

Access class method by using class name if it is present in another class.

Note: If the class contains instance variable then use instance method otherwise use class method.

"this" keyword:

It is called as an object reference or reference variable because it refers an object.

It always refers current object.

It is implicitly present in instance method, initialization block & constructor.

It is explicitly required to access an instance variable whenever both instance variable & local variable names are same.

Static method (class method) does not refer this keyword in anyway

Example1:

```
class Demo
{
	int x=10;
	void display()
	{
		System.out.println(x); //here implicitly this.x
	}
	public static void main(String args[])
```

```
        {
                Demo d=new Demo();
                d.display();
        }
}
```

Example2:

```
class Demo
{
        int x=10;
        void display()
        {
                System.out.println(x); //here implicitly this.x
        }
        public static void main(String args[])
        {
                Demo d1=new Demo();
                Demo d2=new Demo();
                d1.x=d1.x+2;
                d1.display();
                d2.display();
        }
}
```

Example3:

```
class Demo
{
	int x=10;
	void display()
	{
		int x=20;
		System.out.println(x);
		System.out.println(this.x);
	}
	public static void main(String args[])
	{
		Demo d=new Demo();
		d.display();
	}
}
```

Example4:

```
class Demo
{
	void show()
	{
		System.out.println("show() method");
	}
	void display()
```

```
    {
        System.out.println("display() method");
        show(); //implicitly this.show();
    }
    public static void main(String args[])
    {
        Demo d=new Demo();
        d.display();
    }
}
```

By

Mr. Venatesh Mansani

Naresh i Technologies

OBJECT ORIENTED PROGRAMMING (OOP)

Constructors:

A constructor is a special method which has same name as the class name and which has no return type.

Constructor is called automatically whenever an object is created.

Constructors are used to initialize instance variables.

Constructors are two types:

1) Default constructor (without arguments)

2) Parameterized constructor (with arguments)

Example1:

```
class Demo
{
	Demo()
	{
		System.out.println("Welcome");
	}
	Demo(int x)
	{
		System.out.println(x);
	}
	public static void main(String args[])
	{
		new Demo();
		new Demo(5);
```

```
	}
}
```

Example2:

```
class Emp
{
	int empNo;
	float salary;
	Emp(int empNo, float salary)
	{
		this.empNo=empNo;
		this.salary=salary;
	}
}
class Demo
{
	public static void main(String args[])
	{
		Emp e1=new Emp(101, 5000.00f);
		Emp e2=new Emp(102, 5500.00f);
		System.out.println(e1.empNo+"\t"+e1.salary);
		System.out.println(e2.empNo+"\t"+e2.salary);
	}
}
```

this(); => It calls default constructor of current class.

this(arg1, arg2, ..); => It calls parameterized constructor of current class.

this(); (or) this(arg1, arg2, ...); must be a first statement in a constructor.

Example3:

```
class Demo
{
	Demo()
	{
		System.out.println("Welcome");
	}
	Demo(int x)
	{
		this();
		System.out.println(x);
	}
	public static void main(String args[])
	{
		new Demo(5);
	}
}
```

Note: If the class does not contain any other constructor then only Java compiler implicitly provides default constructor.

Blocks:

A group of statements between { and } is called as block.

A block is also called as an initialization block because block also can be used to initialize instance variables.

A class can have any number of blocks and all those blocks are executed from top to bottom order whenever an object is created.

Blocks are executed before constructor only.

Example:

```
class Demo
{
	{
		System.out.println("Block1");
	}
	Demo()
	{
		System.out.println(“Constructor”);
	}
	public static void main(String args[])
	{
		new Demo();
		System.out.println("Main Method");
	}
	{
		System.out.println("Block2");
	}
}
```

Static Blocks:

A block with static keyword is called as static block.

A static block is also called as static initialization block because static block can also be used to initialize static variables.

A class can have any number of static blocks and all those static blocks are executed from top to bottom order whenever class is loaded.

Static blocks are executed before main() method only.

Static blocks are used to load libraries in a program.

Example:

```
class Demo
{
	static
	{
		System.out.println("Static Block1");
	}
	public static void main(String args[])
	{
		System.out.println("Main Method");
	}
	static
	{
		System.out.println("Static Block2");
	}
}
```

INHERITANCE

Inheritance:

Creating a new class from an existing class is called as an inheritance.

In inheritance, existed class said to be super class and new class is said to be sub class.

Whenever super class object is created then memory allocated to super class members only.

Whenever sub class object is created then memory allocated to both super class & sub class members.

There are 6 types of inheritance:

1) Single Inheritance

2) Multiple Inheritance

3) Multilevel Inheritance

4) Hierarchical Inheritance

5) Multipath Inheritance

6) Hybrid Inheritance

1) Single Inheritance:

Derivation of a class from only one super class is called as single inheritance.

2) Multiple Inheritance:

Derivation of a class from more than one super class is called as multiple inheritance.

3) Multilevel Inheritance:

Derivation of a class from sub class is called as multilevel inheritance.

4) Hierarchical Inheritance:

Derivation of several classes from only one super class is called as hierarchical inheritance.

5) Multipath Inheritance:

Derivation of a class from more than one sub class, those sub classes get inherited from the same super class is called as multipath inheritance.

6) Hybrid Inheritance:

Derivation of a class involving more than one form of inheritance is called as hybrid inheritance.

Note1: Multipath inheritance is also one example of hybrid inheritance.

Note2: Java does not support multiple inheritance, multipath inheritance & hybrid inheritance.

Java does not support multiple inheritance because of ambiguity to call.

Java does not support multipath & hybrid because both types contain multiple inheritance.

Example:

```
class A
{
	int x=5;
	void show()
	{
		System.out.println("A class");
	}
}
class B extends A
```

```
{
        int y=10;
        void print()
        {
                System.out.println("B class");
        }
        public static void main(String args[])
        {
                B ob=new B();
                System.out.println(ob.x);
                System.out.println(ob.y);
                ob.show();
                ob.print();
        }
}
```

"super" keyword:

It is called as an object reference or reference variable because it refers super class memory.

It is explicitly required to access super class member whenever both super class & sub class member names are same.

Static method does not refer this & super keywords in anyway.

Example1:

```
class A
{
        int x=5;
```

```
}
class B extends A
{
	int x=10;
	void print()
	{
		int x=15;
		System.out.println(x);
		System.out.println(this.x);
		System.out.println(super.x);
	}
	public static void main(String args[])
	{
		B ob=new B();
		ob.print();
	}
}
```

Example2:

```
class A
{
	void show()
	{
		System.out.println("A class");
	}
```

```
}
class B extends A
{
	void show()
	{
		System.out.println("B class");
	}
	void print()
	{
		show();
		this.show();
		super.show();
	}
	public static void main(String args[])
	{
		B ob=new B();
		ob.print();
	}
}
```

super(); => It calls default constructor of super class. It is implicitly present in every constructor as a first line.

super(arg1, arg2, ..); => It calls parameterized constructor of super class.

Example:

```
class A
```

```
{
        A()
        {
                this(20);
                System.out.println("A class");
        }
        A(int x)
        {
                System.out.println(x);
        }
}
class B extends A
{
        B()
        {
                this(10);
                System.out.println("B class");
        }
        B(int y)
        {
                System.out.println(y);
        }
        public static void main(String args[])
        {
```

```
            new B();
        }
}
```

Note: this(); (or) this(arg1, arg2, ..); (or) super(); (or) super(arg1, arg2, ..); must be a first statement in a constructor.

"final" keyword:

It is called as modifier because it modifies the behaviour of a variable, method & class.

By using final keyword, we can prevent value of the variable, method overriding & inheritance.

final keyword can be applied to instance variables, class variables & local variables.

final variable must be initialized otherwise compile time error occurs.

final values cannot be changed

final methods cannot be overridden

final classes cannot be inherited

Access Modifiers:

Access modifiers are also called as access specifiers because they specify access permissions to variables, methods, classes, interfaces, .. etc.,

There are four access modifiers

1) private 2) protected 3) public 4) no name(default)

private scope is limited to class only.

protected scope is limited to package & sub class(Sub class can be out side the package also).

public scope is not restricted.

No name(default) scope is limited to package only.

In Java, outer class cannot be private and cannot be protected.

In Java, outer interface also cannot be private and cannot be protected.

private ======> no name(default) ========> protected ========> public

private is more restrictive access modifier.

public access modifier has no restrictions.

Relationships In Java:

1) “IS-A” Relationship
2) “HAS-A” Relationship

“IS-A” Relationship refers Inheritance where as “HAS-A” Relationship refers composition

“IS-A” Relationship Example:

```
class A
{
	void show()
	{
		System.out.println("Welcome");
	}
}
class B extends A
{
	public static void main(String args[])
	{
		B ob=new B();
		ob.show();
	}
```

```
}
```

"HAS-A" Relationship Example:

```
class A
{
	void show()
	{
		System.out.println("Welcome");
	}
}
class B
{
	A a=new A();
	public static void main(String args[])
	{
		B b=new B();
		b.a.show();
	}
}
```


By

Mr. Venatesh Mansani

Naresh i Technologies

POLYMORPHISM, ABSTRACTION & ENCAPSULATION

Polymorphism:

The ability to take more than one form. Poly means many, morphism means forms and Polymorphism means many forms.

There are two types of polymorphism.

1) Compile time polymorphism(Static polymorphism)

2) Run time polymorphism(Dynamic polymorphism)

1) Compile time polymorphism:

Binding of method call statement with method definition is done at compile time is known as compile time polymorphism.

Example:

Method Overloading.

Method Overloading:

If two or more methods with the same name and with different parameters list then it is said to be method overloading.

In method overloading, the parameters list can be different in no. of arguments, data types or order of an arguments.

In method overloading, return type can be same or different.

Example:

```
class Demo
{
        void add(int a, int b)
        {
```

```
            System.out.println(a+b);
        }
        void add(int a, int b, int c)
        {
            System.out.println(a+b+c);
        }
        public static void main(String args[])
        {
            Demo d=new Demo();
            d.add(43, 53);
            d.add(38, 45, 34);
        }
}
```

2) Run time polymorphism:

Binding of method call statement with method definition is done at run time is known as run time polymorphism.

Example:

Method Overriding.

Method Overriding:

If two or more methods with the same name and with the same parameters list then it is said to be method overriding.

Note: Methods cannot be overridden in the same class because of ambiguity.

Methods can be overridden only in inheritance.

Example:

class A

```
{
        void show()
        {
                System.out.println("A class");
        }
}
class B extends A
{
        void show()
        {
                System.out.println("B class");
        }
        public static void main(String args[])
        {
                A ob=new A();
                     (or)
                A ob=new B();
                ob.show();
        }
}
```

Method Overloading Vs. Method Overriding

Method Overloading	Method Overriding
1) If two or more methods with the same name and with different	1) If two or more methods with the same name and with the same parameters

parameters list, then it is said to be method overloading.	list, then it is said to be method overriding.
2) In method overloading, return type can be same or different	2) In method overriding, return type must be same except covariant return type. **Covariant return type:** In method overriding, Java permits sub class type as a return type. This is known as covariant return type
3) Methods can be overloaded in the same class also.	3) Methods cannot be overridden in the same class because of ambiguity to call
4) Methods can be overloaded in inheritance also.	4) Methods can be overridden only in inheritance.
5) final methods can be overloaded.	5) final methods cannot be overridden because final keyword used to prevent overriding.
6) static methods can be overloaded.	6) static methods cannot be overridden because static method does not require object to call.

7) In method overloading, access modifiers can be same or different.	7) In method overriding, overriding method can have same access modifier or less restrictive access modifer.
8) private methods can be overloaded.	8) private methods cannot be overridden because private methods cannot be inherited.

Upcasting:

Assigning an object or object reference of sub class to super class type is known as upcasting.

Upcasting done by system implicitly.

Upcasting always valid.

Examples:

1) A ob=new B();=> Valid upcasting

2) B ob1=new B();

A ob2=ob1;=> Valid upcasting

Downcasting:

Assigning an object or object reference of super class to sub class type is known as downcasting.

Downcasting must be done by programmer explicitly otherwise compile time error occurs.

Downcasting always needs upcasting to get memory.

1) B ob=new A();=> Invalid downcasting

2) A ob1=new A();

B ob2=ob1;=> Invalid downcasting

3) B ob=(B)new A();=> Invalid downcasting

4) A ob1=new A();

B ob2=(B)ob1;=> Invalid downcasting

5) A ob1=new B();=> Valid upcasting

B ob2=ob1;=> Invalid downcasting

6) A ob1=new B();=> Valid upcasting

B ob2=(B)ob1;=> Valid downcasting

Abstraction:

Providing necessary information and hiding unnecessary information.

(or)

Providing necessary information without including background details.

In Java, we use abstract class & interface to implement abstraction.

Abstract class:

A class that is declared with abstract keyword is called as an abstract class.

Abstract class can have only abstract methods, only non abstract methods or both abstract & non abstract methods.

Non abstract methods are also called concrete methods.

Abstract method:

A method that has no body is called as an abstract method.

Abstract method must be declared with abstract keyword in Java, otherwise compile time error occurs.

Concrete method:

A method that has a body is called as concrete method.

abstract void show();=> Abstract method (or) method declaration

```
void print()=> Concrete method (or) Non abstract method
{
    ======
    ======
}
void display(){}=> Null body method
```

Null body method is also called as concrete method.

If the class contains an abstract method then the class must be declared with abstract keyword otherwise compile time error occurs.

Abstract classes cannot be instantiated.

Instantiation means object creation.

Abstract class can be inherited by using extends keyword.

Whenever an abstract class is inherited then all abstract methods of an abstract class must be overridden in a sub class or sub class must be declared with abstract keyword otherwise compile time error occurs.

Example:

```
abstract class A
{
    abstract void show();
    void print()
    {
        System.out.println("print() method");
    }
}
class B extends A
```

```
{
	void show()
	{
		System.out.println("show() method");
	}
	void display()
	{
		System.out.println("display() method");
	}
	public static void main(String args[])
	{
		B ob=new B();
		ob.show();
		ob.print();
		ob.display();
	}
}
```

Abstract methods cannot be final.

Abstract methods cannot be static.

Abstract methods cannot be private.

Abstract class cannot be final.

Abstract class can have constructors and those constructors are called whenever objects are created to sub classes.

Abstract class can have static members also.

Abstract class can have main() method also.

Interfaces:

An interface is a collection of public static final variables & public abstract methods.

In interface all variables are implicitly public static final and all methods are implicitly public abstract.

Every interface implicitly itself abstract.

Interfaces cannot be instantiated.

Interface can be inherited into a class by using implements keyword.

Whenever interface is inherited then all methods of an interface must be overridden in a sub class or sub class must be declared with abstract keyword otherwise compile time error occurs.

Interface can also be inherited in another interface by using extends keyword.

Example:

```
interface A
{
        int x=10;
        void show();
}
class B extends A
{
    int y=20;
        public void show()
        {
                System.out.println("show() method");
        }
```

```
    void display()
    {
        System.out.println("display() method");
    }
    public static void main(String args[])
    {
        System.out.println(A.x);
        B ob=new B();
        System.out.println(B.y);
        ob.show();
        ob.display();
    }
}
```

Note: Class cannot be inherited in interface.

Interfaces are introduced in Java to achieve multiple inheritance.

Encapsulation:

Binding of variables with methods and those methods operating on same variables is known as an encapsulation.

Example:

```
class Emp
{
    int age;
    void setAge(int age)
    {
        if(age<0)
```

```
            this.age=0;
        else if(age>100)
            this.age=100;
        else
            this.age=age;
    }
    int getAge()
    {
        return age;
    }
}
class Demo
{
    public static void main(String args[])
    {
        Emp e=new Emp();
        e.setAge(200);
        int x=e.getAge();
        System.out.println(x);
    }
}
```

In the above example age variable bound with setAge() & getAge() methods and those methods are operating on same age variable. This is known as an encapsulation.

INNER CLASSES & PACKAGES

Inner Classes:

A class that is defined in another class is called as an inner class.

There are four types of inner classes:

1) Member class
2) Static member class
3) Local class
4) Anonymous class

1) Member class:

A class that is defined as member of another class is called as member class.

Example:

```
class A
{
        class B
        {
                void show()
                {
                        System.out.println("Welcome");
                }
         }
}
class Demo
{
        public static void main(String args[])
        {
```

```
        A a=new A();
        A.B b=a.new B();
        b.show();
    }
}
```

2) Static Member Class:

A class that is defined as a static member of another class is called as static member class.

Example:

```
class A
{
    static class B
    {
        void show()
        {
            System.out.println("Welcome");
        }
     }
}
class Demo
{
    public static void main(String args[])
    {
        A.B b=new A.B();
        b.show();
```

```
    }
}
```

3) Local Class:

A class that is defined inside a method is called as local class.

Example:

```
class Demo
{
    public static void main(String args[])
    {
        class Test
        {
            void show()
            {
                System.out.println("Welcome");
            }
        }
        Test t=new Test();
         t.show();
     }
}
```

4) Anonymous Class:

It is a one type of local class which has no name. It is always sub class of a class or interface.

Example:

```
interface Test
{
        void show();
}
class Demo
{
    public static void main(String args[])
    {
        Test t=new Test()
        {
            public void show()
            {
                System.out.println("Welcome");
            }
        };
        t.show();
     }
}
```

Outer classes cannot be private and cannot be protected.
Member class & Static member class can have all access modifiers.
Access modifiers cannot be applied to local class and anonymous class.

Packages:

A package is a collection of sub packages, classes & interfaces.

Example:

```
package pack1;
```

```
public class A
{
	public void add(int a, int b)
	{
		System.out.println(a+b);
	}
}
```

C:\src>javac –d C:\classes A.java

```
package pack1.pack11;
public class B
{
	public void max(int a, int b)
	{
		if(a>b)
		  System.out.println(a);
		else
		  System.out.println(b);
	}
}
```

C:\src>javac –d C:\classes B.java

```
package pack2;
public class C
{
	public void cube(int a)
```

```
        {
                System.out.println(a*a*a);
        }
}
C:\src>javac –d C:\classes C.java
package test;
import pack1.*;
import pack1.pack11.*;
import pack2.*;
class Main
{
        public static void main(String args[])
        {
                A a=new A();
                a.add(83, 45);
                B b=new B();
                b.max(87, 45);
                C c=new C();
                c.cube(8);
        }
}
C:\src>set classpath=”%classpath%”.;C:\classes;
C:\src>javac –d C:\ Main.java
C:\>java test.Main
```

STRING HANDLING & WRAPPER CLASSES

String Handling:

There are four string related classes to handle strings.

1) java.lang.String
2) java.lang.StringBuffer
3) java.lang.StringBuilder
4) java.util.StringTokenizer

Every string literal itself an object of String class in Java.

Example:

“Welcome” is an object of String class.

String s1=”Welcome”;=> This statement creates one object in a string constant pool.

String s2=new String(“Hello”); => This statement creates two string objects (one object created in a string constant pool & one more object created outside the pool)

Note: String constant pool does not create duplicates.

Example:

```
class Demo
{
	public static void main(String args[])
	{
		String s1=new String("hello");
		String s2=new String("hello");
		System.out.println(s1.equals(s2));
		System.out.println(s1==s2);
	}
```

```
}
```

Note: equals() method of String class compares the contents of String objects whereas == operator compares the hash codes.

Differences between String, StringBuffer & StringBuilder

String	StringBuffer	StringBuilder
1) The object of String class is immutable	1) The object of StringBuffer class is mutable.	1) The object of StringBuilder class is mutable.
2) Methods of String class are not synchronized	2) Methods of StringBuffer class are synchronized	2) Methods of StringBuilder class are not synchronized

Immutable object: It means the value of an object cannot be changed.

Synchronization: It is a mechanism that allows to access a shared resource only one thread at a time.

StringTokenizer Class:

It allows an application to break a string into tokens(words).

Example:

"Welcome to Java" is a one string and it has 3 tokens(words).

Program to count the no. of words in a given String

```
import java.util.*;
class Demo
{
        public static void main(String args[])
        {
                String s="Welcome to Java";
```

```
                StringTokenizer st=new StringTokenizer(s);
                int n=st.countTokens();
                System.out.println(n);
        }
}
```

Program to iterate word by word in a given string:

```
import java.util.*;
class Demo
{
        public static void main(String args[])
        {
                String s="Welcome to Java";
                StringTokenizer st=new StringTokenizer(s);
                while(st.hasMoreTokens())
                {
                        System.out.println(st.nextToken());
                }
        }
}
```

Command Line Arguments:

The arguments that are passed at the command prompt are called command line arguments.

Command line arguments are received by main method only.

The arguments are received string format only.

Examples:

1) class Demo

```
{
        public static void main(String args[])
        {
            for(String s : args)
            {
                System.out.println(s);
            }
        }
}
```

2) class Demo

```
    {
        public static void main(String args[])
        {
                System.out.println(args[0]+args[1]);
        }
     }
```

To run the above application:

C:\> java Demo Taj Mahal

Output: TajMahal

C:\> java Demo 10 20

Output: 1020 because command line arguments are received as a string format only.

To convert string format to other formats we need wrapper classes.

Wrapper Classes:

Each of Java's 8 primitive data types has a class and those classes are called wrapper classes because they wrap the data into an object.

List of primitive data types:

1) byte 2) short 3) int 4) long 5) float 6) double 7) char 8) boolean

List of wrapper classes:

1) Byte 2) Short 3) Integer 4) Long 5) Float 6) Double 7) Character 8) Boolean

Example:

```
class Demo
{
	public static void main(String args[])
	 {
		int x=Integer.parseInt(args[0]);
		int y=Integer.parseInt(args[1]);
		System.out.println(x+y);
	 }
}
```

To run the above application:

C:\> java Demo 10 20

Output: 30

To covert primitive type to reference type

```
int a=5;
Integer i=new Integer(a);
```

To covert reference type to primitive type:

```
Integer i=new Integer(5);
int x=i.intValue();
```

Auto boxing:

The process of converting primitive type to the corresponding reference type is known as auto boxing.

Example:

int x=5;

Integer i=x; => It is called as auto boxing

Auto unboxing:

The process of converting reference type to the corresponding primitive type is known as auto unboxing.

Example:

Integer i=new Integer(5);

int x=i; => it is called as auto unboxing.

Both auto boxing and auto unboxing features are introduced in JDK 1.5 verion in 2004.

By

Mr. Venatesh Mansani

Naresh i Technologies

EXCEPTION HANDLING

Exception Handling:

Errors: A programming mistake is said to be an error.

There are three types of errors:

1) Compile time errors (Syntax errors)

2) Runtime errors (Exceptions)

3) Logical errors

Exception means run time error.

In exception handling we use the following keywords.

1) try 2) catch 3) throw 4) throws 5) finally

The syntax of try and catch blocks:

```
try
{
      =============      => Task code
}catch(ExceptionClassName ObjectReference)
{
      =============      => Error Message
}
```

try block must be associated with at least one catch block or finally block.

All exceptions are classes in Java.

Whenever exception occurs in a Java program, then the related exception class object is created by JVM, passed to exception handler (catch block) and exception handler code is executed.

There are two types of exceptions:

1) Checked Exceptions

2) Unchecked Exceptions

Checked Exceptions:

The exception classes that are derived from java.lang.Exception class are called checked exceptions.

Checked exceptions do not include java.lang.RuntimeException class and all its sub classes.

All checked exceptions must be handled explicitly otherwise compile time error occurs.

The Java compiler checks for try & catch blocks or throws clause for this kind of exceptions.

All application specific exceptions are comes under this category.

Unchecked Exceptions:

The exception classes that are derived from java.lang.RuntimeException class are called unchecked exceptions.

All unchecked exceptions are handled by system implicitly.

Handling unchecked exceptions are optional by programmer.

Unchecked exceptions are handled by programmer to display user friendly error messages only.

The Java compiler does not check for try & catch blocks or throws clause for this kind of exceptions.

All general exceptions comes under this category.

Example:

```
class Demo
{
	public static void main(String args[])
	{
		try{
		int x=Integer.parseInt(args[0]);
		int y=Integer.parseInt(args[1]);
		int z=x/y;
		System.out.println(z);
		}catch(ArrayIndexOutOfBoundsException ae)
		{
		    System.err.println("Please pass two arguments");
		}catch(NumberFormatException ne)
		{
				System.err.println("Please pass two numbers only");
		}catch(ArithmeticException ae)
		{
		System.err.println("Please pass second argument except zero");
		}
	}
}
```

Execution:

1) C:\> java Demo

Please pass two arguments

2) C:\> java Demo abc xyz

Please pass two numbers only

3) C:\> java Demo 10 0

Please pass second argument except zero

4) C:\> java Demo 10 2

5

Program to create & handle checked exception:

```
class NegativeNumberException extends Exception
{
}
class Demo
{
      void cube(a) throws NegativeNumberException
      {
            if(a>0)
              System.out.println(a*a*a);
            else
              throw new NegativeNumberException();
  }
  public static void main(String args[])
  {
```

```
        try{
        int x=Integer.parseInt(args[0]);
        Demo d=new Demo();
        d.cube(x);
        }catch(NegativeNumberException ne)
        {
                System.err.println(ne);
        }
}
}
```

Program to create unchecked exception:

```
class NegativeNumberException extends RuntimeException
{
}
class Demo
{
        void cube(a) throws NegativeNumberException
        {
             if(a>0)
               System.out.println(a*a*a);
             else
                throw new NegativeNumberException();
        }
        public static void main(String args[])
```

```
    {
        int x=Integer.parseInt(args[0]);
        Demo d=new Demo();
        d.cube(x);
    }
}
```

Try & Catch Blocks Examples:

1) try
```
{
}catch(Exception e)
{
}
```
The above catch block handles all exceptions because all exceptions are sub classes of java.lang.Exception class.

2) try
```
{
}catch(RuntimeException e)
{
}
```
The above catch block handles all unchecked exceptions because all unchecked exceptions are sub classes of java.lang.RuntimeException class.

3) try
```
{
}catch(IOException | InterruptedException e)
{
}
```

The above catch block handles IOException, InterruptedException, sub classes of IOException & sub classes of InterruptedException.

This concept is called as handling multiple exceptions with single catch blocks.

This concept is introduced in JDK 1.7 version in 2011.

throw keyword:

It is used to pass an object of exception class to a catch block.

throws keyword:

It is used to apply an exception to a method and it is also used to handle exception.

finally block:

It is used to perform cleanup activities.

Cleanup activities are closing a file, closing a database connection, closing a socket, .. etc.,

finally block is executed even exception occurs in a program.

By

Mr. Venatesh Mansani

Naresh i Technologies

JAVA STREAMS & SERIALIZATION

Streams:

A stream is a flow of data from source to destination. A source can be a keyboard, file, client, server, ... etc., A destination can be a monitor, file, client, server, .. etc.,

In Java, streams are divided into 3 categories:

1) Console Input/Output Streams

2) File Input/Output Streams

3) Network Input/Output Streams

Predefined Streams:

There are 3 predefined streams

1) in 2) out 3) err

"in" is an object reference of java.io. InputStream class

"out" & "err" are object reference of java.io.PrintStream class.

The above all predefined streams are static members of java.lang.System class.

Differences between System.out & System.err

System.out	System.err
1) It is used to display output messages.	1) It is used to display error messages.
2) This stream data can be redirected to a file.	2) This stream data cannot be redirected to a file.

Example:

```
class Demo
{
        public static void main(String args[])
```

```
    {
        System.out.println("Core Java");
        System.err.println("Advanced Java");
    }
}
```

C:\> javac Demo.java

C:\> java Demo

Output:

Core Java

Advanced Java

C:\>java Demo>a.txt

Output:

Advanced Java

C:\>start notepad a.txt

Core Java appears in a a.txt file because Core Java written with System.out

This stream data can be redirected to a file.

File Streams:

1) FileInputStream

2) FileOutputStream

FileInputStream used to read data from file.

FileOutputStream used to write data to a file.

Program to read data from file:

```
class ReadDemo
{
```

```
    public static void main(String args[])
    {
        try{
        FileInputStream fis=new FileInputStream(args[0]);
        int n=fis.available();
        byte[] b=new byte[n];
        fis.read(b);
        String s =new String(b);
        System.out.println(s);
        }catch(Exception e)
        {
            System.err.println(e);
        }
    }
}
```

Program to copy data from one file to another file:

```
class CopyDemo
{
    public static void main(String args[])
    {
        try{
        FileInputStream fis=new FileInputStream(args[0]);
        int n=fis.available();
        byte[] b=new byte[n];
```

```
            fis.read(b);
            FileOutputStream fos=new FileOutputStream(args[1]);
            fout.write(b);
            }catch(Exception e)
            {
                System.err.println(e);
            }
        }
}
```

There are two types of streams:

1) Byte streams

2) Character streams

Byte streams handle all types of data where as character streams handle text only.

List of byte stream classes:

1) InputStream

2) PrintStream

3) FileInputStream

4) FileOutputStream

5) ObjectInputStream

6) ObjectOutputStream

7) DataInputStream

8) DataOutputStream

DataInputStream & DataOutputStream classes support all primitive data types & strings.

ObjectInputStream & ObjectOutputStream classes support objects

List of Character streams:

1) FileReader

2) FileWriter

3) BufferedReader

4) BufferedWriter

Finally block:

It is used to perform cleanup activities. Clean up activities are closing a file, closing a database connection, closing a socket, ..etc.,

Finally block is executed even exception occurs in a program.

Example:

```
class ReadDemo
{
	public static void main(String args[])
	{
		FileInputStream fis=null;
		try{
		fis=new FileInputStream(args[0]);
		int n=fis.available();
		byte[] b=new byte[n];
		fis.read(b);
		String s =new String(b);
		System.out.println(s);
		}catch(Exception e)
```

```
        {
            System.err.println(e);
        }
        finally
        {
                try{
                fis.close();
                }catch(Exception e)
                {
                    System.err.println(e);
                }
        }
    }
}
```

Object Streams:

1) ObjectInputStream: It is used to read an object from file

2) ObjectOutputStream: It is used to write an object to a file

Serialization:

It is a process of converting object into a series of bits.

In Java, object must be serializable to do the following operations:

1) Writing object to a file

2) Reading object from file

3) Writing object to a network

4) Reading object from network

The class must implements java.io.Serializable interface to make serializable object

java.io.Serializable interface is a marker interface, tag interface or empty interface because no members in this interface.

Example:

```
import java.io.*;
class Emp implements Serializable
{
	transient int empNo=101;
	float salary=5000.00f;
}
class Demo
{
	public static void main(String args[])
	{
		try{
		Emp e1=new Emp();
		FileOutputStream fos=new FileOutputStream("emp.txt");
		ObjectOutputStream oos=new ObjectOutputStream(fos);
		oos.writeObject(e1);
		oos.close();
		fos.close();
		FileInputStream fis=new FileInputStream("emp.txt");
		ObjectInputStream ois=new ObjectInputStream(fis);
		Emp e2=(Emp)ois.readObject();
```

```
            System.out.println(e2.empNo+"\t"+e2.salary);
            ois.close();
            fis.close();
            }catch(Exception e)
            {
                  System.err.println(e);
            }
      }
}
```

transient keyword:

It is used to prevent serialization. It is used in real time applications with passwords, PIN numbers, security code, ... etc.,

By

Mr. Venatesh Mansani

Naresh i Technologies

NETWORK PROGRAMMING

Network:

A set of co-operative interconnected computers.

Co-operation is possible with network software and interconnection is possible with network hardware.

Socket: It is a connection end at client side.

ServerSocket: It is a connection end at server side.

Port Number:

It is used to identify the service. Port numbers range is 0 to 65535.

Reserved port numbers are 0 to 1023 and free port numbers are 1024 to 65535.

Steps to develop client application:

1) Create a Socket object with server address and port number.
2) Create an OutputStream that can be used to send information to the server.
3) Create an InputStream that can be used to receive information from the server.
4) Do Input/Output Operations.
5) Close the Socket

Steps to develop server application:

1) Create a ServerSocket object with port number.
2) Call accept() method to receive requests from the client.
3) Create an InputStream that can be used to receive information from the client.
4) Create an OutputStream that can be used to send information to the client.
5) Do Input/Output operations.
6) Close the ServerSocket.

Client Program:

```
import java.io.*;
import java.net.*;
class Client
{
	public static void main(String args[])
	{
	  try{
	  String address=args[0];
	  int port=Integer.parseInt(args[1]);
	  Socket s=new Socket(address, port);
	  OutputStream os=s.getOutputStream();
	  InputStream is=s.getInputStream();
	  byte[] b1=new byte[100];
	  byte[] b2=new byte[100];
	  while(true)
	  {
		System.out.print("To Server: ");
		System.in.read(b1);
		os.write(b1);
		is.read(b2);
		String s1=new String(b2);
		String s2=s1.trim();
		System.out.println("From Server: "+s2);
```

```
        }
        }catch(Exception e)
        {
                System.err.println(e);
        }
      }
}
```

Server Program:

```
import java.io.*;
import java.net.*;
class Server
{
      public static void main(String args[])
      {
        try{
        int port=Integer.parseInt(args[0]);
        ServerSocket ss=new ServerSocket(port);
        Socket s=ss.accept();
        InputStream is=s.getInputStream();
        OutputStream os=s.getOutputStream();
        byte[] b1=new byte[100];
        byte[] b2=new byte[100];
        while(true)
        {
```

```
                is.read(b1);
                String s1=new String(b1);
                String s2=s1.trim();
                System.out.println("From Client: "+s2);
                System.out.print("To Client: ");
                System.in.read(b2);
                os.write(b2);
        }
        }catch(Exception e)
        {
                System.err.println(e);
        }
      }
}
```

Execution at Client side:

C:\>java Client localhost 1024

Execution at Server side:

C:\>java Server 1024

By

Mr. Venkatesh Mansani

Naresh i Technologies

COLLECTIONS FRAMEWORK & GENERICS

Collections Framework:

A set of collection classes and interfaces is called as collections framework.

(or)

A set of data structures related classes and interfaces is called as collections framework.

Collection:

A collection is an object that represents group of objects.

Data Structures:

Arranging data in different formats is called data structures.

Advantages of Collections Framework:

1) Reduces programming effort

2) Increases programming speed & quality

3) Allows interoperability among unrelated APIs.

Collection classes & interfaces are the part of java.util package.

java.util package classes & interfaces are divided into two categories:

1) Collections Framework Collections

2) Legacy Collections

1) Collections Framework Collections:

JDK 1.2 & above versions collection classes & interfaces are called Collections framework collections.

2) Legacy Collections:

JDK 1.0 & 1.1 versions collection classes & interfaces are called legacy collections.

Collections Framework Collections are divided into 3 sub categories:

1) Core Collection Interfaces

2) General Purpose Implementations

3) More Utility Collections

1) Core Collection Interfaces:

These interfaces are the foundation of collections framework.

1) Collection

2) List

3) Set

4) Map

5) SortedSet

6) SortedMap

7) NavigableSet

8) NavigableMap

9) Queue

10) Deque

1) Collection interface:

It is a root interface in a one dimensional collections hierarchy.

2) List interface:

It extends Collection interface and it maintains sequences. It allows duplicate elements.

3) Set interface:

It extends Collection interface and it maintains sets. It does not allow duplicate elements.

Differences between List & Set:

List	Set
1) It maintains sequences.	1) It maintains sets.
2) It allows duplicates.	2) It does not allow duplicates.

4) Map interface:

It is a root interface in a two dimensional collections hierarchy. It maintains data as a key/value pairs. It does not allow duplicate keys(Values may be duplicated).

Differences between Set & Map

Set	Map
1) It is a one dimensional collection interface	1) It is a two dimensional collection interface.
2) It contains elements.	2) It contains key/value pairs.
3) It does not allow duplicates	3) It does not allow duplicate keys. (Values may be duplicated).
4) It is an index based Collection.	4) It is a key based collection.

5) SortedSet interface:

A SortedSet is a Set in which elements are sorted. It extends Set interface.

6) SortedMap interface:

A SortedMap is a Map in which key/value pairs are sorted based on keys. It extends Map interface.

7) NavigableSet interface:

It is used to navigate elements of a Set. It extends SortedSet interface.

8) NavigableMap interface:

It is used to navigate elements of a Map. It extends SortedMap interface.

9) Queue interface:

It is called as First In First Out(FIFO) list. It allows insertion at rear end and deletion at front end.

10) Deque interface:

It is called as Double Ended QUEue data structure. It allows both insertion & deletion at both the ends(front end & rear end).

Differences between Queue & Deque:

Queue	Deque
1) It is a queue data structure	1) It is a double ended queue data structure
2) It is called as First In First First Out List	2) It is not called as First In Out List
3) It allows insertion at rear both the ends.	3) It allows insertion at end only.
4) It allows deletion at front end only.	4) It allows deletion at both the ends.

2) General Purpose Implementations:

The core collection interfaces implementation classes are called general purpose implementations.

1) ArrayList

2) LinkedList

3) HashSet

4) LinkedHashSet

5) TreeSet

6) HashMap

7) LinkedHashMap

8) TreeMap

9) PriorityQueue

10) ArrayDeque

1) ArrayList class:

It is an array representation of list implementation class.

It allows duplicate elements because it implements List interface.

It is an implementation linear list data structure.

It supports all the operations of linear list data structure.

It supports all types of data.

It allows null values also

Generics:

Generics allows to write type safe programs.

Generics are introduced in JDK 1.5 version in 2004.

Advantages of generics:

1) Allows to write type safe programs

2) It does not require type casting

The syntax to create an object to generic class:

```
ClassName<ReferenceDataType> ObjectReference =
                        new Constructor<ReferenceDataType>();
```

Examples:

1) ArrayList<Integer> al1=new ArrayList<Integer>();

The above ArrayList object is type safe because it allows only integer type elements.

2) ArrayList<String> al2=new ArrayList<String>();

The above ArrayList object is type safe because it allows only string type elements.

3) ArrayList<Float> al3=new ArrayList<Float>();

The above ArrayList object is type safe because it allows only float type elements.

Generic Type Inference:

This feature allows to create an object to generic class in a new way. This feature introduced in JDK 1.7 verion in 2011.

Example:

ArrayList<Integer> al=new ArrayList<Integer>();

The above code can be written from JDK 1.7 onwards as follows:

ArrayList<Integer> al=new ArrayList<>();

2) LinkedList class:

It is linked representation of list implementation class.

It allows duplicate elements because it implements List interface.

It is an implementation double linked list data structure.

It supports all the operations of double linked list data structure.

It supports all types of data.

It allows null values also

Differences between ArrayList & LinkedList

ArrayList	LinkedList
1) It is an array representation implementation class	1) It is a linked representation of list of list implementation class
2) It is a linear list data structure.	2) It is a double linked list data structure
3) It occupies less memory	3) It occupies more memory because data stored in nodes.
4) In ArrayList, insertion & deletion operations require shuffling of data.	4) In LinkedList, it does not require shuffling of data.

3) HashSet class:

It is an implementation of hashing technique with array representation.

Hashing is a technique, in which insertion, deletion & find operation in a constant average time.

It does not allow duplicate elements because it implements Set interface.

It supports all types of data.

It allows null values also.

It is an unordered set.

4) LinkedHashSet class:

It is an implementation of hashing technique with linked representation.

It does not allow duplicate elements because it implements Set interface.

It supports all types of data.

It allows null values also.

It is an ordered set.

5) TreeSet class:

It is an implementation of binary search technique with linked representation.

A binary tree is said to be binary search tree if it is follows the following rules.

1) If the element is less than root element, then it must be left sub tree.

2) If the element is greater than root element, then it must be right sub tree.

It does not allow duplicate elements because it implements Set interface.

It supports all types of data.

It does not allow null values.

It is a sorted set.

6) HashMap class:

It is an implementation of hashing technique with array representation.

It is a two dimensional collection class and it maintains data as a key/value pairs because it implements Map interface

It does not allow duplicate keys(Values may be duplicated)

It supports all types of keys and all types of values.

It allows null keys & null values.

It is an unordered map.

7) LinkedHashMap class:

It is an implementation of hashing technique with linked representation.

It is a two dimensional collection class and it maintains data as a key/value pairs because it implements Map interface

It does not allow duplicate keys(Values may be duplicated)

It supports all types of keys and all types of values.

It allows null keys & null values. It is an ordered map.

8) TreeMap class:

It is an implementation of binary search technique with linked representation.

It is a two dimensional collection class and it maintains data as a key/value pairs because it implements Map interface

It does not allow duplicate keys(Values may be duplicated)

It supports all types of keys and all types of values.

It does not allow null keys(null values allowed)

It is a sorted map.

Differences between Set and Map implementation classes:

HashSet	HashMap
LinkedHashSet	LinkedHashMap
TreeSet	TreeMap

1) These are one dimensional collections.	1) These are two dimensional collections.
2) These collections contain elements.	2) These collections contain Key/value pairs.
3) These collections do not allow duplicate elements.	3) These collections do not allow duplicate keys.
4) These are index based collections.	4) These are key based collections.

9) PriorityQueue class:

It is an array representation of queue implementation class.

It allows insertion at rear end and deletion at front end only.

It allows duplicate elements.

It supports all types of data.

It does not allow null values.

10) ArrayDeque class:

It is an array representation of Deque implementation class.

It allows both insertion & deletion at both the ends(front end & rear end)

It allows duplicate elements.

It supports all types of data.

It does not allow null values.

More Utility Collections:

1) Iterator (interface)

2) ListIterator (interface)

3) Arrays (class)

4) Collections (class)

5) Scanner (class)

Iterator interface:

It is used to iterate elements of a collection.

ListIterator interface:

It is also used to iterate elements of a collection.

Differences between Iterator & ListIterator

Iterator	ListIterator
1) It is used to iterate elements of any collection.	1) It is used to iterate elements of ArrayList & LinkedList only.

2) It supports only forward direction to iterate elements.	2) It supports both forward and backward directions to iterate elements.
3) It allows only remove operation while iterating elements.	3) It allows add, remove & set operations while iterating elements.

Iterator Example:

```
import java.util.*;
class Demo
{
    public static void main(String args[])
    {
        ArrayList<Integer> al=new ArrayList<>();
        al.add(83);
        al.add(38);
        al.add(81);
        al.add(78);
        al.add(73);
        System.out.println(al);
        Iterator<Integer> i=al.iterator();
        while(i.hasNext())
        {
            int x=i.next();
            System.out.println(x+5);
        }
```

```
    }
}
```

Scanner:

It is used accept the data from keyboard

Example:

```
import java.util.*;
class Demo
{
    public static void main(String args[])
    {
        System.out.print("Enter any number: ");
        Scanner s=new Scanner(System.in);
        int x=s.nextInt();
        System.out.println(x);
    }
}
```

Legacy Collections:

JDK 1.0 & 1.1 versions collection classes & interfaces are called legacy collections.

1) Enumeration(interface)
2) Vector(class)
3) StringTokenizer(class)
4) Hashtable(class)
5) Random(class)
6) Stack(class)

7) Date(class)

1) Enumeration interface:

It is used to iterate elements of a collection. It is similar to Iterator interface.

Differences between Enumeration & Iterator:

Enumeration	Iterator
1) It is a legacy collection interface.	1) It is a collections framework interface.
2) It does not allow other operations while iterating elements.	2) It allows remove operations while iterating elements.

2) Vector class:

It is an implementation of linear list data structure. It is similar to ArrayList class.

Differences between Vector & ArrayList

Vector	ArrayList
1) It is a legacy collection class.	1) It is a collections framework class.
2) Methods of Vector class are not synchronized.	2) Methods of ArrayList class are synchronized.

3) StringTokenizer class:

It allows an application to break a string into tokens.

Example:

"Welcome to Sathya Technologies" is a one string and it has 4 tokens(words).

4) Hashtable class:

It is a two dimensional collection class and it maintains data as a key/value pairs. It is an implementation of hashing technique with array representation. It is similar to HashMap class.

Differences between Hashtable & HashMap

Hashtable	HashMap
1) It is a legacy collection class.	1) It is a collections framework class.
2) Methods of Hashtable are synchronized.	2) Methods of HashMap are not synchronized.
3) Hashtable does not allow null keys & null values.	3) HashMap allows one null key and many null values.

5) Random class:

It is used to get random integers, floating point numbers & boolean values.

Example:

```
import java.util.*;
class Demo
{
    public static void main(String args[])
    {
        Random r=new Random();
        for(int i=1;i<=10;i++)
        {
            System.out.println(r.nextInt(1000));
```

```
        }
    }
}
```

6) Stack class:

It is called as Last In First Out (LIFO) list.

Example:

```
import java.util.*;
class Demo
{
    public static void main(String args[])
    {
        Stack<Integer> s=new Stack<>();
        s.push(10);
        s.push(83);
        s.push(75);
        s.push(87);
        s.push(73);
        System.out.println(s);
        System.out.println(s.pop());
        System.out.println(s);
    }
}
```

Differences between Stack & Queue:

Stack	Queue
1) It is called as Last In First Out(FIFO) list.	1) It is called as First In First Out(LIFO) list.
2) It is a Stack data structure.	2) It is a Queue data structure.
3) It allows both insertion and deletion at top end only.	3) It allows insertion at rear end and deletion at front end.
4) It is a legacy collection interface.	4) It is a collections framework class.

7) Date class:

It is used to get the system date & time.

Example:

```
import java.util.*;
class Demo
{
	public static void main(String args[])
	{
		Date d=new Date();
		int x=d.getHours();
		int y=d.getMinutes();
		int z=d.getSeconds();
		System.out.println(x+":"+y+":"+z);
	}
}
```

COLLECTIONS FRAMEWORK

ArrayList Collection with Emp Objects Example:

```
class Emp
{
	int empNo;
	String name;
	float salary;
	Emp(int empNo, String name, float salary)
	{
		this.empNo=empNo;
		this.name=name;
		this.salary=salary;
	}
	public String toString()
	{
		return empNo+"\t"+name+"\t"+salary;
	}
}
class Demo
{
	public static void main(String... args)
	{
		Emp e1=new Emp(101, "bbb", 5000.00f);
```

```
        Emp e2=new Emp(104, "aaa", 5500.00f);
        Emp e3=new Emp(102, "eee", 6000.00f);
        Emp e4=new Emp(103, "ccc", 4500.00f);
        Emp e5=new Emp(105, "ddd", 6500.00f);
        ArrayList<Emp> al=new ArrayList<>();
        al.add(e1);
        al.add(e2);
        al.add(e3);
        al.add(e4);
        al.add(e5);
        System.out.println(al);
    }
}
```

Comparator Vs Comparable

Comparator	Comparable
1) It is in java.util package	1) It is in java.lang package
2) It contains the following method to compare elements. int compare(T, T)	2) It contains the following methods to compare elements. int compareTo(T)
3) The comparison logic can be in any other class also.	3) The comparison logic must be present in the same class.

Example:

```
import java.util.*;


class Emp
```

```
{
        int empNo;
        String name;
        float salary;
        Emp(int empNo, String name, float salary)
        {
                this.empNo=empNo;
                this.name=name;
                this.salary=salary;
        }
        public String toString()
        {
                return empNo+"\t"+name+"\t"+salary;
        }
}
class EmpNoComparator implements Comparator<Emp>
{
        public int compare(Emp e1, Emp e2)
        {
                if(e1.empNo>e2.empNo)
                    return 1;
                else if(e1.empNo<e2.empNo)
                    return -1;
                else
```

```
                return 0;
        }
}
class NameComparator implements Comparator<Emp>
{
        public int compare(Emp e1, Emp e2)
        {
                return e1.name.compareTo(e2.name);
        }
}
class SalaryComparator implements Comparator<Emp>
{
        public int compare(Emp e1, Emp e2)
        {
                if(e1.salary>e2.salary)
                    return 1;
                else if(e1.salary<e2.salary)
                    return -1;
                else
                    return 0;
        }
}
class Demo
{
```

```
	public static void main(String... args)
	{
		Emp e1=new Emp(101, "bbb", 5000.00f);
		Emp e2=new Emp(104, "aaa", 5500.00f);
		Emp e3=new Emp(102, "eee", 6000.00f);
		Emp e4=new Emp(103, "ccc", 4500.00f);
		Emp e5=new Emp(105, "ddd", 6500.00f);
		ArrayList<Emp> al=new ArrayList<>();
		al.add(e1);
		al.add(e2);
		al.add(e3);
		al.add(e4);
		al.add(e5);
		System.out.println(al);
		Collections.sort(al, new EmpNoComparator());
		System.out.println(al);
		Collections.sort(al, new NameComparator());
		System.out.println(al);
		Collections.sort(al, new SalaryComparator());
		System.out.println(al);
	}
}
```


Mr. Venkatesh Mansani

MULTITHREADING & SYNCHRONIZATION

Multithreading:

Execution of more than one thread at a time is called as multithreading.

Thread:

A thread is a piece of code that executes independently.

Every program contains at least one thread. i.e. main thread

Main thread default name is main only.

Main thread default priority is normal priority(Priority number is 5)

There are two ways to create a new thread:

1) By extending java.lang.Thread class
2) By implementing a java.lang.Runnable interface

Life cycle of a thread (or) thread states:

Whenever thread class constructor is called new thread will born.

Thread comes to a ready state whenever start() method is called.

Thread will run whenever run() method is called.

Thread comes to a sleeping state whenever sleep() method is called.

Sleeping thread will wake up automatically whenever time interval is finished.

Thread is suspended whenever suspend() method is called.

Suspended thread will run whenever resume() method is called.

Thread comes to a waiting state whenever wait() method is called.

Waiting thread will run whenever notify() method is called.

Thread will die whenever destroy() method is called.

Program to get current thread information:

```
class Demo
{
	public static void main(String args[])
	{
		Thread t=Thread.currentThread();
		System.out.println(t.getName());
		System.out.println(t.getPriority());
		t.setName("demo");
		t.setPriority(Thread.MAX_PRIORITY);
		System.out.println(t.getName());
		System.out.println(t.getPriority());
	 }
}
```

Program to demonstrate sleep() method:

```
class Demo
{
	Public static void main(String args[])
	{
		for(int i=1;i<=10;i++)
		{
			System.out.println(i);
			try{
			Thread.sleep(1000);
```

```
                    }catch(Exception e)
                    {
                        System.err.println(e);
                    }
                }
        }
}
```

Steps to develop multithreaded application by extending java.lang.Thread class:

1) Create a class that extends java.lang.Thread class

2) Override run() method.

Note: run() method given as a null body method to write child thread task code.

3) Write child thread task code in a run() method.

4) Write main() method

5) Create an object of current class.

6) Call start() method.

Note: start() method implicitly calls run() method.

7) Write main thread task code in a main() method.

Example:

```
class Demo extends Thread
{
    public void run()
    {
        try{
```

```
        for(int i=1;i<=10;i++)
        {
            System.out.println("Child Thread: "+i);
            Thread.sleep(1000);
        }
        }catch(Exception e)
        {
            System.err.println(e);
        }
    }
    public static void main(String args[])
    {
        try{
        Demo d=new Demo();
        d.start();
        for(int i=1;i<=10;i++)
        {
            System.out.println("Main Thread: "+i);
            Thread.sleep(1000);
            if(i==5)
              d.suspend();
            if(i==10)
              d.resume();
        }
```

```
        }catch(Exception e)
        {
            System.err.println(e);
        }
    }
}
```

Steps to create multithread application by implementing java.lang.Runnable interface:

1) Create a class that implements java.lang.Runnable interface.

2) Override run() method.

3) Write child thread task code in a run() method.

4) Write main() method

5) Create an object of current class & assign to Runnable reference

6) Create an object Thread class by passing Runnable reference

7) Call start() method.

8) Write main thread task code in a main() method.

Example:

```
class Test implements Runnable
{
    public void run()
    {
        try{
        for(int i=1;i<=10;i++)
        {
```

```
                System.out.println("JavaEE");
                Thread.sleep(1000);
            }
            }catch(Exception e)
            {
                System.err.println(e);
            }
        }
}
class Demo implements Runnable
{
        public void run()
        {
            try{
            for(int i=1;i<=10;i++)
            {
                System.out.println("Core Java");
                Thread.sleep(2000);
            }
            }catch(Exception e)
            {
                System.err.println(e);
            }
        }
```

```
    public static void main(String args[])
    {
        try{
        Runnable r=new Demo();
        Thread t=new Thread(r);
        t.start();
        Runnable r2=new Test();
        Thread t2=new Thread(r2);
        t2.start();
        for(int i=1;i<=10;i++)
        {
            System.out.println("Advanced Java");
            Thread.sleep(3000);
        }
        }catch(Exception e)
        {
            System.err.println(e);
        }
    }
}
```

Synchronization:

It is a mechanism that allows to access a shared resource only one thread at a time.

There are two ways to synchronize the code:

1) synchronizing a method

2) synchronizing block of code

1) synchronizing a method:

Syntax:

```
synchronized ReturnType MethodName(arg1, arg2, ......)
{
    ================
    ================
    ================
}
```

2) synchronizing a block of code:

Syntax:

```
ReturnType MethodName(arg1, arg2, ......)
{
    ================
    ================
    synchronized(Object)
    {
        ================
        ================
    }
    ================
    ================
}
```

Example:

```
class Bank
{
	float balance=5000.00f;
	synchronized void withdraw(float amount)
	{
		try{
		System.out.println("Withdraw Started");
		if(balance<amount)
		{
			System.out.println("Less Balance, Waiting for Deposit");
			wait();
		}
		balance=balance-amount;
		System.out.println("Withdraw Completed");
		}catch(Exception e)
		{
			System.err.println(e);
		}
	}
	synchronized void deposit(float amount)
	{
		System.out.println("Deposit Started");
		balance=balance+amount;
```

```
                System.out.println("Deposit Completed");
                notify();
        }
}
class Customer1 extends Thread
{
        Bank b;
        Customer1(Bank b)
        {
                this.b=b;
        }
        public void run()
        {
                b.withdraw(8000.00f);
        }
}
class Customer2 extends Thread
{
        Bank b;
        Customer2(Bank b)
        {
                this.b=b;
        }
        public void run()
```

```
        {
                b.deposit(5000.00f);
        }
}
class Demo
{
        public static void main(String args[])
        {
                Bank b=new Bank();
                Customer1 c1=new Customer1(b);
                c1.start();
                Customer2 c2=new Customer2(b);
                c2.start();
        }
}
```

NEW FEATURES

"strictfp" keyword:

strictfp is a keyword and it is used to restrict floating-point calculations and ensuring same result on every platform while performing operations.

strictfp modifier is used with classes, interfaces and methods only.

strictfp modifier can be used with main() method also.

When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface are implicitly strictfp.

strictfp cannot be used with abstract methods, variables & constructors.

It can be used with abstract classes/interfaces.

Example:

```
strictfp class Demo
{
	public static void main(String[] args)
	{
		float a=3.24f;
		float b=2.6654f;
		System.out.println(a/b);
	}
}
```

Assertions:

An assertion is a condition that must be true during the program execution.

It is mainly used for testing purpose.

It provides an effective way to detect and correct programming errors.

There are two ways to use assertions.

1) assert expression;

2) assert expression1 : expression2;

Examples:

1) assert((x>0)&&(x<=10));

2) assert((x>0)&&(x<=10)) : "Invalid";

Example:

```
import java.util.*;
class Demo
{
	public static void main(String[] args)
	{
		Scanner s=new Scanner(System.in);
		System.out.print("Enter any number between 1 and 10: ");
		int x=s.nextInt();
		assert((x>0)&&(x<=10));
		System.out.println(x);
	}
}
```

VarArgs:

It means Variable Arguments.

It allows to pass 0 to any number of arguments to a method.

There can be only one variable argument in the method.

Variable argument (varargs) must be the last argument.

Example:

```
class Demo
{
	void display(int... x)
	{
		for(int y : x)
		{
			System.out.println(y);
		}
	}
	public static void main(String[] args)
	{
		Demo d=new Demo();
		d.display(34, 23, 13, 45);
		d.display(1223, 8881, 8234, 8183, 1234);
	}
}
```

Enumerations("enum" keyword):

It allows us to create a new data type in Java.

The Enum in Java is a data type which contains a fixed set of constants.

The Java enum constants are static and final implicitly.

we can define an enum either inside the class or outside the class.

We can have fields, constructors, methods, and main methods in Java enum.

Note: Java compiler internally adds values(), valueOf() and ordinal() methods

within the enum at compile time.

Example1:

```
enum Day
{
	Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
class Demo
{
	public static void main(String args[])
	{
		Day d=Day.Mon;
		System.out.println(d);
	}
}
```

Example2:

```
enum Day
{
	Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
class Demo
{
	public static void main(String args[])
	{
		for(Day d : Day.values())
```

```
            {
                System.out.println(d);
            }
        }
}
```

Example3:

```
class Demo
{
    enum Day{Mon, Tue, Wed, Thu, Fri, Sat, Sun}
    public static void main(String args[])
    {
        Day d1=Day.valueOf("Sat");
        Day d2=Day.Sun;
        System.out.println(d1.ordinal());
        System.out.println(d2.ordinal());
    }
}
```

Annotations:

Annotations are used pass some additional information to compiler about methods, classes, interfaces, .. etc.,

All annotations are begins with @ symbol.

@Override annotation assures that the subclass method is overriding the super class method. If it is not compile time error occurs.

Example:

```
class Test
```

```
{
	void show()
	{
		System.out.println("Test class");
	}
}
class Demo extends Test
{
	@Override
	void show()
	{
		System.out.println("Demo class");
	}
	public static void main(String[] args)
	{
		Test t=new Demo();
		t.show();
	}
}
```

Static Imports:

This feature allows to avoid class name with static members.

Syntax:

import static package-name.sub-package-name.ClassName.*;

Examples:

import static java.lang.Integer.*;

This statement allows to access all static members of Integer class without class name.

import static java.lang.System.*;

This statement allows to access all static members of System class without class name.

Example:

```
import static java.lang.Integer.*;
import static java.lang.System.*;
class Demo
{
	public static void main(String[] args)
	{
		int x=parseInt(args[0]);
		int y=parseInt(args[1]);
		out.println(x+y);
	}
}
```

Underscores in numeric literals

By

Mr. Venkatesh Mansani

Naresh i Technologies

NEW FEATURES PART-2

Functional Interface:

A functional interface is an interface that contains only one abstract method.

Functional interface can have default methods & static methods.

Functional interface can have variables also.

@FunctionalInterface Annotation:

It ensures that the functional interface cannot have more than one abstract method.

In case more than one abstract method is present, then the compiler shows unexpected @FunctionalInterface annotation message.

It is not mandatory to use this annotation.

Example:

```
@FunctionalInterface
interface Test
{
        void show();
}
class Demo
{
       public static void main(String args[])
       {
        Test t=new Test()
        {
              public void show()
```

```
            {
              System.out.println("Welcome");
            }
        };
        t.show();
      }
}
```

Lambda Expressions:

This feature allows us to write anonymous method. Anonymous method means a method which has no name in the implementation. It is used to provide concise code in the implementation of anonymous class.

Lambda expression is an important feature of Java 8. It provides a concise way to represent method as an expression.

The Lambda expression is used to provide the implementation of functional interface.

Example1:

```
@FunctionalInterface
interface Test
{
      void show();
}
class Demo
{
      public static void main(String args[])
      {
```

```
        Test t=()->System.out.println("Welcome");
        t.show();
       }
}
```

Example2:

```
@FunctionalInterface
interface Test
{
        void add(int a, int b);
}
class Demo
{
       public static void main(String args[])
       {
        Test t=(a, b)->System.out.println(a+b);
        t.add(5,3);
       }
}
```

Example3:

```
@FunctionalInterface
interface Test
{
        int add(int a, int b);
}
```

```
class Demo
{
	public static void main(String args[])
	{
	 Test t=(a, b)->a+b;
	 int x=t.add(5,3);
	 System.out.println(x);
	}
}
```

Example4:

```
class Demo
{
	public static void main(String args[])
	{
	 Runnable r=()->
	 {
		try{
		for(int i=1;i<=10;i++)
		{
		 System.out.println("Child Thread: "+i);
		  Thread.sleep(1000);
		    }
		}catch(Exception e)
		{
```

```
                System.err.println(e);
            }
        };
        Thread t=new Thread(r);
        t.start();
        try{
        for(int i=1;i<=10;i++)
        {
                System.out.println("Main Thread: "+i);
                Thread.sleep(1000);
            }
        }catch(Exception e)
        {
                System.err.println(e);
        }
       }
}
```

Predefined Functional Interfaces:

java.util.function package

1) Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface

public interface Function<T,R>

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is apply(Object).

Example:

```
import java.util.function.*;
class Demo
{
	/*int cube(int a)
	{
	 return a*a*a;
	}*/
	public static void main(String args[])
	{
	 /*Demo d=new Demo();
	 System.out.println(d.cube(5));*/
	 Function<Integer, Integer> f=a->a*a*a;
	 System.out.println(f.apply(3));
	}
}
```

2) Interface Predicate<T>

Type Parameters:

T - the type of the input to the predicate

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface

public interface Predicate<T>

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is test(Object).

Example:

```
import java.util.function.*;
class Demo
{
	/*boolean isEven(int a)
	{
	 if(a%2==0)
	   return true;
	 else
	   return false;
	}*/
	public static void main(String args[])
	{
	 /*Demo d=new Demo();
	 System.out.println(d.isEven(5));*/
	 Predicate<Integer> p=a->a%2==0;
	 System.out.println(p.test(4));
```

```
    }
}
```

Java Method References:

Method reference is used to refer method of functional interface. It is also one more way and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

Types of Method References:

There are following types of method references in java:

1) Reference to a static method.
2) Reference to an instance method.
3) Reference to a constructor.

Reference to a static method:

Syntax:

```
ContainingClass::staticMethodName
```

Reference to an instance method of a particular object:

Syntax:

```
containingObject::instanceMethodName
```

Reference to a constructor:

Syntax:

```
ClassName::new
```

Static Method Reference Example:

```
interface Test
{
    void show();
```

```
}
class Demo
{
	static void print()
	{
		System.out.println("Welcome");
	}
	public static void main(String args[])
	{
		Test t=Demo::print;
		t.show();
	}
}
```

Instance Method Reference Example:

```
interface Test
{
	void show();
}
class Demo
{
	void print()
	{
		System.out.println("Welcome");
	}
```

```
    public static void main(String args[])
    {
        Demo d=new Demo();
        Test t=d::print;
        t.show();
    }
}
```

Constructor Reference Example:

```
interface Test
{
    void show();
}
class Demo
{
    Demo()
    {
        System.out.println("Welcome");
    }
    public static void main(String args[])
    {
        Test t=Demo::new;
        t.show();
    }
}
```

Runnable interface run() method reference example:

```
class Test
{
	static void show()
	{
		try{
		for(int i=1;i<=10;i++)
		{
		  System.out.println("Child Thread: "+i);
		  Thread.sleep(1000);
		}
		}catch(Exception e)
		{
		  System.err.println(e);
		}
	}
}
class Demo
{
	public static void main(String args[])
	{
		Runnable r1=Test::show;
		Thread t1=new Thread(r1);
		t1.start();
```

```
            try{
            for(int i=1;i<=10;i++)
            {
              System.out.println("Main Thread: "+i);
              Thread.sleep(1000);
            }
            }catch(Exception e)
            {
              System.err.println(e);
            }
        }
}
```

By

Mr. Venkatesh Mansani

Naresh i Technologies