jain computer classes

Java Solutions

Code in java

Shishav jain
3/10/2014

Indexing

Chapter 1 .

Chapter 2 : Control Statements , Objects and classes

Chapter 3 : Packages and Interfaces , String handling

Chapter 4 : Exception Handling and File Handling

4.1 Exceptions
4.2 Exception Hierarchy
4.3 Exception Handling
4.4 Try, catch and finally
4.5 Throw Statements
4.6 throws Clause
- ❖ **Difference between final, finally and finalize**
- ❖ **Difference between throw and throws**

4.7 Java I/O
4.8 Java file handling

Chapter 5 : Concurrency and Applet

5.1 Multitasking (process and thread)
5.2 Threads
5.3 Creating Threads
5.4 Synchronization
5.5 Thread States (Thread life cycle).
5.6 Thread Priorities
5.7 Applets
5.8 Difference between remote and local applet
5.9 Applet Life Cycle

Chapter 6 : Java and Database

6.1 Drivers
6.2 Steps for Connectivity between java program and database
6.3 Select Query Program
6.4 Insert Query Program
6.5 Delete Query Program
6.6 Update Query Program

Chapter 1: language fundamental and operators

Java is product of Oracle Company, but initially it is developed by sun micro system. Java is conceived by a team headed by James Gosling. Initially it is called Ook, but after renamed to java. Java follow the coding procedure of object oriented programming, so it is called object oriented language. In Object Oriented Programming, emphasis gives to object rather than the procedure. There are four pillars of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

a. **Encapsulation**: Binding of data and the methods(functions) which works on that data is called Encapsulation. Encapsulation also decides which members of this binding is accessible from outside of it or which are not. Using Class concept with visibility controls we can achieve encapsulation in object oriented programming.
b. **Abstraction**: Hiding the implementation to the outside world is called Abstraction. Abstraction makes our work easier to only concentrate on the required features without knowing the background implementation of it.
c. **Inheritance**: Inheritance is a concept which supports reusability of components in object oriented programming. By Inheritance we can get already compiled code reused without recompiling it.
d. **Polymorphism**: Polymorphism is a concept through which we can create more than one forms of functions which have same name. Using same name, programmer can create more than one functionality under one name, but in separate function.

Difference between Java and C++

Java	**C++**
Java is a true and complete object oriented language.	C++ is an extension of C with object oriented behaviour. C++ is not a complete object oriented language as that of Java.
Java does not provide template classes.	C++ offers Template classes.
Java supports multiple inheritance using interface.	C++ achieves multiple inheritance by permitting classes to inherit from multiple classes.
Java does not provide global variables.	Global variables can be declared in C++.
Java does not support pointers. Java	C++ supports pointers.

used reference to point to some object.	
In Java, destruction of objects is performed in finalize method.	In C++, destruction of objects is performed by destructor function.
In Java, we create objects dynamically only. Objects are in heap memory always	In C++, Object can be stored in stack or heap memory.
In Java Exception Handling, we can throw only objects which are inhered from Throwable Class.	In C++, we can throw any thing.
Java doesn't provide header files.	C++ has header files.

Java and Internet: Java is strongly associated with the Internet. Internet users can use Java to create applet programs and run them locally using a "Java-enabled browser" such as HotJava. They can also use a Java-enabled browser to download an applet located on a computer anywhere in the Internet and run it on his local computer. In fact, Java applets have made the Internet a true extension of the storage system of the local computer.

Internet users can also setup their websites containing java applets that could be used by other remote users of Internet. This feature made Java most popular programming language for Internet.

Java and World Wide Web: World Wide Web (WWW) is an open-ended information retrieval system designed to be used in the Internet's distributed environment. This system contains Web pages that provide both information and controls. Web system is open-ended and we can navigate to a new document in any direction. This is made possible with the help of a language called *Hypertext Markup Language* (HTML). Web pages contain HTML tags that enable us to find, retrieve, manipulate and display documents worldwide.

Java was meant to be used in distributed environments such as Internet. Since, both the Web and Java share the same philosophy, Java could be easily incorporated into the Web system. Before Java, the World Wide Web was limited to the display of still images and texts. However, the

incorporation of Java into Web pages has made it capable of supporting animation, graphics, games, and a wide range of special effects.

Symbolic Constants :

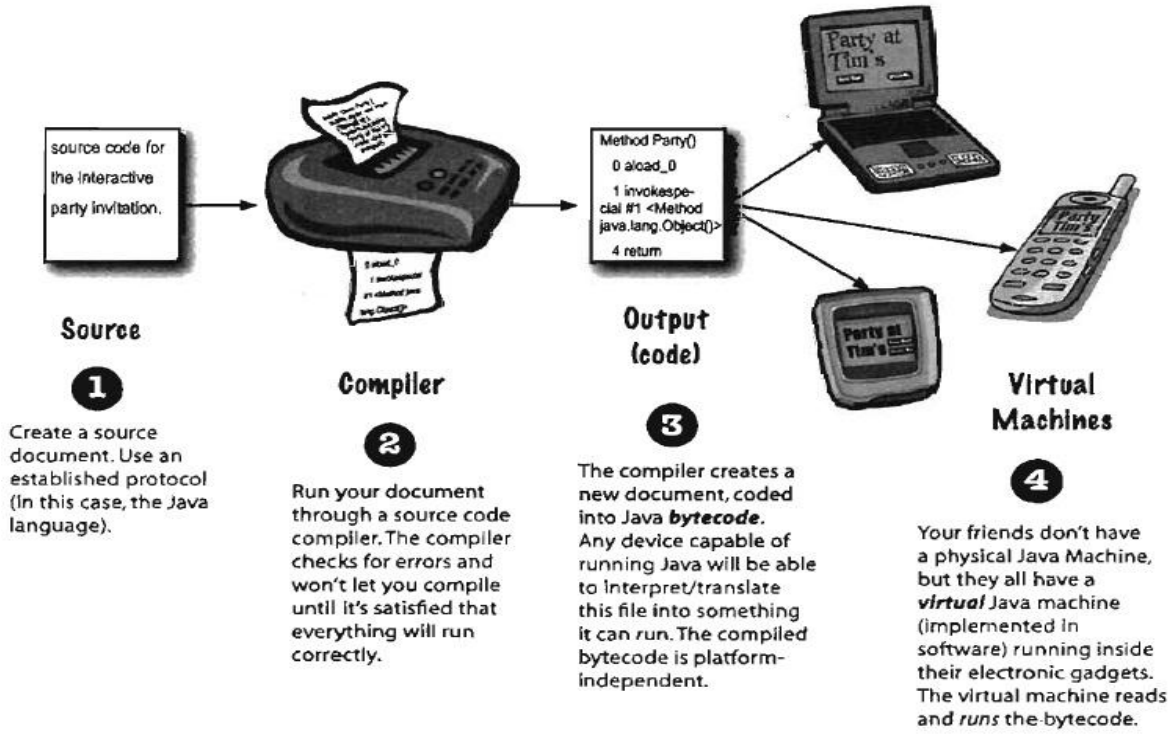
Final variables serve as symbolic constants. A final variable declaration is qualified with the reserved word **final**. The variable is set to a value in the declaration and cannot be reset. Any such attempt is caught at compile time.

Example:

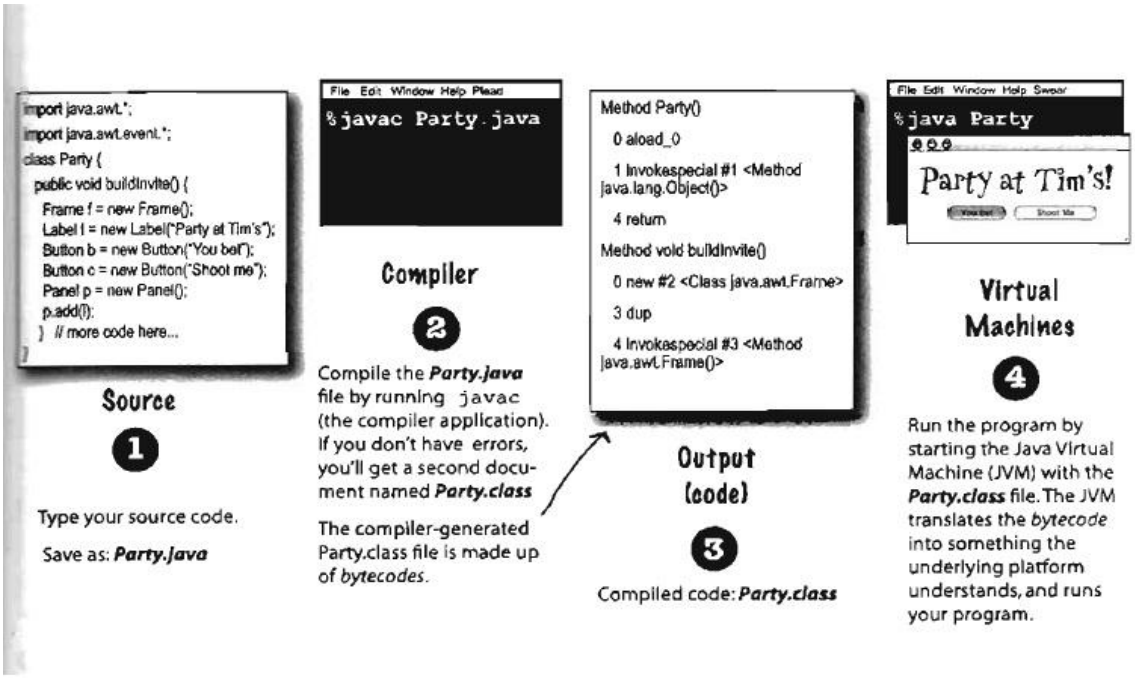
```
public class Student{
  public static final int NUM_GRADES = 5;

   private String name;
   private int[] grades;
   public Student(String name){
      this.name = name;
      this.grades = new int[NUM_GRADES];
   }
}
```

1.1 Java byte code power : As java compiler internally convert the java source code into byte code , and this byte code is platform independent means byte code generated by java compiler can be accessible in every platform that install java virtual machine.



What we will do in java – Programmer writes java source file using note pad or some java editors and save this source file with .java extension. This .java file is compiled using **javac** compiler which removes all the syntax errors in your code and if it satisfied , it creates a file with .class extension which contains the byte code as shown in below diagram.



This byte code is accessible on any platform , that's why java is called platform independent.

1.2 Java Features

1.Simple:

Java is easy to learn when compared to other languages, because it adapted the syntax's from c and c++. This makes java programmer to learn the language syntax faster. All the features which is adapted from c have the same syntax in java as in c.

2. OO (Object Orientedness) implementation

In java, if we don't know Object Oriented concepts, we cannot even write a hello world program also. Because in java , main is also kept in class , so even to run the hello world program , we have to write the class . Java implements code according to Object Oriented Paradigm. Each and every thing is java is kept inside classes and to access the features that class provides we require objects. Using classes and objects and access modifiers of language we get encapsulation. Using Interface we can get abstraction. Method Overloading and Method Overriding is an example of Polymorphism. In java, Inheritance is also possible. So java implements all the four pillars of Object Oriented Concepts.

3. Dynamic
In java, All the user defined things are stored on heap memory , and heap memory is dynamic memory . In java , we can't define user defined objects on stack. All the user defined objects are created using the new operator, which is used to allocate memory on heap.

Dynamic, that means java has got pointers.

java has got pointers, but pointers cannot misbehave in a freak way and in all java, Because they can't ptr++ or ptr--. Internally java compiler controls the pointer working.

4. Robust
Memory leaks cannot happen, pointer cannot freak around in the memory. This happens because java itself have the responsibly of garbage collection , we can allocate memory using new operator in java , but java itself collected back this memory when object are not reachable .And exception handling is much more powerful in java as compared to previous languages , that make the java robust.

5. java is platform independent

What is the meaning of platform independence ?
Platform independence technically means We don't need to recompile our application across platforms. For Example , it we write a c code and save it as A.c , the after compiling this c code we get A.exe. A.c and A.exe both are not portable across operating system .

now in java world.
If we write A.java and compiled it using javac compiler , then it creates A.class , A.java and A.class both are same for windows and unix or for that matter any platform. but windows and unix don't understand A.class.to make windows and unix understand A.class , we need to use something called JVM. JVM is not same for windows and unix.jvm is **platform dependant** we don't need to think about it, because we don't build jvm's.

JVM is compulsorily required to run java programs. JVM is like an intermediary between Os and the java program. so we got a intermediary, so we got slowness. **So java is slow**.

6. **Type casting rules**
Data type enforcement is very strict, and we don't have this kind of a scrap called **sizeof**. In java , if we try to convert one datatype to another data type intentionally or mistakenly , then java don't allowed us to do so .Refer the type casting topic for further assistant.

7. **Multithreaded enviouement** :
Java supports multithreading , through which we can do multitasking in our program , so that we can utilize our processor more effectively.

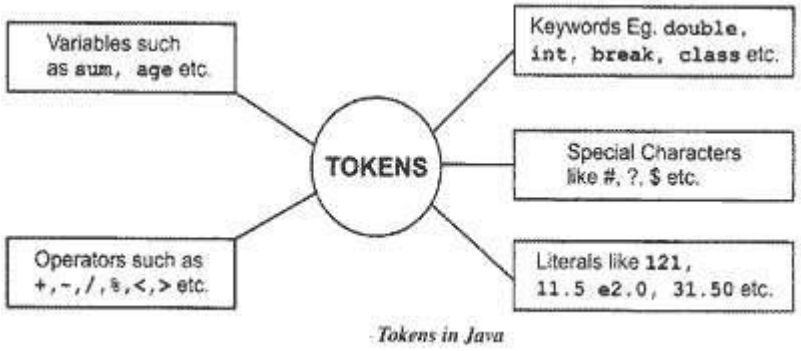
1.3 Language Building Blocks:

Like any other programming languages , java programming languages is defined by grammar rules that specify how the syntactically legal constructs can be formed using languages elements.

1.3.1 Lexical tokens/ tokens

Java Tokens:- A java Program is made up of Classes and Methods and in the Methods are the Container of the various Statements And a Statement is made up of Variables, Constants, operators etc .

Tokens are the various Java program elements which are identified by the compiler. A token is the smallest element of a program that is meaningful to the compiler. Tokens supported in Java include keywords, variables, constants, special characters, operations etc.



Tokens in Java

When you compile a program, the compiler scans the text in your source code and extracts individual tokens. While tokenizing the source file, the compiler recognizes and subsequently removes whitespaces (spaces, tabs, newline and form feeds) and the text enclosed within comments. Now let us consider a program

```
//Print Hello
Public class Hello
{
Public static void main(String args[])
{
System.out.println("Hello Java");
}
}
```

The source code contains tokens such as public, class, Hello, {, public, static, void, main, (, String, [], args, {, System, out, println, (, "Hello Java",),

}. The resulting tokens· are compiled into Java bytecodes that is capable of being run from within an interpreted java environment. Token are useful for compiler to detect errors. When tokens are not arranged in a particular sequence, the compiler generates an error message.

Tokens are the smallest unit of Program There is Five Types of Tokens

1) Reserve Word or Keywords
2) Identifier
3) Literals
4) Operators (Topic 1.7)
5) Separators

Identifiers

A name in a program is called identifiers . Identifiers can be used to denote classes , methods and variables.

In java identifiers is composed as a sequence of characters where each letter can be either a digit, a letter , a currency symbol like $ or can be underscore (_).

Note:

- Java identifiers cannot start with the digit .
- Java is case sensitive languages e.g price and Price are different in java

Keywords

Keywords are the reserved identifiers that are predefined in the languages and cannot be used to denote other entities.

Keywords in Java

abstract	default	implements	protected	throw
assert	Do	import	public	throws
boolean	double	instanceof	return	transient
break	else	int	short	try
byte	extends	interface	static	void
case	final	long	strictfp	volatile
catch	finally	native	super	while
char	float	new	switch	

class	for	package	synchronized	
continue	If	private	this	
Reserved Literals in Java				
Null		true	false	
Reserved Keywords not Currently in Use				
Const		goto		

- Separators

  Separators help define the structure of a program. The separators used in HelloWorld are parentheses, ( ), braces, { }, the period, ., and the semicolon, ;. The table lists the six Java separators (nine if you count opening and closing separators as two). Following are the some characters which are generally used as the separators in Java.

Separator	Name	Use
.	Period	It is used to separate the package name from sub-package name & class name. It is also used to separate variable or method from its object or instance.
,	Comma	It is used to separate the consecutive parameters in the method definition. It is also used to separate the consecutive variables of same type while declaration.
;	Semicolon	It is used to terminate the statement in Java.
()	Parenthesis	This holds the list of parameters in method definition. Also used in control statements & type casting.
{}	Braces	This is used to define the block/scope of code, class, methods.
[]	Brackets	It is used in array declaration.
Separators in Java		

1.4 Comments

A program can be documented by inserting comments at relevant places . these comments are ignored by the compiler. It is not considered as a coding part of a project .

Java provides three types of comments

- Single Line comment
- Multi line comments or Block Comment
- A documentation (or Javadoc) Comment

Single line comments

Single line Comments is used to comment a single line in a program . Single line comment syntax is

// this comments ends at the end of this line.

MultiLine Comments

A multiline comments can span several lines . Such a comments starts with /* and ends with */.Multiline comments are also called block comment.

/* A comment

On several lines

*/

Documentation Comment

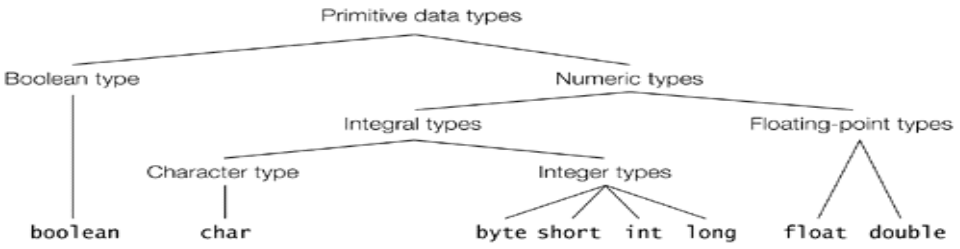
A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the javadoc tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods and field definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /** and ends with */:

1.5 Primitive Datatypes

Primitive data types in java is divided into three main categories

- Integral types consisting of integers and char.
- Floating Point types includes float and double.
- Boolean Type

Primitive Data Types in Java



- **Integral datatypes :** As shown in above diagram , integral datatypes are divided in integers types and character types in java .

  Integer types : in java integers types are divided in four sub types

Range of Integer Values			
Data Type	**Width (bits)**	**Minimum value MIN_VALUE**	**Maximum value MAX_VALUE**
byte	8	-2^7 (-128)	2^7-1 (+127)
short	16	-2^{15} (-32768)	$2^{15}-1$ (+32767)
int	32	-2^{31} (-2147483648)	$2^{31}-1$ (+2147483647)
long	64	-2^{63} (-9223372036854775808L)	$2^{63}-1$ (+9223372036854775807L)

Note:

- Range of datatype specifies the range of value that a variable can hold. For Example range of byte is -128 to 127 . so we can represent number between -128 to 127 by using byte datatype. If value is greater then 127 or less then -128 , then it is not possible with byte.
- Primitive data values are atomic values and are not objects. Atomic means they can't be further divided.
- Each Primitive data type has a corresponding wrapper class that can be used to represent a primitive data type as an object.

Character type

Character datatype is used to hold the character value , it is represented using the char keyword in java.

Range of Character Values			
Data Type	**Width (bits)**	**Minimum Unicode value**	**Maximum Unicode value**
char	16	0x0 (\u0000)	0xffff (\uffff)

Note : java supports Unicode character set to represent the characters and other special symbols .

2.Floating Point Numbers

Floating points numbers are used to represents the numbers that have fractional values like 10.10 .There are two ways to represent floating number is java described in below table.

Range of Floating-point Values			
Data Type	**Width (bits)**	**Minimum Positive Value MIN_VALUE**	**Maximum Positive Value MAX_VALUE**
float	32	1.401298464324817E-45f	3.402823476638528860e+38f
double	64	4.94065645841246544e-324	1.79769313486231570e+308

3.Booleans Datatypes

Boolean data types is used to represent logical values that can be either true or false . Width is not applicable for Boolean types variables.

Note :

- In java , all the relational , conditional and Boolean logical operators returns Boolean values. All the control statements in java depends on the Boolean value.
- In java , Booleans values cannot be converted to other primitive data types and vice versa is also true.

Summary of Primitive Data Types			
Data Type	**Width (bits)**	**Minimum Value, Maximum Value**	**Wrapper Class**
boolean	not applicable	true, false (no ordering implied)	Boolean
byte	8	-2^7, 2^7-1	Byte
short	16	-2^{15}, $2^{15}-1$	Short
char	16	0x0, 0xffff	Character
int	32	-2^{31}, $2^{31}-1$	Integer
long	64	-2^{63}, $2^{63}-1$	Long
float	32	±1.40129846432481707e-45f, ±3.40282347663852860e+38f	Float
double	64	\'b14.94065645841246544e-324, \'b11.79769313486231570e+308	Double

1.6 Default values for primitive data types

Default values of primitive data types are used when we are not initialize the non local variables in java . Non local variables in java are initialized by its default values. For Example , if we initialize the non local int variable , it its initializes with 0. Variables for a function/methods are called local variables , all other variables are called non local variables.

Default Values	
Data Type	**Default Value**
Boolean	false
Char	'\u0000'
Integer (byte, short, int, long)	0L for long, 0 for others
Floating-point (float, double)	0.0F or 0.0D
Reference types	null

1.7 Operators in java

A symbol that represents a specific action is called operators . For Example plus sign (+) is an operator that represents the addition of two numbers.

Precedence and Associativity Rules for operators : Precedence and Associativity rules are necessary for the deterministic evaluation of the expression . For example if we have the below expression

```
int a = 5+4*5;
```

it can be understand as (5+4) * 5 or 5 + (4*5). In both cases the value for the variable a is different , so to avoid such type of undeterministic result , java provides the Associativity and precedence (priority) rules on operators.

Precedence rules are used to determine which operator should be applied first if there are two operators with difference precedence. Operator with the highest precedence is applied first . Precedence for each operators is study during the explanation of operator in this chapter later.

Associativity rules are used to determine which operator should be applied first if there are two operators with the same precedence.

Left Associativity implies grouping from left to right.

e.g. 1+2-3 if it applies left Associativity then it is considered as

((1+2)-3)

Right Associativity implies grouping from right to left.

e.g. 1+2-3 if it applies right Associativity then it is considered as

(1+(2-3))

1.7.1 Simple Assignment Operator (=)

Assignment operators assign the right hand side value(expression) of = to the left hand side of variable. Assignment operator has the following syntax

<variable>=<expression>

Note:

- Expression value is calculated first, then the result of expression is assign to variable.
- Assignment operators = writes over the previous value of the destination variable

  For Ex: suppose if we have int i=5;

  In the next line ,Let we write i=5+4;

  Then i becomes 9 , overwrites the 5.
- The destination variable and source expression must be type compatible.

For Ex :

```
int i=5;  //fine
int j=5.5;//not fine because 5.5 is a flaot value, integer datatye is not able to take float values , so we need type casting.
```

- Precedence of Assignment operator is minimum in all operators.
- Associativity of Assignment operator is right to left.

  Ex : int i,j;

  ```
  i=j=10;   //(i=(j=10))
  ```

  so Associtivity is right to left , so first 10 is assign to j , then j value is assign to i .

  i=j=10; //Mutiple Assignment in same line.

1.7.2 Arithmetic operators

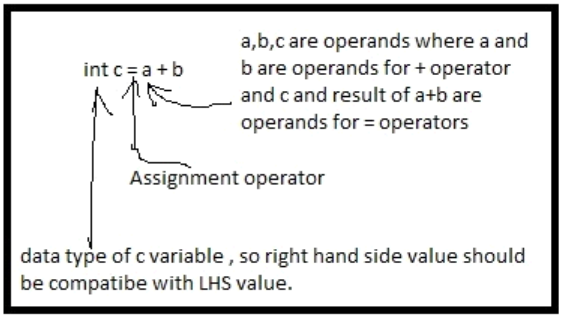
Arithmetic operators are used to constructs the mathematical expression as in algebra.

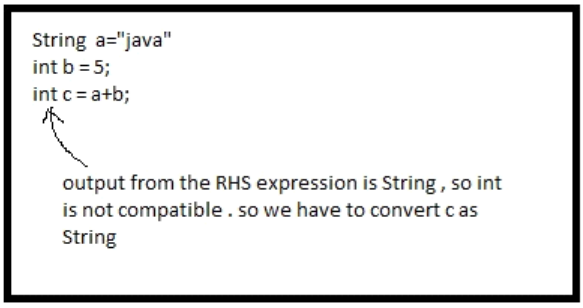
Ex: int a =5;

int b =10;

int c = a+b;

In above Example + operator is used to add two operand . if any one of two is of string type then it concatenates the two operands.





Arithmetic Operators						
Binary	*	Multiplication	/	Division	%	Remainder
	+	Addition	-	Subtraction		

Note:

- All the arithmetic operators are binary operators. They require two operands as shown in diagram.
- All arithmethic operators have left to right associativity.
- Precedence of arithmetic operators are shown in operators diagram above.Multiplication , division and Remainder(modulus) operator have higher precedence then addition and subtraction. Operators in the same row hava equal precedence in above diagram.
- If there are different precedence operators in an expression , then we applied the precedence rules for avoiding undeterministic behaviour .
- If there are same precedence operators in an expression , then we applied the associativity rules for avoiding underministic behaviour.

int a = 5+4*6-3;

1.in the expresion we hava three operators +,*,- .
2. Applied the precedence rules to solve the exprssion.
3. Multiply have higher precdence then +,-.
4. Expression will be reduced as 5+24-3.
5. Now + and - hve same precedence , so we have to applied the associativity rules.
6. As Associativity for arithmetic operators is left to right.Expression will be reduced as 29-3.
7. Variable a will be 26.

Multiplication Operator: *

Multiplication operator * multiplies two numbers.

```
int    sameSigns     = -4    * -8;        // result:  32
double oppositeSigns =  4.0  * -8.0;      // result: -32.0
int    zero          =  0    * -0;        // result:   0
```

Division Operator: /

The division operator / is overloaded. If its operands are integral, the operation results in integer division.

```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1 by integer division. d1 gets the value 1.0.
```

Integer division always returns the quotient as an integer value, i.e. the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type.

If any of the operands is a floating-point type, the operation performs floating-point division.

```
double d2 = 4.0 / 8;      // result: 0.5
double d3 = 8 / 8.0;      // result: 1.0
double d4 = 12.0F / 8;    // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;      // ((12.0 / 4.0) * 3.0) which is 9
double result2 = 12.0 * 3.0 / 4.0;      // ((12.0 * 3.0) / 4.0) which is 9
```

Remainder Operator: %

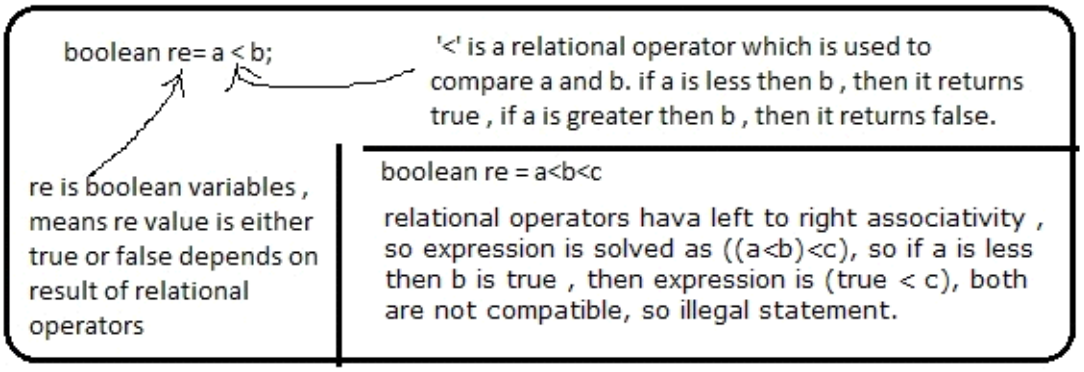
In mathematics, when we divide a number (the dividend) by a another number (the divisor), the result can be expressed in terms of a quotient and a remainder. For example, dividing 7 by 5, the quotient is 1 and the remainder is 2. The remainder operator % returns the remainder of the division performed on the operands.

```
int quotient  = 7 / 5;   // Integer division operation: 1
int remainder = 7 % 5;   // Integer remainder operation: 2
```

1.7.3 Relational operators

Relational Operators	
a < b	a less than b?
a <= b	a less than or equal to b?
a > b	a greater than b?
a >= b	a greater than or equal to b?

- Relational operators are used to compare(relate) the two operands, All the relational operators returns Boolean value in java.
- Relational operators are binary operators means they require two operands . and there operands are numeric expressions.
- Relational operators are non associative.
- Relational operators have precedence lower then the arithmetic operators but higher than assignment operator.



1.7.4 Equality Operator(==)

Equality Operators	
a == b	a and b are equal? That is, have the same primitive value? (Equality)
a != b	a and b are not equal? That is, do not have the same primitive value? (Inequality)

The equality operator == and the inequality operator != can be used to compare primitive data values, including boolean values.

Note:

1.Equality operators have precedence less then the relational operators but greater then the assignment operator.

2. Associativity for equality operators are left to right.

Ex:

```
double  hours = 45.5;

boolean overtime = hours >= 35.0;     // true.
boolean order = 'A' < 'a';            // true. It compares the integer value for A and a . A = 65 and a=97 , so condition is true.
```

1.7.5 Boolean Logical Operators: !, ^, &, |

Boolean logical operators include the unary operator ! (logical complement) and the binary operators & (logical AND), | (logical inclusive OR), and ^ (logical exclusive OR, a.k.a. logical XOR). Boolean logical operators can be applied to boolean operands, returning a boolean value. The operators &, |, and ^ can also be applied to integral operands to perform bitwise logical operations

Boolean Logical Operators		
Logical complement	!x	Returns the complement of the truth-value of x.
Logical AND	x & y	true if both operands are true; otherwise, false.
Logical OR	x | y	true if either or both operands are true; otherwise, false.
Logical XOR	x ^ y	true if and only if one operand is true; otherwise, false.

Truth-values for Boolean Logical Operators					
x	**y**	**!x**	**x & y**	**x | y**	**x ^ y**
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Note:

- Boolean logical operators have precedence less then arithmetic and relational operator but greater then assignment and Conditional AND and OR operator.
- In evaluation of boolean expressions involving boolean logical AND, XOR, and OR operators, both the operands are evaluated. The order of operand evaluation is always from left to right.

1.7.6 Conditional Operators: &&, ||

Conditional operators && and || are similar to their counterpart logical operators & and |, except that their evaluation is short-circuited. Given that x and y represent values of boolean expressions, the conditional operators are defined in below table. In the table, the operators are listed in decreasing precedence order.

Conditional Operators		
Conditional AND	x && y	true if both operands are true; otherwise, false.
Conditional OR	x || y	true if either or both operands are true; otherwise, false.

Unlike their logical counterparts & and |, which can also be applied to integral operands for bitwise operations, the conditional operators && and || can only be applied to boolean operands. Their evaluation results in a boolean value. Truth-values for conditional operators are shown in Table. Not surprisingly, they have the same truth-values as their counterpart logical operators.

Truth-values for Conditional Operators			
x	**y**	**x && y**	**x || y**
true	true	True	true
true	false	False	true
false	true	False	true
false	false	False	false

Short-circuit Evaluation

In evaluation of boolean expressions involving conditional AND and OR, the left-hand operand is evaluated before the right one, and the evaluation is short-circuited (i.e., if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated). In other words, the right-hand operand is evaluated conditionally.

The binary conditional operators have precedence lower than either arithmetic, relational, or logical operators, but higher than assignment operators. The following examples illustrate usage of conditional operators:

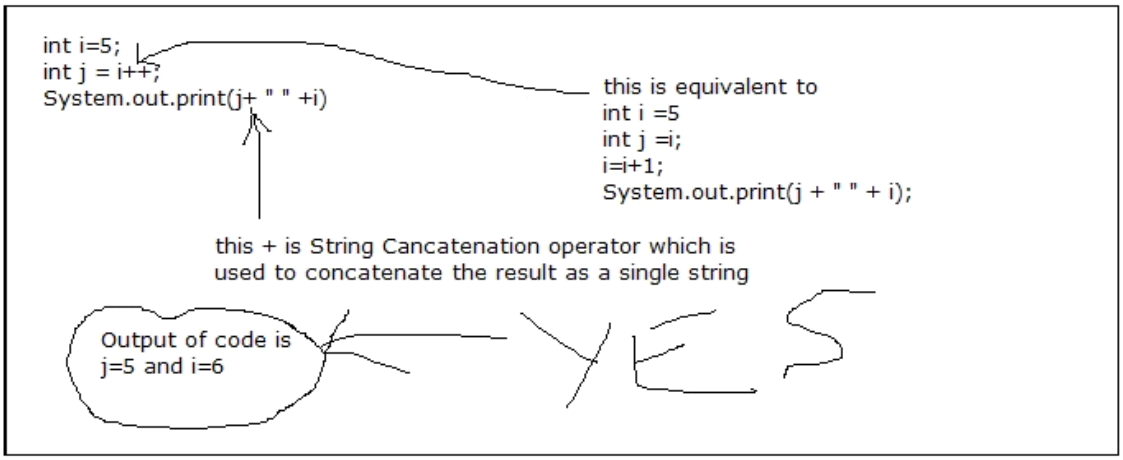
```
boolean b1 = 4 == 2 && 1 < 4;   // false, short-circuit evaluated as
                                // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;    // true, short-circuit evaluated as
                                // (b2 = ((!b1) || (2.5 > 8)))
boolean b3 = !(b1 && b2);       // true
boolean b4 = b1 || !b3 && b2;   // false, short-circuit evaluated as
                                // (b4 = (b1 || ((!b3) && b2)))
```

1.7.7 Increment(++) and decrement(--) operators

Increment operators and decrement operators are used to increment and decrement the value by 1.Increment and decrement operators comes in two flavours: postfix and prefix.

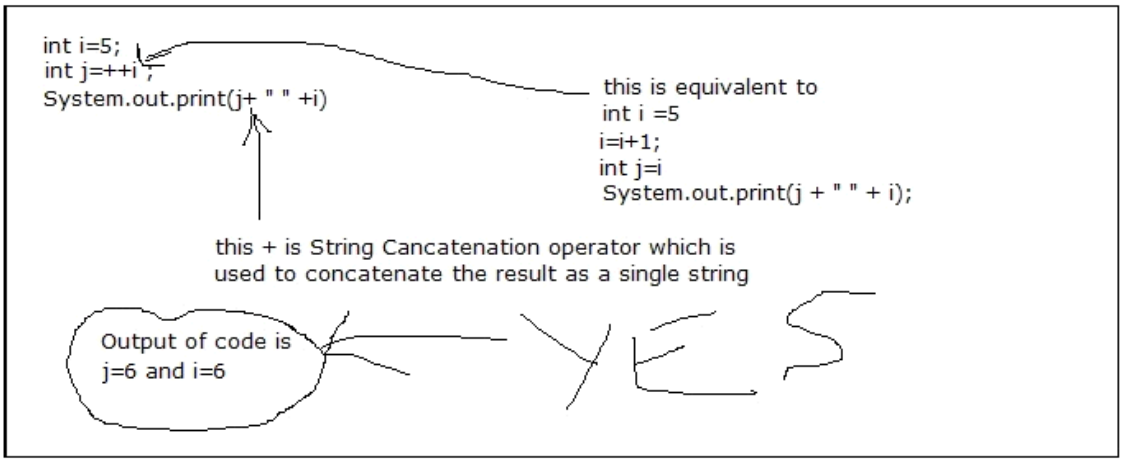
Postfix increment operator: post increment operator have the below syntax

i++ uses the current value of i as the value of expression first , then it is incremented by 1.
Below diagram explains the postIncrement operator.



PreIncrement operator : pre increment operator have the below syntax

++i add 1 to i first , then uses the new value of i as the value for the expression .



Same story is applicable for the postdecrement and predecrement except it decrements the value by 1.

1.8 Arrays:

An array is a data structure that defines an indexed collection of a fixed number of homogeneous data elements. This means that all elements in the array have the same data type. A position in the array is indicated by a non-negative integer value called the index. An element at a given position in the

array is accessed using the index. The size of an array is fixed and cannot increase to accommodate more elements.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a final field called length, which specifies the array size, that is, the number of elements the array can accommodate. The first element is always at index 0 and the last element at index n-1, where n is the value of the length field in the array.

Simple arrays are one-dimensional arrays, that is, a simple sequence of values. Since arrays can store object references, the objects referenced can also be array objects. This allows implementation of array of arrays.

Declaring Array Variables

An array variable declaration has either the following syntax:

<element type>[] <array name>;

Or

<element type> <array name>[];

where <element type> can be a primitive data type or a reference type. The array variable <array name> has the type <element type>[]. Note that the array size is not specified. This means that the array variable <array name> can be assigned an array of any length, as long as its elements have <element type>.

It is important to understand that the declaration does not actually create an array. It only declares a reference that can denote an array object.

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

These two declarations declare anIntArray and mediumPizzas to be reference variables that can denote arrays of int values and arrays of Pizza objects, respectively. The variable largePizzas can denote an array of pizzas, but the variable oneInteger cannot denote an array of int values—it is simply an int variable.

When the [] notation follows the type, all variables in the declaration are arrays. Otherwise the [] notation must follow each individual array name in the declaration.

An array variable that is declared as a member of a class, but is not initialized to denote an array, will be initialized to the default reference value null. This default initialization does not apply to local reference variables and, therefore, does not apply to local array variables .

Constructing an Array

An array can be constructed for a specific number of elements of the element type, using the new operator. The resulting array reference can be assigned to an array variable of the corresponding type.

<array name> = new <element type> [<array size>];

The minimum value of <array size> is 0 (i.e., arrays with zero elements can be constructed in Java). If the array size is negative, a **NegativeArraySizeException** is thrown.

Given the following array declarations:

```
int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

the arrays can be constructed as follows:

```
anIntArray = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];       // array of 5 pizzas
largePizzas = new Pizza[3];        // array of 3 pizzas
```

The array declaration and construction can be combined.

<element type$_1$>[] <array name> = new <element type$_2$>[<array size>];

However, here array type <element type$_2$>[] must be assignable to array type <element type$_1$>[] .When the array is constructed, all its elements are initialized to the default value for <element type$_2$>. This is true for both member and local arrays when they are constructed.

In all the examples below, the code constructs the array and the array elements are implicitly initialized to their default value. For example, the element at index 2 in array anIntArray gets the value 0, and the element at index 3 in array mediumPizzas gets the value null when the arrays are constructed.

```
int[] anIntArray = new int[10];            // Default element value: 0.

Pizza[] mediumPizzas = new Pizza[5];       // Default element value: null.

// Pizza class extends Object class
Object objArray = new Pizza[3];            // Default element value: null.
```

```
// Pizza class implements Eatable interface
Eatable[] eatables = new Pizza[2];          // Default element value: null.
```

The value of the field length in each array is set to the number of elements specified during the construction of the array; for example, medium Pizzas.length has the value 5.

Once an array has been constructed, its elements can also be explicitly initialized individually; for example, in a loop. Examples in the rest of this section make heavy use of a loop to traverse through the elements of an array for various purposes.

Initializing an Array

Java provides the means of declaring, constructing, and explicitly initializing an array in one declaration statement:

<element type>[] <array name> = { <array initialize list> };

This form of initialization applies to member as well as local arrays. The <array initialize list> is a comma-separated list of zero or more expressions. Such an array initialization block results in the construction and initialization of the array.

```
int[] anIntArray = {1, 3, 49, 2, 6, 7, 15, 2, 1, 5};
```

The array anIntArray is declared as an array of ints. It is constructed to hold 10 elements (equal to the length of the list of expressions in the block), where the first element is initialized to the value of the first expression (1), the second element to the value of the second expression (3), and so on.

```
// Pizza class extends Object class
Object[] objArray = { new Pizza(), new Pizza(), null };
```

The array objArray is declared as an array of the Object class, constructed to hold three elements. The initialization code sets the first two elements of the array to refer to two Pizza objects, while the last element is initialized to the null reference. Note that the number of objects created in the above declaration statement is actually three: the array object with three references and the two Pizza objects.

The expressions in the <array initialize list> are evaluated from left to right, and the array name obviously cannot occur in any of the expressions in the list. In the examples above, the <array initialize list> is terminated by the right curly bracket , }, of the block. The list can also be legally terminated by a comma. The following array has length two, not three:

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

Using an Array

The whole array is referenced by the array name, but individual array elements are accessed by specifying an index with the [] operator. The array element access expression has the following syntax:

<array name> [<index expression>]

Each individual element is treated as a simple variable of the element type. The index is specified by the <index expression>, which can be any expression that evaluates to an non-negative int value. Since the lower bound of an array is always 0, the upper bound is one less than the array size, that is, (<array name>.length-1). The ith element in the array has index (i-1). At runtime, the index value is automatically checked to ensure that it is within the array index bounds. If the index value is less than 0 or greater than or equal to <array name>.length in an array element access expression, an **ArrayIndexOutOfBoundsException** is thrown.

1.9 Anonymous Arrays

As shown earlier in this section, the following declaration statement

<element type$_1$>[] <array name> = new <element type$_2$>[<array size>]; // (1)
int[] intArray = new int[5];

can be used to construct arrays using an array creation expression. The size of the array is specified in the array creation expression, which creates the array and initializes the array elements to their default values. On the other hand, the following declaration statement

<element type>[] <array name> = { <array initialize list> }; // (2)
int[] intArray = {3, 5, 2, 8, 6};

both creates the array and initializes the array elements to specific values given in the array initializer block. However, the array initialization block is not an expression.

Java has another array creation expression, called anonymous array, which allows the concept of the array creation expression from (1) and the array initializer block from (2) to be combined, to create and initialize an array object:

new <element type>[] { <array initialize list> }
new int[] {3, 5, 2, 8, 6}

The construct has enough information to create a nameless array of a specific type. Neither the name of the array nor the size of the array is specified. The construct returns an array reference that can be assigned and

passed as parameter. In particular, the following two examples of declaration statements are equivalent.

```
int[] intArray = {3, 5, 2, 8, 6};          // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6}; // (2)
```

In (1), an array initializer block is used to create and initialize the elements. In (2), an anonymous array expression is used. It is tempting to use the array initialization block as an expression; for example, in an assignment statement as a short cut for assigning values to array elements in one go. However, this is illegal—instead, an anonymous array expression should be used.

```
int[] daysInMonth;

daysInMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // Not ok.
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // ok.
```

1.10 Multidimensional Arrays

Since an array element can be an object reference and arrays are objects, array elements can themselves reference other arrays. In Java, an array of arrays can be defined as follows:

<element type>[][]...[] <array name>;or

<element type> <array name>[][]...[];

In fact, the sequence of square bracket pairs, [], indicating the number of dimensions, can be distributed as a postfix to both the element type and the array name. Arrays of arrays are also sometimes called multidimensional arrays.

The following declarations are all equivalent:

```
int[][] mXnArray;      // 2-dimensional array
int[]   mXnArray[];    // 2-dimensional array
int     mXnArray[][];  // 2-dimensional array
```

It is customary to combine the declaration with the construction of the multidimensional array.

```
int[][] mXnArray = new int[4][5];    // 4 x 5 matrix of ints
```

The previous declaration constructs an array mXnArray of four elements, where each element is an array (row) of 5 int values. The concept of rows and columns is often used to describe the dimensions of a 2-dimensional

array, which is often called a matrix. However, such an interpretation is not dictated by the Java language.

Multidimensional arrays can also be constructed and explicitly initialized using array initializer blocks discussed for simple arrays. Note that each row is an array which uses an array initializer block to specify its values:

```
double[][] identityMatrix = {
    {1.0, 0.0, 0.0, 0.0 }, // 1. row
    {0.0, 1.0, 0.0, 0.0 }, // 2. row
    {0.0, 0.0, 1.0, 0.0 }, // 3. row
    {0.0, 0.0, 0.0, 1.0 }  // 4. row
}; // 4 x 4 Floating-point matrix
```


Chapter 2 : Control Statements , Objects and classes

2.1 Selection Statements

Java provides selection statements that allow the program to choose between alternative actions during execution. The choice is based on criteria specified in the selection statement. These selection statements are

- simple if Statement
- if-else Statement
- switch Statement

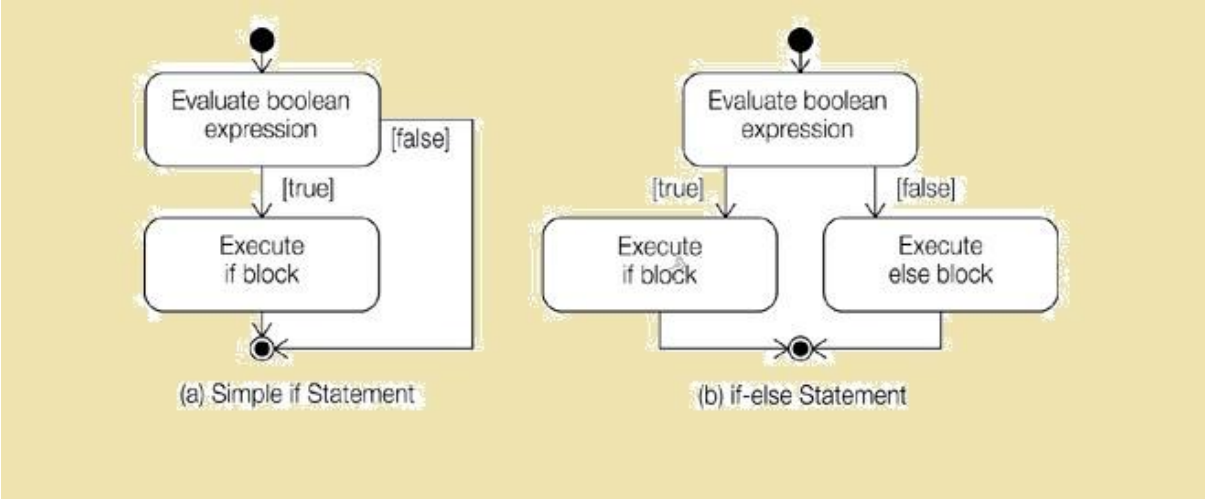
Simple if Statement

The simple if statement has the following syntax:

```
if (<conditional expression>)
      <statement>
```

It is used to decide whether an action is to be performed or not, based on a condition. The condition is specified by <conditional expression> and the action to be performed is specified by <statement>.

The semantics of the simple if statement are straightforward. The <conditional expression> is evaluated first. If its value is true, then <statement> (called the if block) is executed and execution continues with the rest of the program. If the value is false, then the if block is skipped and execution continues with the rest of the program.



In the following examples of the if statement, it is assumed that the variables and the methods have been defined appropriately:

```
if (emergency)            // emergency is a boolean variable
   operate(); // single statement in if , no need to mention curley braces

if (temperature > critical)
   soundAlarm();

if (isLeapYear() && endOfCentury())
   celebrate();

if (catIsAway()) {        // Block , we have to mention curley braces
   getFishingRod();
   goFishing();
}
```

Note that <statement> can be a block, and the block notation is necessary if more that one statement is to be executed when the <conditional expression> is true.

Since the <conditional expression> must be a **boolean** expression, it avoids a common programming error: using an expression of the form (a=b) as the condition, where inadvertently an assignment operator is used instead of a relational operator. The compiler will flag this as an **error**, unless both a and b are boolean.

Note that the if block can be any valid statement. In particular, it can be the empty statement (;) or the empty block ({}). A common programming error is an inadvertent use of the empty statement.

if (emergency); // Empty if block
operate(); // Executed regardless of whether it was an emergency or not.

if-else Statement

The if-else statement has the following syntax:

if (<conditional expression>)
    <statement$_1$>
else
    <statement$_2$>

It is used to decide between two actions, based on a condition.

The <conditional expression> is evaluated first. If its value is true, then <statement$_1$> (the if block) is executed and execution continues with the rest of the program. If the value is false, then <statement$_2$> (the else block) is executed and execution continues with the rest of the program. In other words, one of two mutually exclusive actions is performed. The else clause is optional; if omitted, the construct reduces to the simple if statement.

In the following examples of the if-else statement, it is assumed that all variables and methods have been defined appropriately:

Ex 1:
```
if (emergency)
    operate();
else
    joinQueue();
```

Ex 2
```
if (temperature > critical)
    soundAlarm();
else
    businessAsUsual();
```

The rule for matching an else clause is that an else clause always refers to the nearest if that is not already associated with another else clause. Block notation and proper indentation can be used to make the meaning obvious.

switch Statement

Conceptually the switch statement can be used to choose one among many alternative actions, based on the value of an expression. Its general form is as follows:

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

The syntax of the switch statement comprises a switch expression followed by the switch body, which is a block of statements. The type of the switch expression is non-long integral (i.e., char, byte, short, or int). The statements in the switch body can be labeled, defining entry points in the switch body where control can be transferred depending on the value of the switch expression. The semantics of the switch statement are as follows:

- The switch expression is evaluated first.
- The value of the switch expression is compared with the case labels. Control is transferred to the <statement$_i$> associated with the case label that is equal to the value of the switch expression. After execution of the associated statement, control falls through to the next statement unless appropriate action is taken.
- If no case label is equal to the value of the switch expression, the statement associated with the default label is executed.

All labels (including the default label) are optional and can be defined in any order in the switch body. There can be at most one default label in a switch statement. If it is left out and no valid case labels are found, the whole switch statement is skipped.

The case labels are constant expressions whose values must be unique, meaning no duplicate values are allowed. The case label values must be assignable to the type of the switch expression . In particular, the case label values must be in the range of the type of the switch expression. Note that the type of the case label cannot be boolean, long, or floating-point.

Example 5.2 Using break in switch Statement

```
public class Digits {

    public static void main(String[] args) {

   System.out.println(digitToString('7') + " " + digitToString('8') + " "
+digitToString('6'));
    }

    public static String digitToString(char digit) {
        String str = "";
        switch(digit) {
            case '1': str = "one";   break;
```

```
            case '2': str = "two";   break;
            case '3': str = "three"; break;
            case '4': str = "four";  break;
            case '5': str = "five";  break;
            case '6': str = "six";   break;
            case '7': str = "seven"; break;
            case '8': str = "eight"; break;
            case '9': str = "nine";  break;
            case '0': str = "zero";  break;
            default:  System.out.println(digit + " is not a digit!");
        }
        return str;
    }
}
```

Output from the program:

seven eight six

2.2 Iteration Statements

Loops allow a block of statements to be executed repeatedly (i.e., iterated). A boolean condition (called the loop condition) is commonly used to determine when to terminate the loop. The statements executed in the loop constitute the loop body. The loop body can be a single statement or a block.

Java provides three language constructs for constructing loops:

- while statement
- do-while statement
- for statement

These loops differ in the order in which they execute the loop body and test the loop condition. The while and the for loops test the loop condition before executing the loop body, while the do-while loop tests the loop condition after execution of the loop body.

while Statement

The syntax of the while loop is

```
while (<loop condition>)
    <loop body>
```

The loop condition is evaluated before executing the loop body. The while statement executes the loop body as long as the loop condition is true. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop. If the loop condition is false to begin with, the loop body is not executed at all. In other

words, a while loop can execute zero or more times. The loop condition must be a boolean expression.

The while statement is normally used when the number of iterations is not known a priori.

```
while (noSignOfLife())
    keepLooking();
```

Since the loop body can be any valid statement, inadvertently terminating each line with the empty statement (;) can give unintended results.

```
while (noSignOfLife());      // Empty statement as loop body!
    keepLooking();           // Statement not in the loop body.
```

do-while Statement

The syntax of the do-while loop is

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

The loop condition is evaluated after executing the loop body. The do-while statement executes the loop body until the loop condition becomes false. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop. Note that the loop body is executed at least once.

The loop body in a do-while loop is invariably a statement block. It is instructive to compare the while and the do-while loops. In the examples below, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

```
while (cat.isAway()) {       // (1)
    mice.play();
}

do {                         // (2)
    mice.play();
} while (cat.isAway());
```

for Statement

The for loop is the most general of all the loops. It is mostly used for counter-controlled loops, that is, when the number of iterations is known beforehand.

The syntax of the loop is as follows:

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

The <initialization> usually declares and initializes a loop variable that controls the execution of the <loop body>. The <loop condition> is a boolean expression, usually involving the loop variable, such that if the loop condition is true, the loop body is executed; otherwise, execution continues with the statement following the for loop. After each iteration (i.e., execution of the loop body), the <increment expression> is executed. This usually modifies the value of the loop variable to ensure eventual loop termination. The loop condition is then tested to determine if the loop body should be executed again. Note that the <initialization> is only executed once on entry to the loop.

```
<initialization>
while (<loop condition>) {
    <loop body>
    <increment expression>
}
```

The following code creates an int array and sums the elements in the array.

```
int sum = 0;
int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++)   // (1)
    sum += array[index];
```

The loop variable index is declared and initialized in the <initialization> section of the loop. It is incremented in the <increment expression> section. The for loop defines a local block such that the scope of this declaration is the for block, which comprises the <initialization>, the <loop condition>, the <loop body> and the <increment expression> sections.

The loop at (1) showed how a declaration statement can be specified in the <initialization> section. Such a declaration statement can also specify a comma-separated list of variables.

```
for (int i = 0, j = 1, k = 2; ... ; ...) ...;      // (2)
```

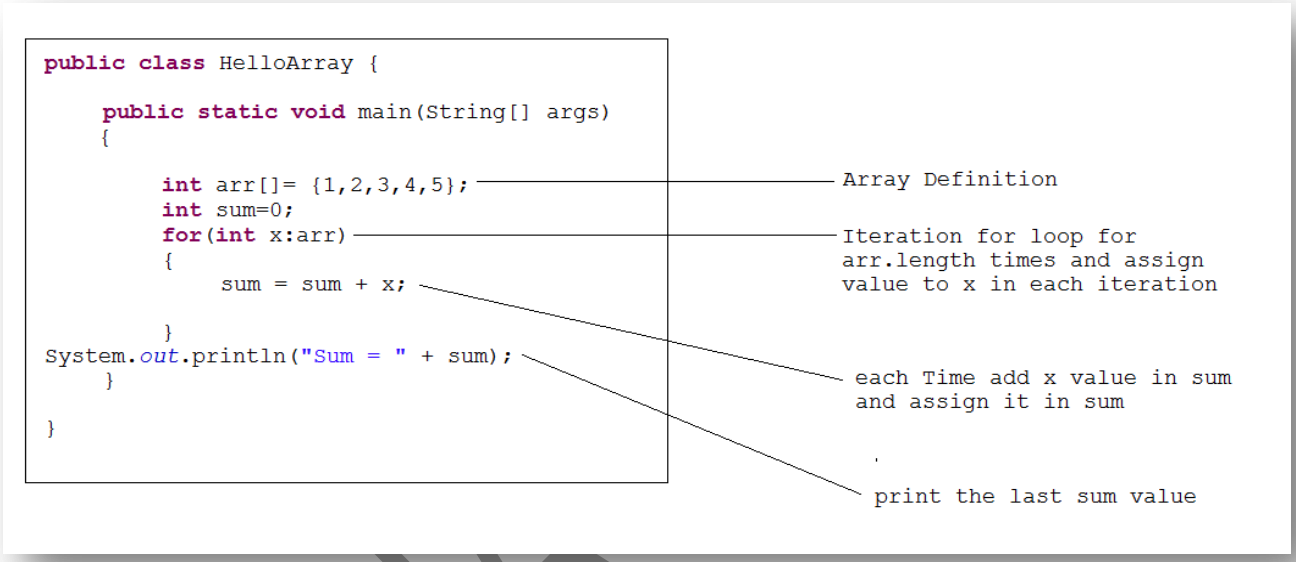
The variables i, j, and k in the declaration statement all have type int. All variables declared in the <initialization> section are local variables in the for block and obey the scope rules for local blocks.

For Each Statement

For each statement is an alternative to for loop which is generally used to iterate over collections and array. Suppose if we have a linear array which is initialized using initialize list as

int a[]={2,5,6,,8,4,9,10};

then to sum all the elements in this array , we require to traverse each element of the array and add it, so for this, we can used loops. Check the below diagram



2.3 Transfer Statements

Java provides six language constructs for transferring control in a program:

- break
- continue
- return
- try-catch-finally
- throw
- assert.

Note that Java does not have a goto statement, although goto is a reserved word.

break Statement

The break statement comes in two forms: the unlabeled and the labeled form.

break; // the unlabeled form

break <label>; // the labeled form

The unlabeled break statement terminates loops (for, while, do-while) and switch statements which contain the break statement, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, terminating the enclosing statement, with execution continuing after this statement.

In Below Example , the break statement at (1) is used to terminate a for loop. Control is transferred to (2) when the value of i is equal to 4 at (1), skipping the rest of the loop body and terminating the loop.

Example also shows that the unlabeled break statement only terminates the innermost loop or switch statement that contains the break statement. The break statement at (3) terminates the inner for loop when j is equal to 2, and execution continues in the outer switch statement at (4) after the for loop.

Example break Statement

```
class BreakOut {

    public static void main(String[] args) {

        for (int i = 1; i <= 5; ++i) {
            if (i == 4) break;       //      (1) Terminate loop. Control to (2).
              // Rest of loop body skipped when i gets the value 4.
            System.out.println(i + "\t" + Math.sqrt(i));
        } // end for
                                     //          (2) Continue here.

        int n = 2;
        switch (n) {
            case 1: System.out.println(n); break;
            case 2: System.out.println("Inner for loop: ");
                    for (int j = 0; j < n; j++)
                      if (j == 2)
                          break;         // (3) Terminate loop. Control to (4).
                      else
                          System.out.println(j);
            default: System.out.println("default: " + n); // (4) Continue here.
        }
    }
}
```

Output from the program:

```
1    1.0
2    1.4142135623730951
```

```
3    1.7320508075688772
Inner for loop:
0
1
default: 2
```

A labeled break statement can be used to terminate any labeled statement that contains the break statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

```
out:
{                       // (1) Labeled block
   // ...
   if (j == 10) break out; // (2) Terminate block. Control to (3).
   System.out.println(j);  // Rest of the block not executed if j == 10.
   // ...
}
// (3) Continue here.
```

continue Statement

Like the break statement, the continue statement also comes in two forms: the unlabeled and the labeled form.

```
continue;           // the unlabeled form
continue <label>;   // the labeled form
```

The continue statement can only be used in a for, while, or do-while loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible. In the case of the while and do-while loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the <loop condition>. In the case of the for loop, the rest of the loop body is skipped, with execution continuing with the <increment expression>.

In Example , an unlabeled continue statement is used to skip an iteration in a for loop. Control is transferred to (2) when the value of i is equal to 4 at (1), skipping the rest of the loop body and continuing with the <increment expression> in the for statement.

Example : continue Statement

```
class Skip {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; ++i) {
         if (i == 4) continue;                  // (1) Control to (2).
         // Rest of loop body skipped when i has the value 4.
         System.out.println(i + "\t" + Math.sqrt(i));
```

```
            // (2). Continue with increment expression.
        }                                   // end for
    }
}
```

Output from the program:

```
1       1.0
2       1.4142135623730951
3       1.7320508075688772
5       2.23606797749979
```

return Statement

The return statement is used to stop execution of a method and transfer control back to the calling code (a.k.a. the caller). The usage of the two forms of the return statement is dictated by whether it is used in a void or a non-void method. The first form does not return any value to the calling code, but the second form does. Note that the keyword void does not represent any type.

The <expression> must evaluate to a primitive value or a reference value, and its type must be assignable to the return type in the method prototype. A void method need not have a return statement —in which case control normally returns to the caller after the last statement in the method's body has been executed.

return Statement

Form of return Statement	In void Method	In Non-void Method
return;	optional	not allowed
return <expression>;	not allowed	mandatory

2.4 Classes

- Classes is a concept through which we can represent real world entity, Suppose if we want to represent house then class is used to describe the features of house
- Classes acts like a blueprint of the object. Through classes , we can define properties and behaviours of the object which is used to differentiate one object from other object .

  In java , properties of an object of a class, also called attributes, are defined using variables in java. Behaviours of an object of the class also known as operations , are defined using methods /functions in java.

- Class made the distinction between the contract and implementation provides for its object .

  Contract defines what services , and implementation defines how these services are provided by the classes.

- Through classes , we can achieve the abstraction in java . Abstraction is one of the fundamental way to handle the complexity . Abstraction denotes the essential properties and behaviours of an object .

Class syntax:

```
 class [className]
{

//Propeties of class by defining the variables in class , also called class
//variables or instance variables ;

//Behaviours of class by defining the methods in class , also called class
//methods

}
```

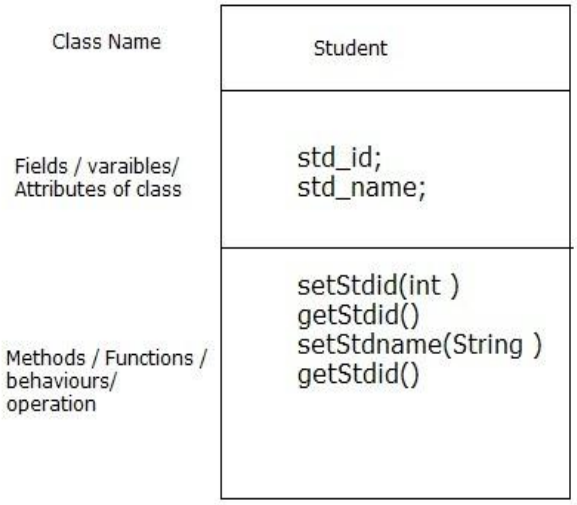
Ex: Suppose we want to implement the student ,so we create a student class having student_id and student_name as properties of the class and methods that set the student_id and student_name and methods that return the student_id and student_name on request.

```
class  student
{
Private int  std_id;
Private String  std_name;

      void setStdid(int stdid)
      {
            std_id=stdid;
      }
      int getStdid()
      {
            return std_id;
      }
      void setStdname(String stdname)
      {
            std_name=stdname;
      }
      String  getStdid()
      {
            return std_name;
```

```
    }
}
```

- Both class variables and methods constitutes the class Members.
- UML Notation



UML Diagram to represent class

2.5 Objects

- An objects is an instance of the class .
- Objects are something that have real existence . Classes without objects are not worth creating . Objects are the handler for accessing the class data members and member functions.
- An object must be explicitly created before it can be used in a program
- In java , objects are manipulated through object references.
- Process of creating object in java involves

1. **Declaration of reference variable**

   [Class name] [reference variable name];

   For Ex: Student st;

   Here Student is a class that is used to denote the student entity and st is the references of the student class .

2. **Creating an Object:**

Object creation in java involves the new operator.

Syntax: [Reference variable of class] = new [ClassName];

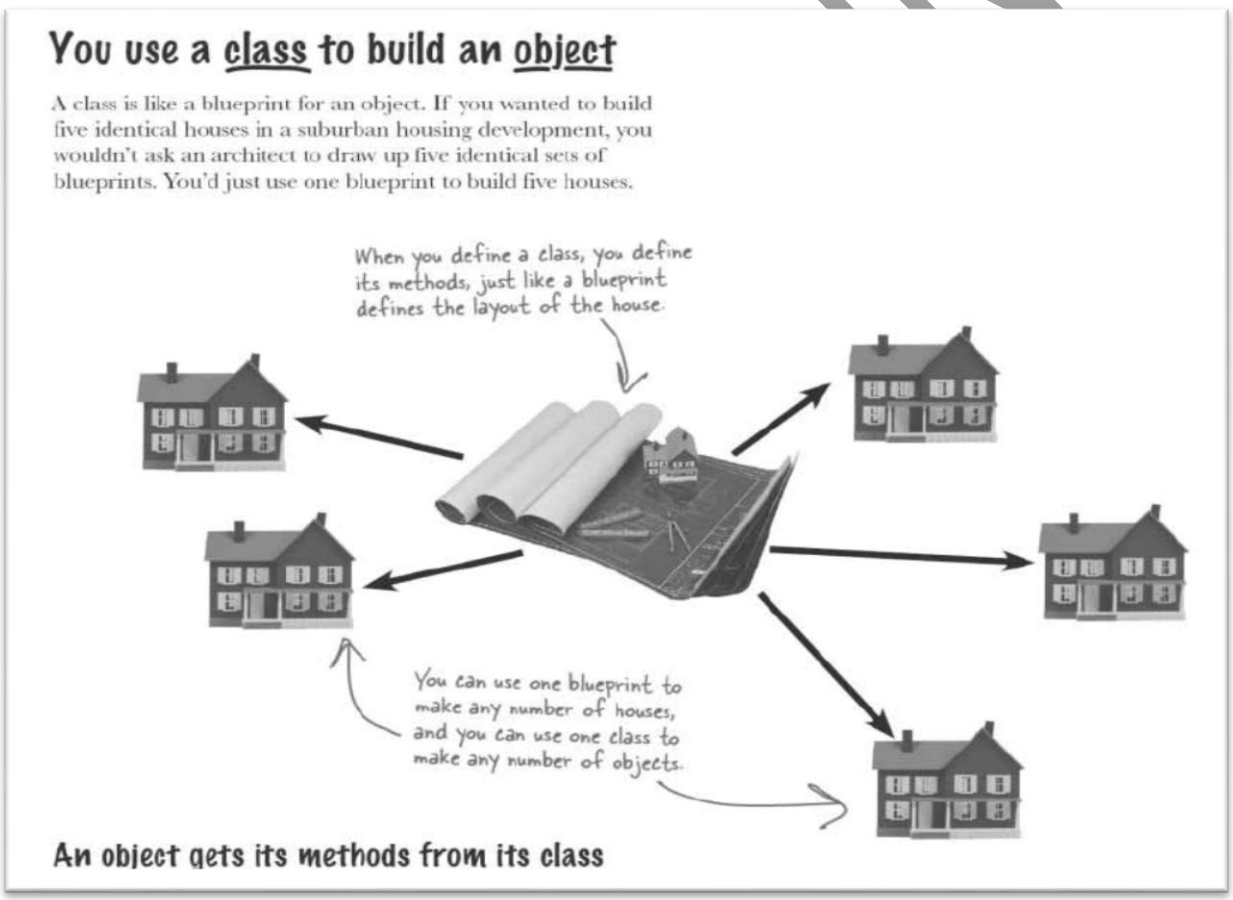
Ex: st=new Student();

new returns a reference to a new instance of the student class . which is assigned to the st reference variable. Now using the st reference variable , we can access the student class.

Note: In general , Object provides a handlers for the class , handler is used to access the variables and methods in a class .

We can also combine the a and b step.

For Ex: Student st = new Student();



2.6 Static and Non Static Members:

Static Members :

1. Static members are specific for the class .
2. Static members calls either using object name or class name.
3. Static members are initializes when the class is loaded in the memory.
4. Class Static variables are initialized with their default values when class loads in memory.
5. “this” keyword is not available .

Non Static Members :

1. Non static members are specific for the object , means its number of copies is dependent on the number of objects .
2. Non static member is called(used) by using the object of the class
3. Non static members are initialize when the constructor of the class is called.(when object is created)
4. “this” is available.

2.7 Accessing Class members :

Class members are access by using the object of the class with the dot operator (.), followed by the variable or method name . Static variables and methods can also be called without using object name . Static member can access either object name or class name followed by the variable name and method name , concatenated using dot operator .

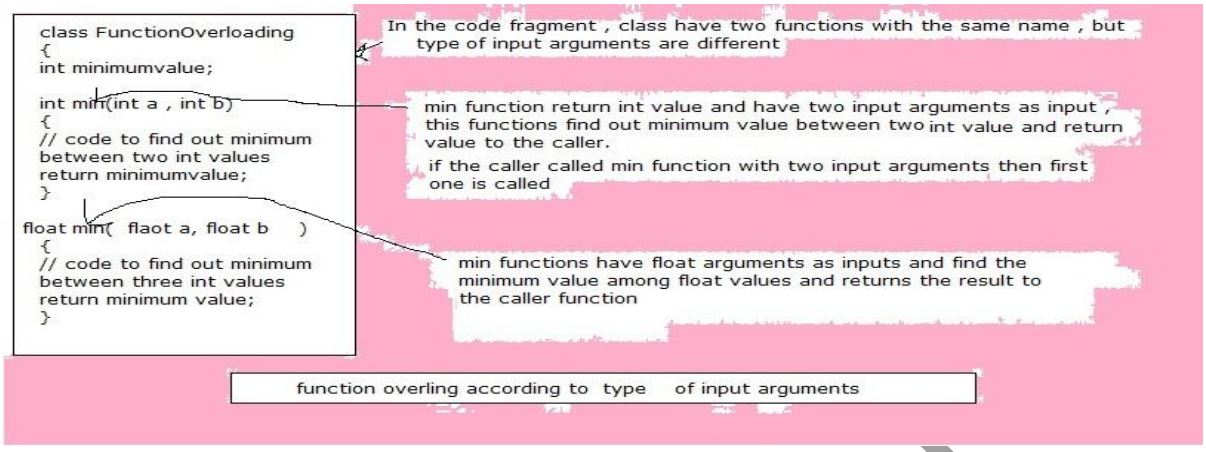
Non static variable and functions can be access by object of the class followed by the variable and method name concatenate with the dot operator.

2.8 Method Overloading :

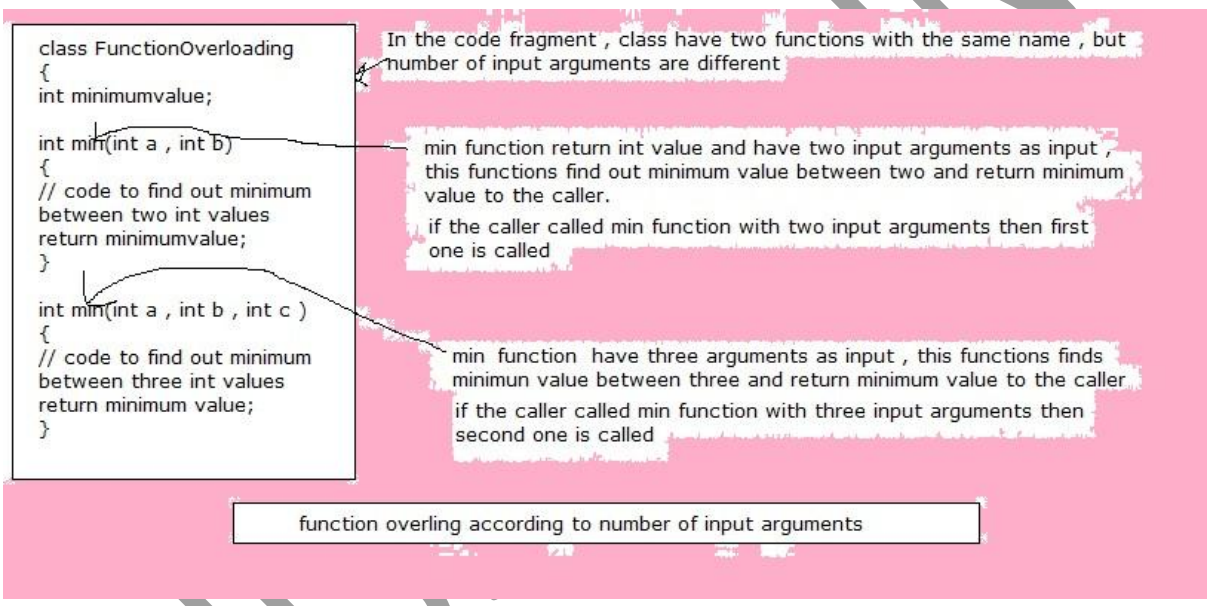
Each method has a signature , which is comprised of the name of the method and return type and number of the parameters in the argument list. Method overloading allows a method with the same name with different parameters , thus with different signature.

Method Overloading is a concept through which we can define more then one method with the same name in a class, but these methods must have different types of input arguments. Input arguments of methods can be differ according to

1. **Order of input arguments** : Order of input arguments means that the method should have argument that differ according to the order, in which it appears in the function . Suppose if we create two methods with the same name called max in a class, first method takes float as first argument and integer as second argument, and the second function takes integer as first argument and float as second argument.
2. **Type of input arguments :** Type of input arguments means that methods should have different type of input arguments as shown in the diagram.



3. Number of input arguments



Note: Methods with same name and same argument ,but different return type in a class are not overloaded methods.

2.9 Constructors

The main purpose of constructor is to set the initial state of an object when the object is created using the new operator

1. Constructor is a special function , special means , we never calls it , system calls it when we create an object of the class
2. Constructor name must be same as class name with no return type .For Example , if our class name is student , then constructor name is also student.

```
class Student
{
  int studid;
```

```
 Student() // constructor of student class
 {
 System.out.print("This is constructor of student class");
 }
}
```

3. Constructor is used to set the initial value of object with their defaults values .
4. There are two types of constructor , default constructor and parameterized constructor , constructor with no input arguments are called default constructor , constructor with the input arguments are called parameterized constructor.
5. By default , if we don't write any constructor , then java provides us a default constructor with empty body which initialize the state of object with default vales.
6. If we write any constructor, either default or parameterized then java don't provide us a default constructor.
7. Modifiers other than Access modifiers are not permitted .
8. Constructor can used not to create an object of the class
9. Constructor can used to create an object conditionally.
10. Constructor overloading is possible.

2.10 Nested and Inner class :

A class that is declared within another class or interface, is called a nested class. Similarly, an interface that is declared within another class or interface, is called a nested interface. A top-level class or a top-level interface is one that is not nested.

In addition to the top-level classes and interfaces, there are four categories of nested classes and one of nested interfaces, defined by the context these classes and interfaces are declared in:

- static member classes and interfaces
- non-static member classes
- local classes
- anonymous classes

The last three categories are collectively known as inner classes. They differ from non-inner classes in one important aspect: that an instance of an inner class may be associated with an instance of the enclosing class. The instance of the enclosing class is called the immediately enclosing instance. An instance of an inner class can access the members of its immediately enclosing instance by their simple name.

A static member class or interface is defined as a static member in a class or an interface. Such a nested class can be instantiated like any ordinary top-level class, using its full name. No enclosing instance is required to instantiate a static member class. Note that there are no non-static member,

local, or anonymous interfaces. Interfaces are always defined either at the top level or as static members.

Non-static member classes are defined as instance members of other classes, just like fields and instance methods are defined in a class. An instance of a non-static member class always has an enclosing instance associated with it.

Local classes can be defined in the context of a block as in a method body or a local block, just as local variables can be defined in a method body or a local block.

Anonymous classes can be defined as expressions and instantiated on the fly. An instance of a local (or an anonymous) class has an enclosing instance associated with it, if the local (or anonymous) class is declared in a non-static context.

A nested class or interface cannot have the same name as any of its enclosing classes or interfaces

2.11 Inheritance:

Inheritance is required because we need two things.
1 Extensibility
2 Substitutability.
3 Reusability.

Inheritance.

1. Inheritance in English means you should get something from the other person.
2 . In terms of Programming , It is not about only getting something from the other person but also getting something exclusive to you. Meaning of you in object oriented terminology is sub class object.

in inheritance, we should never talk about parent and child, because then there is never ending story.

```
class A
class B extends A    // we tell B is inheriting from A
class C extends B    // C is inheriting from B
class D extends C    // D is inheriting from C
```

A is a parent (stupid word)
B is a child class (stupid word)
why stupid, what is relationship between C and A.

A is the base class for B, C and D. and not only for B.
A is the super class for B, C and D. and not only for B.

is D a subclass of C, B and A, answer is Yes
is D a derived of C, B and A, answer is Yes

What is Extensibility

1. when we add a new feature into the super class, without making any changes in the Sub class ,the sub class object is able to access the new feature added in the super class.

Example
One day, customer comes tells in Employee class you need to add deptid, we will add deptid into employee class, provide the necessary functions. Then without making any changes in the SE class, SE object, will access the deptid feature of Employee this is called extensibility. we can access feature of the super class directly by the subclass object.

when we talk about inheritance the day we forget subclass object, we are dead in inheritance.

Substitutability
Substitute means you don't get what you were actually told you will get. in real life, we like generalizing things. about we have always deal with specializations.

Generalization - super class
Specialization - subclass.

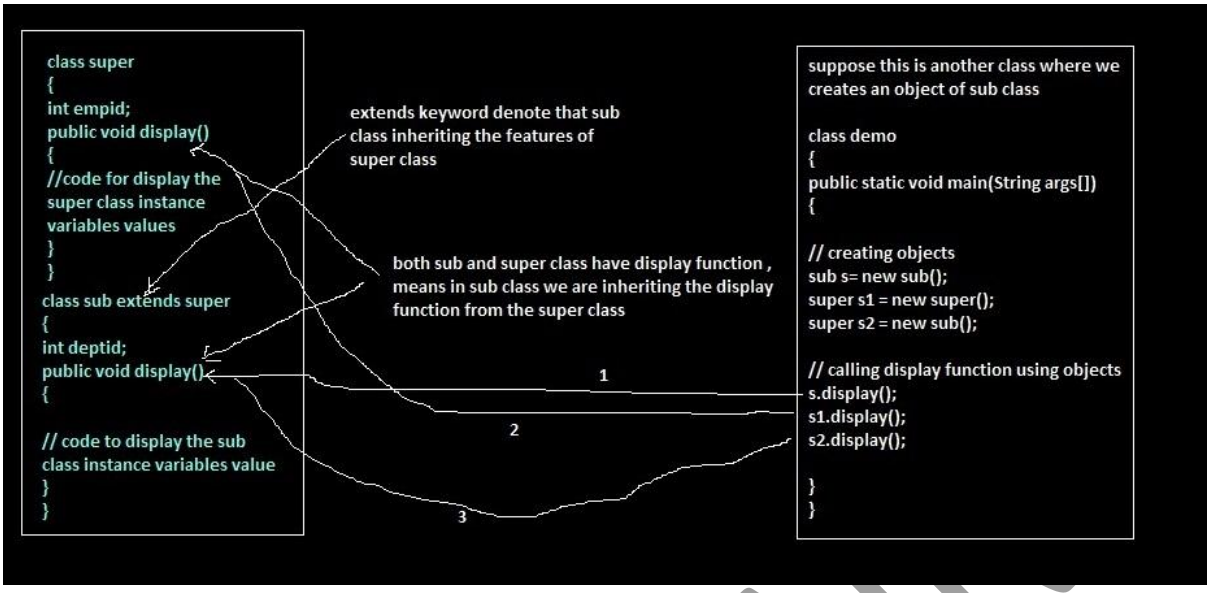
Subclasses can always be generalized through a super class. coding addicts what is substitutability
Base class pointer can be assigned with either a base class object or a derived class object. this is called substitutability, get it without using Inheritance, no you won't get it.

2.12 Method Overriding

if we want to add some functionality in the super class method or if we want that the functionality provided by super class is not what we want in sub class , then we override the method prototype in the sub class and add the additional functionality in that method in the subclass .

So method overriding is a concept through which we can hide the functionality of the super class from the sub class object .

Function calling is dependent on the type of object , not on the type of references. So we create an object of the sub class , and through this object , if we call the overrided method, then its sub class method that is called , not super class method.



In the diagram , arrow 1 shows that the display function of the sub class is called when we access it by declaring object as :

Sub s = new Sub();

In above object creation , object of Sub class is created and this object are handled by the Sub class reference . Class left to the = sign , shows the type of reference and class afer the = shows the type of the object.

Arrow 2 in the diagram shows that display function of super class will call if we create object as :

Super s1 = new Super();

As object is of type super class it calls the super class display function .

Arrow 3 shows that display function of the sub class is called , because the object is of type sub , which is handled by super class reference , thanks to substitutability feature of inheritance , because a sub class object can be assigned either super class reference or sub class reference.

Super s2= new Sub();

2.13 Difference between Method overloading and overriding

	Method Overloading	Method Overriding
1	If we create more than one function with the same name in a class then this concept is called method overloading , but compiler can differentiate through these functions according to a. Number of input arguments b. Type of inputs arguments c. Order of input arguments	If we create a function in sub class that is similar in the super class , then this concept is called method overriding. Compiler is not able to differentiate these functions while calling

2	Method overloading is done within the single class , means if we create two functions with the same name in different class , then it is not method overloading	Method overriding is done in inheritance scenario only .means there should be super – sub relationship between classes.
3	There is no dependency on the return value of the function	Dependency on the return value also , means return value also same in super and sub class methods.
4.	Function overloading is a concept through which we avoid different names for functions that works similarly , but operate on different type of data	Function overriding is a concept through which we can change or add the functionality in the super class method
5.	It is also called compile time polymorphism	It is also called run time polymorphism

2.14 Final with Inheritance :

A **final variable** is a constant, despite being called a variable. Its value cannot be changed once it has been initialized. This applies to instance, static and local variables, including parameters that are declared final.

- A final variable of a primitive data type cannot change its value once it has been initialized.
- A final variable of a reference type cannot change its reference value once it has been initialized, but the state of the object it denotes can still be changed.

These variables are also known as blank final variables. Final static variables are commonly used to define manifest constants (also called named constants), for example Integer.MAX_VALUE, which is the maximum int value. Variables defined in an interface are implicitly final . Note that a final variable need not be initialized at its declaration, but it must be initialized once before it is used.

A **final method** in a class is complete (i.e., has an implementation) and cannot be overridden in any subclass. Subclasses are then restricted in changing the behavior of the method.

Final variables ensure that values cannot be changed, and final methods ensure that behavior cannot be changed.

A **Final class** is a complete class and connot be inherited in other class. So by making a class final , we can restrict the inheritance of that class . For Ex: String class in java.lang package is final class , in our class we can't inherit String class .

Final Variable : Initialized only once before its use

Final Method : cannot be overrided in the subclasses

Final Class : cannot be inherited in other class

2.15 Abstract :

Abstract method in a class means that it is not provided what the function do in class , but the body of method is implemented in the sub class , so declaring a method abstract means that overriding of such method is compulsory.

Abstract Class means that it is necessary to inherit such class , if we forget it , then it is compile time error. Abstract keyword have reverse functionality from final keyword.

Chapter 3 : Packages and Interfaces , String handling

3.1 Packages

1. Packages are used to group related classes, interfaces and sub packages.
2. Packages are used to avoid naming collision.
3. Defining package

   package [package name];

4. Dot operator is used to uniquely identify packages members . Suppose if we declare package as

   package com.practice.demoprograms ;

   then it creates a folder demoprograms which is inside the practice folder , which is inside com folder.

5. If we want to use functions from others packages then we have two ways for using it

   a. **Import that package in our class using import keyword .**

      import [package name which we want to import];

   b. **Use full qualified name for the class**
      Suppose we want to acces Date class from java.util package , then we have to write java.util.Date d = new java.util.Date();
      It will create the object of date class and by this object , we access the Date class functions.

6. All java inbuild packages names start with either java or javax. So it is advisable not to start a package name with the java or javax.

7. Access modifiers plays an important role while importing a function\class from package , means if our class/ function is declared private in other class , then it is not possible to use it . Refer to access modifiers topic for more discussion.

3.2 Concept of CLASSPATH

Where to look? The Java runtime system needs to know where to find programs that you want to run and libraries that are needed. It knows where the predefined Java packages are, but if you are using additional packages, you must tell specify where they are located.

Classpath is environment variable which is used to specify the path where from the java interpreter searches for the file whose name is specified with java keyword.

For example

C:/>set classpath=d:

After writing it press “ENTER”. Now if we write following command.

C:/>java Hello

Now the interpreter will search this file “Hello” in directory D:/

Advantages of Classpath:-

1. Using classpath variable largest path can be written once but can be used multiple times.
2. Using classpath we can clearly specify where from to search the required file in computer.
3. Classpath is much useful in Packages.

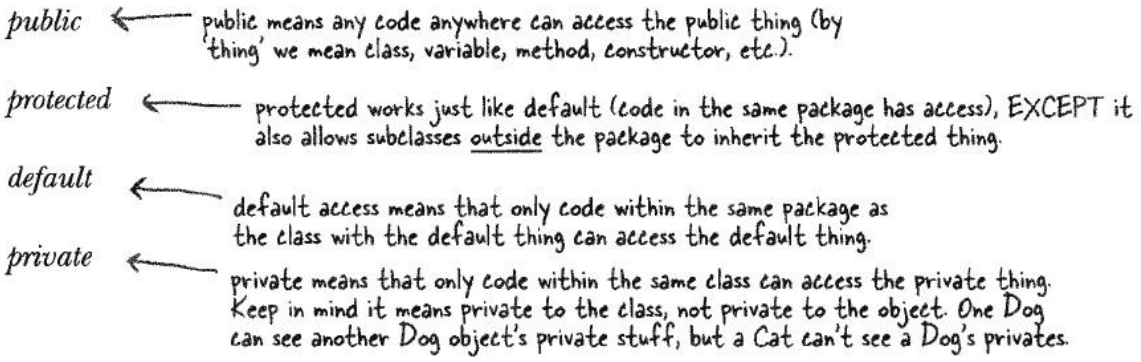
Setting classpath in windows

The CLASSPATH variable can be set on Windows XP with the following steps.

- Click the Start button in the lower left of the screen.
- Select Control Panel from the pop-up menu.
- Choose System from the submenu.
- Click the Advanced tab.
- Click the Environment Variables button near the bottom and you will see two lists of variables.
- Look in the System variables list for a variable named CLASSPATH. If you find it, click Edit. If you don't find it, click New and enter CLASSPATH in the Variable name field.
- The Variable value field is a list of file paths of directories or jar files. The first thing in this list should be the *current directory*, which is represented in windows just as it is in Unix, as a single period. If there's more than one thing in this list, separate items with a semicolon. For example, my CLASSPATH variable starts with these three items (there are a couple more, but this should be enough to give you the idea). The only part you need is the first "dot".

3.3 **Access Modifier or Visibility control** :Java has *four* access *levels* and *three* access *modifiers*. There are only *three* modifiers because the *default* (what you get when you don't use any access modifier) *is* one of the four access levels.

Access Levels (in order of how restrictive they are, from least to most restrictive)



public
Use public for classes, constants (static final variables) , and methods that you're exposing to other code (for example getters and setters) and most constructors.

private
Use private for virtually all instance variables, and for methods that you don't want outside code to call (in other words, methods *used* by the public methods of your class). But although you might not use the other two (protected and default), you still need to know what they do because you'll see them in other code.

default
Both protected and default access levels are tied to packages. Default access is simple-it means that only code *within the samepackage* can access code with default access. So a default class, for example can be accessed by only classes within the same package as the default class.

protected
Protected access is almost identical to default access, with one exception: it allows subclasses to *inherit* the protected thing, *even if those subclasses are outside the package of the super-class they extend:* That's it. That's *all* protected buys you-the ability to let your subclasses be outside your superclass package, yet still *inherit* pieces of the class, including methods and constructors.

3.4 Using other package in class :

There are two ways to used different classes from different package in our class .

a. **By using the fully qualified name** : Using fully qualified name is a tedious thing to do if we want to used a class from different package many times . In fully qualified name , before using the class from

different package , we have to mention the name of the package before the class and each subfolder is differentiate through dot(.) operator . For Example , if we want to use JOptionPane class in our class , then we have to write like :

String s = javax.swing.JOptionPane.showInputDialog("enter the string value");

this is tedious activity to used package because writing such a big package name always makes our code more errorous and more typing effort we have to put while coding .

b. **Import the required package** : class from different Package which we want to used in our class , requires to import the package in our class . By importing a required class , we can used such class without its fully qualified name . There are two formats for importing classes

- for importing whole package

  import javax.swing.*;
- for importing specific class

  import javax.swing.JOptionPane;

Note : import statements should be the after package declaration statement , or if there is no package declaration statements then it should comes before class declaration .

3.5 Interfaces

Extending classes using single implementation inheritance creates new class types. A super class reference can denote objects of its own type and its subclasses strictly according to the inheritance hierarchy. Because this relationship is linear, it rules out multiple implementation inheritance, that is, a subclass inheriting from more than one super class. Instead Java provides interfaces, which not only allow new named reference types to be introduced, but also permit multiple interface inheritance.

Defining Interfaces

A top-level interface has the following general syntax:

```
<accessibility modifier> interface <interface name>
                        <extends interface clause> // Interface header
{ // Interface body
    <constant declarations>
    <method prototype declarations>
    <nested class declarations>
```

```
    <nested interface declarations>
}
```

In the interface header, the name of the interface is preceded by the keyword interface. In addition, the interface header can specify the following information:

- scope or accessibility modifier
- any interfaces it extends

The interface body can contain member declarations which comprise

- constant declarations
- method prototype declarations

An interface does not provide any implementation and is, therefore, **abstract** by definition. This means that it cannot be instantiated, but classes can implement it by providing implementations for its method prototypes. Declaring an interface abstract is superfluous and seldom done.

The member declarations can appear in any order in the interface body. Since interfaces are meant to be implemented by classes, interface members implicitly have **public** accessibility and the **public modifier is omitted**.

Interfaces with empty bodies are often used as markers to tag classes as having a certain property or behavior. Such interfaces are also called ability interfaces. Java APIs provide several examples of such marker interfaces: **java.lang.Cloneable, java.io.Serializable, java.util.EventListener**.

Method Prototype Declarations

An interface defines a contract by specifying a set of method prototypes, but no implementation. The methods in an interface are all implicitly abstract and public by virtue of their definition. A method prototype has the same syntax as an abstract method . However, only the modifiers abstract and public are allowed, but these are invariably omitted.

<return type> <method name> (<parameter list>) <throws clause>;

Implementing Interfaces

Any class can elect to implement, wholly or partially, zero or more interfaces. A class specifies the interfaces it implements as a comma-separated list of unique interface names in an implements clause in the class header. The interface methods must all have **public accessibility** when implemented in the class (or its subclasses). A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's throws clause, as attempting to do so would amount to altering the interface's contract, which is illegal. The criteria for overriding methods also apply when implementing interface methods.

A class can provide implementations of methods declared in an interface, but it does not reap the benefits of interfaces unless the interface name is explicitly specified in its implements clause.

Extending Interfaces

An interface can extend other interfaces, using the extends clause. Unlike extending classes, an interface can extend several interfaces. The interfaces extended by an interface (directly or indirectly), are called superinterfaces. Conversely, the interface is a subinterface of its superinterfaces. Since interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes, respectively.

A subinterface inherits all methods from its superinterfaces, as their method declarations are all implicitly public. A subinterface can override method prototype declarations from its superinterfaces. Overridden methods are not inherited. Method prototype declarations can also be overloaded, analogous to method overloading in classes.

Constants in Interfaces

An interface can also define named constants. Such constants are defined by field declarations and are considered to be public, static and final. These modifiers are usually omitted from the declaration. Such a constant must be initialized with an initializer expression .

An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly without using the fully qualified name. Such a client inherits the interface constants.

Extending an interface that has constants is analogous to extending a class having static variables. In particular, these constants can be hidden by the subinterfaces. In the case of multiple inheritance of interface constants, any name conflicts can be resolved using fully qualified names for the constants involved.

Example 6.10 Variables in Interfaces

```
interface Constants {

    double PI_APPROXIMATION = 3.14;
    String AREA_UNITS = " sq.cm.";
    String LENGTH_UNITS = " cm.";
}

public class Client implements Constants {
    public static void main(String[] args) {
```

```
        double radius = 1.5;
        System.out.println("Area of circle is " +
                           (PI_APPROXIMATION*radius*radius) +
                           AREA_UNITS);            // (1) Direct access.
        System.out.println("Circumference of circle is " +
                           (2*Constants.PI_APPROXIMATION*radius) +
                           Constants.LENGTH_UNITS); // (2) Fully qualified name.
    }
}
```

Output from the program:

```
Area of circle is 7.0649999999999995 sq.cm.
Circumference of circle is 9.42 cm.
```

3.6 String Handling

The String class is defined in the **java.lang package** and hence is implicitly available to all the programs in Java. The String class is declared as **final**, which means that it cannot be subclassed. It extends the Object class and implements the **Serializable**, **Comparable**, and **CharSequence** interfaces.

Java implements strings as objects of type String. A string is a sequence of characters. Unlike most of the other languages, Java treats a string as a single value rather than as an array of characters.

The String objects are **immutable**, i.e., once an object of the String class is created, the string it contains cannot be changed. In other words, once a String object is created, the characters that comprise the string cannot be changed. Whenever any operation is performed on a String object, a new String object will be created while the original contents of the object will remain unchanged. However, at any time, a variable declared as a String reference can be changed to point to some other String object.

3.7 Constructors defined in the String class

The String class defines several constructors. The most common constructor of the String class is the one given below:

```
public String(String value)
```

This constructor constructs a new String object initialized with the same sequence of the characters passed as the argument. In other words, the newly created String object is the copy of the string passed as an argument to the constructor.

Other constructors defined in the String class are as follows:

public String()

This constructor creates an empty String object. However, the use of this constructor is unnecessary because String objects are immutable.

public String(char[] value)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the array that is passed as the argument to it.

public String(char[] value, int startindex, int len)

This constructor creates a new String object initialized with the same sequence of characters currently contained in the subarray. This subarray is derived from the character array and the two integer values that are passed as arguments to the constructor. The int variable startindex represents the index value of the starting character of the subarray, and the int variable len represents the number of characters to be used to form the new String object.

public String(StringBuffer sbf)

This constructor creates a new String object that contains the same sequence of characters currently contained in the string buffer argument.

public String(byte[] asciichars)

The array of bytes that is passed as an argument to the constructor contains the ASCII character set. Therefore, this array of bytes is first decoded using the default charset of the platform. Then the constructor creates a new String object initialized with same sequence of characters obtained after decoding the array.

public String(byte[] asciiChars, int startindex, int len)

This constructor creates the String object after decoding the array of bytes and by using the subarray of bytes.

3.8 Special String Operations

Finding the length of string

The String class defines the length() method that determines the length of a string. The length of a string is the number of characters contained in the string. The signature of the length() method is given below:

public int length()

String Concatenation using the + operator

The + operator is used to concatenate two strings, producing a new String object as the result. For example,

```
String sale = "500";

String s = "Our daily sale is" + sale + "dollars";

System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars".

The + operator may also be used to concatenate a string with other data types. For example,

```
int sale = 500;

String s = "Our daily sale is" + sale + "dollars";

System.out.println(s);
```

This code will display the string "Our daily sale is 500 dollars". In this case, the variable sale is declared as int rather than String, but the output produced is the same. This is because the int value
contained in the variable sale is automatically converted to String type, and then the + operator concatenates the two strings.

Character Extraction Functions

The String class provides a number of ways in which characters can be extracted from a String object. Each is examined here. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:

```
char charAt(int where)
```

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location. For example,

```
char ch;
String s=”java solutions”;
ch= s.charAt(3);
assigns the value "a" to ch.
```

getChars( )

If you need to extract more than one character at a time, you can use the getChars( ) method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring. The following program demonstrates getChars( ):

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

Here is the output of this program:

```
demo
```

getBytes( )

There is an alternative to getChars( ) that stores the characters in an array of bytes. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
byte[ ] getBytes( )
```

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

String Comparison

The String class defines various methods that are used to compare strings or substrings within strings. Each of them is discussed in the following sections:

equals()

The equals() method is used to check whether the Object that is passed as the argument to the method is equal to the String object that invokes the method. It returns true if and only if the argument is a String object that represents the same sequence of characters as represented by the invoking object. The signature of the equals() method is as follows:

public boolean equals(Object str)

equalsIgnoreCase()

The equalsIgnoreCase() method is used to check the equality of the two String objects without taking into consideration the case of the characters contained in the two strings. It returns true if the two strings are of the same length and if the corresponding characters in the two strings are the same ignoring case. The signature of the equalsIgnoreCase() method is:

public boolean equalsIgnoreCase(Object str)

compareTo()

The compareTo() method is used in conditions where a Programmer wants to sort a list of strings in a predetermined order. The compareTo() method checks whether the string passed as an argument to the method is less than, greater than, or equal to the invoking string. A string is considered less than another string if it comes before it in alphabetical order. The signature of the compareTo() method is as follows:

public int compareTo(String str)

where, str is the String being compared to the invoking String. The compareTo() method returns an int value as the result of String comparison. The meaning of these values are given in the following table:

Value Meaning

Less than zero The invoking string is less than the argument string.

Zero The invoking string and the argument string are same.

Greater than zero The invoking string is greater than the argument string.

The String class also has the compareToIgnoreCase() method that compares two strings without taking into consideration their case difference. The signature of the method is given below:

public int compareToIgnoreCase(String str)

regionMatches()

The regionMatches() method is used to check the equality of two string regions where the two string regions belong to two different strings. The signature of the method is given below:

public boolean regionMatches(int startindex, String str2, int startindex2, int len)

There is also an overloaded version of the method that tests the equality of the substring ignoring the case of characters in the substring. Its signature is given below:

public boolean regionMatches(boolean ignoreCase, int startindex, String str2, int startindex2, int len)

In both signatures of the method, startindex specifies the starting index of the substring within the invoking string. The str2 argument specifies the string to be compared. The startindex2 specifies the starting index of the substring within the string to be compared. The len argument specifies the length of the substring being compared. However, in the latter signature of the method, the comparison is done ignoring the case of the characters in the substring only if the ignoreCase argument is true.

startsWith()

The startsWith() method is used to check whether the invoking string starts with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean startsWith(String prefix)

There is also an overloaded version of the startsWith() method with the following signature:

public boolean startsWith(String prefix, int startindex)

In both signatures of the method given above, the prefix denotes the substring to be matched within the invoking string. However, in the second version, the startindex denotes the starting index into the invoking string at which the search operation will commence.

endsWith()

The endsWith() method is used to check whether the invoking string ends with the same sequence of characters as the substring passed as an argument to the method. The signature of the method is given below:

public boolean endsWith(String prefix)

Modifying a String

The String objects are immutable. Therefore, it is not possible to change the original contents of a string. However, the following String methods can be used to create a new copy of the string with the required modification:

substring()

The substring() method creates a new string that is the substring of the string that invokes the method. The method has two forms:

public String substring(int startindex)

public String substring(int startindex, int endindex)

where, startindex specifies the index at which the substring will begin and endindex specifies the index at which the substring will end. In the first form where the endindex is not present, the substring begins at startindex and runs till the end of the invoking string.

Concat()

The concat() method creates a new string after concatenating the argument string to the end of the invoking string. The signature of the method is given below:

public String concat(String str)

replace()

The replace() method creates a new string after replacing all the occurrences of a particular character in the string with another character. The string that invokes this method remains unchanged. The general form of the method is given below:

public String replace(char old_char, char new_char)

trim()

The trim() method creates a new copy of the string after removing any leading and trailing whitespace. The signature of the method is given below:

public String trim(String str)

toUpperCase()

The toUpperCase() method creates a new copy of a string after converting all the lowercase letters in the invoking string to uppercase. The signature of the method is given below:

public String toUpperCase()

toLowerCase()

The toLowerCase() method creates a new copy of a string after converting all the uppercase letters in the invoking string to lowercase. The signature of the method is given below:

public String toLowerCase()

split()

this method can split the string into the given format and will return the string in the form of string array. Actually, it is based on regex expression with some characters which have a special meaning in a regex expression.

Searching Strings

The String class defines two methods that facilitate in searching a particular character or sequence of characters in a string. They are as follows:

IndexOf()

The indexOf() method searches for the first occurrence of a character or a substring in the invoking string. If a match is found, then the method returns the index at which the character or the substring first appears. Otherwise, it returns -1. The indexOf() method has the following signatures:

public int indexOf(int ch)

public int indexOf(int ch, int startindex)

public int indexOf(String str)

public int indexOf(String str, int startindex)

lastIndexOf()

3.9 String Buffer

A string buffer implements a mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are **synchronized** where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

String buffers are used by the compiler to implement the binary string concatenation operator +. For example, the code:

```
x = "a" + 4 + "c"
```

is compiled to the equivalent of:

```
x = new StringBuffer().append("a").append(4).append("c")
                      .toString()
```

which creates a new string buffer (initially empty), appends the string representation of each operand to the string buffer in turn, and then converts the contents of the string buffer to a string. Overall, this avoids creating many temporary strings.

The principal operations on a StringBuffer are
the append and insert methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer.
The append method always adds these characters at the end of the buffer; the insert method adds the characters at a specified point.

For example, if z refers to a string buffer object whose current contents are "start", then the method call z.append("le") would cause the string buffer to contain "startle", whereas z.insert(4, "le") would alter the string buffer to contain "starlet".

In general, if sb refers to an instance of a StringBuffer,
then sb.append(x) has the same effect as sb.insert(sb.length(), x).

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

3.10 Vector

Vectors are dynamic arrays which can extend or shrink when objects are added in the vector. Vector class is part of java.util package. Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program. Vector comes under the collection of java. Vectors are synchronized objects.

The Vector class supports four constructors. The first form creates a default vector, which has an initial size of 10:

```
Vector( )
```

The second form creates a vector whose initial capacity is specified by size:

```
Vector(int size)
```

The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward:

```
Vector(int size, int incr)
```

The fourth form creates a vector that contains the elements of collection c:

```
Vector(Collection c)
```

Adding element in vector :

void add(int index, Object element)

Inserts the specified element at the specified position in this Vector. If Index value is not in the range of already added objects then it throws IndexOutOfBoundException.

boolean add(Object o)

Appends the specified element to the end of this Vector.

boolean addAll(Collection c)

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

boolean addAll(int index, Collection c)

Inserts all of the elements in in the specified Collection into this Vector at the specified position.

void addElement(Object obj)

Adds the specified component to the end of this vector, increasing its size by one.

Removing element from vector

boolean removeAll(Collection c)

Removes from this Vector all of its elements that are contained in the specified Collection.

void removeAllElements()

Removes all components from this vector and sets its size to zero.

boolean removeElement(Object obj)

Removes the first (lowest-indexed) occurrence of the argument from this vector.

void removeElementAt(int index)

removeElementAt(int index)

protected void removeRange(int fromIndex, int toIndex)

Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.

boolean remove(Object o)

Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.

Get Object from vector

Object get(int index)

Returns the element at the specified position in this Vector.
int indexOf(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
int indexOf(Object elem, int index) Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.

3.11 Wrapper Classes

Java is an object-oriented language and as said everything in java is an object. But what about the primitives? They are sort of left out in the world of objects, that is, they cannot participate in the object activities, such as being returned from a method as an object, and being added to a Collection of objects, etc. . As a solution to this problem, Java allows you to include the primitives in the family of objects by using what are called **wrapper classes**.

There is a wrapper class for every primitive date type in Java. This class encapsulates a single value for the primitive data type. For instance the wrapper class for int is Integer, for float is Float, and so on. Remember that the primitive name is simply the lowercase name of the wrapper except for char, which maps to Character, and int, which maps to Integer.

The wrapper classes in the Java API serve two primary purposes:

- To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.
- To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

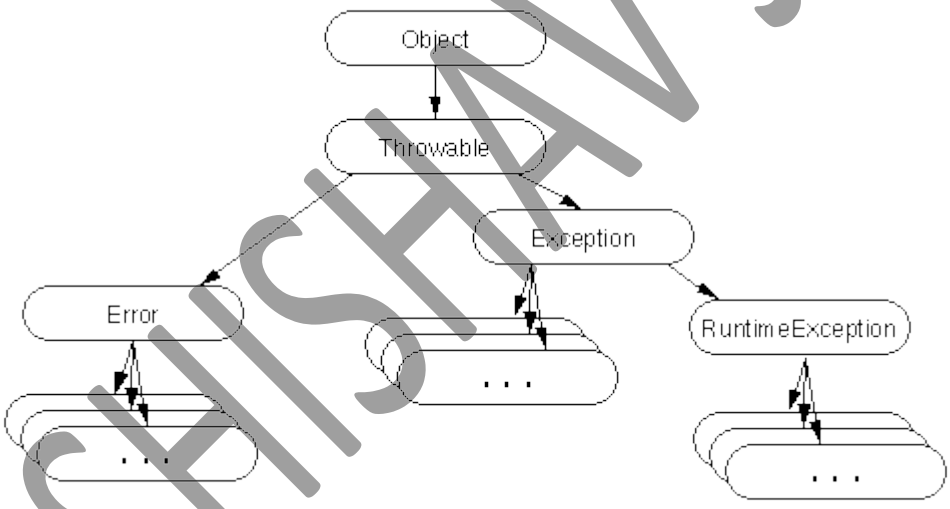
Chapter 4 : Exception Handling and File Handling

4.1 Exceptions :

An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution. For example, a requested file cannot be found, or an array index is out of bounds, or a network link failed. Explicit checks in the code for such conditions can easily result in incomprehensible code. Java provides an exception handling mechanism for systematically dealing with such error conditions.
The exception mechanism is built around the throw-and-catch paradigm. To throw an exception is to signal that an unexpected error condition has occurred. To catch an exception is to take appropriate action to deal with the exception. An exception is caught by an exception handler. The throw-and-catch principle is embedded in the trycatch- finally construct.

4.2 Exception Hierarchy

Exceptions in Java are objects. All exceptions are derived from the java.lang. Throwable class. The two main subclasses Exception and Error constitute the main categories of throwables, the term used to refer to both exceptions and errors.



4.3 Exception Handling :

Exception handling through which we can separate the normal flow of code from error rectification code . Exceptions are the run time errors in java . There are three types of errors in programming.

1. Compile time error : if we have any syntax error , then this errors are rectified during compilation of code , and your code is aborted until all the compile time errors are removed . for example , if during the code designing , instead of while(condition) , we make a spell mistake for while , then it is compile time error.

2. Run Time Error : If error occurs while running your application code , then it is called run time error , all the exception are run time errors .

3. Logical Error : if we have some logic problem then such types of errors are called logical errors.

There are two types of Exceptions :

1. System Defined Exception
2. User Defined Exception

1. System defined Exceptions : System defined exceptions are those exceptions whose occurance are already known by the jvm means such type of exception occurrence is already defined in pre build java classes . All the inbuild exceptions are classes in java which are subclasses of the exception class , which is subclass of the Throwable class .

In system defined exceptions

1. System knows that something is an Exception
2. It also knows when it is happening or occurring during 2, system creates the object of the exception throws.
3. Only thing it does not know what needs to be done. so we mention only 3, forget 1 and 2.

4.4 Try, catch and finally

The mechanism for handling exceptions is embedded in the try-catch-finally construct, which has the following general form:

```
try {

// try block <statements>

} catch (<exception type1> <parameter1>) {

// catch block <statements>
}
...
catch (<exception typen> <parametern>)
{

 // catch block <statements>

} finally {

 // finally block<statements>
}
```

Exceptions thrown during execution of the try block can be caught and handled in a catch block. A finally block is guaranteed to be executed, regardless of the cause of exit from the try block, or whether any catch block was executed.

A few aspects about the syntax of this construct should be noted. The block notation is mandatory. For each try block there can be zero or more catch blocks, but only one finally block. The catch blocks and finally block must always appear in conjunction with a try block, and in the above order. A try block must be followed by either at least one catch block or one finally block. Each catch block defines an exception handler. The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the Throwable class or one of its subclasses.

Each block (try, catch, or finally) of a try-catch-finally construct can contain arbitrary code, which means that a try-catch-finally construct can also be nested in any such block. However, such nesting can easily make the code difficult to read and is best avoided.

try Block

The try block establishes a context that wants its termination to be handled. Termination occurs as a result of encountering an exception, or from successful execution of the code in the try block.

For all exits from the try block, except those due to exceptions, the catch blocks are skipped and control is transferred to the finally block, if one is specified . For all exits from the try block resulting from exceptions, control is transferred to the catch blocks—if any such blocks are specified—to find a matching catch block .If no catch block matches the thrown exception, control is transferred to the finally block, if one is specified .

catch Block

Only an exit from a try block resulting from an exception can transfer control to a catch block. A catch block can only catch the thrown exception if the exception is assignable to the parameter in the catch block .The code of the first such catch block is executed and all other catch blocks are ignored.

After a catch block has been executed, control is always transferred to the finally block, if one is specified. This is always true as long as there is a finally block, regardless of whether the catch block itself throws an exception.

In Example the method printAverage() calls the method computeAverage() in a try catch construct at (4). The catch block is declared to catch exceptions of type ArithmeticException. The catch block handles

the exception by printing the stack trace and some additional information at (7) and (8), respectively.

```
public class Average2 {

   public static void main(String[] args) {
      printAverage(100, 0);                          // (1)
      System.out.println("Exit main().");            // (2)
   }

   public static void printAverage(int totalSum, int totalNumber) {
      try {                                          // (3)
         int average = computeAverage(totalSum, totalNumber);  // (4)
         System.out.println("Average = " +                     // (5)
            totalSum + " / " + totalNumber + " = " + average);
      } catch (ArithmeticException ae) {             // (6)
         ae.printStackTrace();                       // (7)
         System.out.println("Exception handled in " +
                        "printAverage().");          // (8)
      }
      System.out.println("Exit printAverage().");    // (9)
   }

   public static int computeAverage(int sum, int number) {
      System.out.println("Computing average.");      // (10)
      return sum/number;                             // (11)
   }
}
```

Output from the program, with call printAverage(100, 20) at (1):

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Output from the program, with call printAverage(100, 0) at (1):

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:24)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

Important note :

1. We can mention more then one catch for a try block . each catch can handle exception .
2. We can mention super class reference in the catch block . Super reference can handle any exception of its type or of its sub class type.
3. Any line of code in try block, if we get exception , then execution of try is aborted at that place and control moves to the catch block.
4. If exception thrown by try is not matched with any reference in catch block , then program will terminate due to exception in your code.

finally Block

If the try block is executed, then the finally block is guaranteed to be executed, regardless of whether any catch block was executed. Since the finally block is always executed before control transfers to its final destination, it can be used to specify any clean-up code (for example, to free resources such as, files, net connections).

A try-finally construct can be used to control the interplay between two actions that must be executed in the right order, possibly with other intervening actions.

```
int sum = -1;
try {
sum = sumNumbers();

// other actions
} finally {
if (sum >= 0) calculateAverage();
}
```

The code above guarantees that if the try block is entered sumNumbers() will be executed first and then later calculateAverage() will be executed in the finally block, regardless of how execution proceeds in the try block. As the operation in calculateAverage() is dependent on the success of sumNumbers(), this is checked by the value of the sum variable before calling calculateAverage(). catch blocks can, of course, be included to handle any exceptions.

If the finally block neither throws an exception nor executes a control transfer statement like a return or a labeled break, then the execution of the try block or any catch block determines how execution proceeds after the finally block.If there is no exception thrown during execution of the try block or the exception has been handled in a catch block, then normal execution continues after the finally block.

4.5 Throw Statements

System defined exception is thrown implicitly during execution. A program can explicitly throw an exception using the **throw** statement. The general format of the throw statement is as follows:

throw <object reference expression>;

The compiler ensures that the <object reference expression> is of type Throwable class or one of its subclasses. At runtime a NullPointerException is thrown if the <object reference expression> is null. This ensure that a Throwable will always be propagated. A detail message is often passed to the constructor when the exception object is created.

throw new ArithmeticException("Integer division by 0");

When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a catch block that can handle the exception. The search starts in the context of the current try block, propagating to any enclosing try blocks and through the runtime stack to find a handler for the exception. Any associated finally block of a try block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level. If a handler is found, execution resumes with the code in its catch block.

```
public class Average7 {

   public static void main(String[] args) {
      try {                                                    // (1)
         printAverage(100, 0);                                 // (2)
      } catch (ArithmeticException ae) {                       // (3)
         ae.printStackTrace();                                 // (4)
         System.out.println("Exception handled in " +          // (5)
                            "main().");
      } finally {
         System.out.println("Finally in main().");             // (6)
      }
      System.out.println("Exit main().");                      // (7)
   }

   public static void printAverage(int totalSum, int totalNumber) {
      try {                                                    // (8)
         int average = computeAverage(totalSum, totalNumber);  // (9)
         System.out.println("Average = " +                     // (10)
            totalSum + " / " + totalNumber + " = " + average);
      } catch (IllegalArgumentException iae) {                 // (11)
         iae.printStackTrace();                                // (12)
         System.out.println("Exception handled in " +          // (13)
                            "printAverage().");
      } finally {
         System.out.println("Finally in printAverage().");     // (14)
```

```
        }
        System.out.println("Exit printAverage().");                    // (15)
    }

    public static int computeAverage(int sum, int number) {
        System.out.println("Computing average.");
        if (number == 0)                                                // (16)
            throw new ArithmeticException("Integer division by 0");// (17)
        return sum/number;                                              // (18)
    }
}
```

Output from the program:

```
Computing average.
Finally in printAverage().
java.lang.ArithmeticException: Integer division by 0
        at Average7.computeAverage(Average7.java:35)
        at Average7.printAverage(Average7.java:19)
        at Average7.main(Average7.java:6)
Exception handled in main().
Finally in main().
Exit main().
```

4.6 throws Clause

A throws clause can be specified in the method protoype.

```
... someMethod(...)
    throws <ExceptionType1>, <ExceptionType2>,..., <ExceptionTypen> { ... }
```

Each <ExceptionType$_i$> declares a checked exception (Exception that are not subclasses of RunTimeException class). The compiler enforces that the checked exceptions thrown by a method are limited to those specified in its throws clause. Of course, the method can throw exceptions that are subclasses of the checked exceptions in the throws clause. This is permissable since exceptions are objects, and a subclass object can polymorphically act as an object of its superclass. The throws clause can have unchecked exceptions specified, but this is seldom used and the compiler does not check them.

Any method that can cause a checked exception to be thrown, either directly by using the throw statement or indirectly by invoking other methods that can throw such an exception, must deal with the exception in one of three ways. It can

- use a try block and catch the exception in a handler and deal with it

- use a try block and catch the exception in a handler, but throw another exception that is either unchecked or declared in its throws clause
- explicitly allow propagation of the exception to its caller by declaring it in the throws clause of its method protoype

throws Clause

```
public class Average8 {
   public static void main(String[] args) {
      try {                                          // (1)
         printAverage(100, 0);                       // (2)
      } catch (IntegerDivisionByZero idbze) {        // (3)
         idbze.printStackTrace();
         System.out.println("Exception handled in " +
                            "main().");
      } finally {                                    // (4)
         System.out.println("Finally done in main().");
      }

      System.out.println("Exit main().");            // (5)
    }

   public static void printAverage(int totalSum, int totalNumber)
         throws IntegerDivisionByZero {              // (6)

      int average = computeAverage(totalSum, totalNumber);
      System.out.println("Average = " +
         totalSum + " / " + totalNumber + " = " + average);
      System.out.println("Exit printAverage().");    // (7)
   }

   public static int computeAverage(int sum, int number)
         throws IntegerDivisionByZero {              // (8)

      System.out.println("Computing average.");
      if (number == 0)                               // (9)
         throw new IntegerDivisionByZero("Integer Division By Zero");
      return sum/number;                             // (10)
   }
}

class IntegerDivisionByZero extends Exception {      // (11)
   IntegerDivisionByZero(String str) { super(str); } // (12)
}
```

Output from the program:

Computing average.
IntegerDivisionByZero: Integer Division By Zero
at Average8.computeAverage(Average8.java:33)
at Average8.printAverage(Average8.java:22)
at Average8.main(Average8.java:7)
Exception handled in main().
Finally done in main().
Exit main().

The exception type specified in the throws clause in the method protoype can be a superclass type of the actual exceptions thrown, that is, the exceptions thrown must be assignable to the type of the exceptions specified in the throws clause. If a method can throw exceptions of the type A, B, and C where these are subclasses of type D, then the throws clause can either specify A, B, and C or just specify D. In the printAverage() method, the method protoype could specify the superclass Exception of the subclass IntegerDivisionByZero in a throws clause.

```
public static void printAverage(int totalSum, int totalNumber)
    throws Exception { /* ... */ }
```

It is generally a bad programming style to specify exception superclasses in the throws clause of the method protoype, when the actual exceptions thrown in the method are instances of their subclasses. Programmers will be deprived of information about which specific subclass exceptions can be thrown, unless they have access to the source code.

Important Points About Throws:

1. Don't know whether to put try and catch or throws then better put throws.

2. Throw is used to indicate the person who is using out function that some trouble has occurred in our function, the person who is using it, can deal with it programmatically.

3. In a single function we can put try and catch also and also throws.

4. What is most important thing a person who is using only try and catch when they are supposed throws, this will create a lot of mess

Difference between final, finally and finalize

final:

It is the modifier applicable for classes methods and variables. For final classes we can't create child classes i.e inheritance is not possible.

final() methods can't be override in child classes for final variables reassignments is not possible because they are constants.

finally:

It is a block associated with try catch the main objective of finally block is to maintain cleanup code which should execute always.

finalize:

It is a method should be executed by the "*Garbage Collector*" just before destroying an object. The main objective of finalize method is to maintain cleanup code.

Note:

when compared with finalize, finally is always recommended to maintain cleanup code because there is no guarantee for the exact behavior of "*Garbage Collector*" it is Virtual Machine Dependent.

Difference between throw and throws keyword

throw and throws are two Java keyword related to Exception feature of Java programming language. If you are writing Java program and familiar with What is Exception in Java, its good chance that you are aware of What is throw and throws in Java. In this Session we will compare throw vs throws and see some worth noting difference between throw and throws in Java. Exception handling is an important part of Java programming language which enables you to write robust programs. There are five keywords related to Exception handling in Java e.g. try, catch, finally, throw and throws.

1) throw keyword is used to throw Exception from any method or static block in Java while throws keyword, used in method declaration, denoted which Exception can possible be thrown by this method.

2) If any method throws checked Exception as shown in below Example, than caller can either handle this exception by catching it or can re throw it by declaring another throws clause in method declaration.

```
public void read() throws IOException{
   throw new IOException();
}
```

failure to either catch or declaring throws in method signature will result in

compile time error.

3) throw keyword can be used in switch case in Java but throws keyword can not be used anywhere except on method declaration line.

4) As per Rules of overriding in Java, overriding method can not throw Checked Exception higher in hierarchy than overridden method . This is rules for throws clause while overriding method in Java.

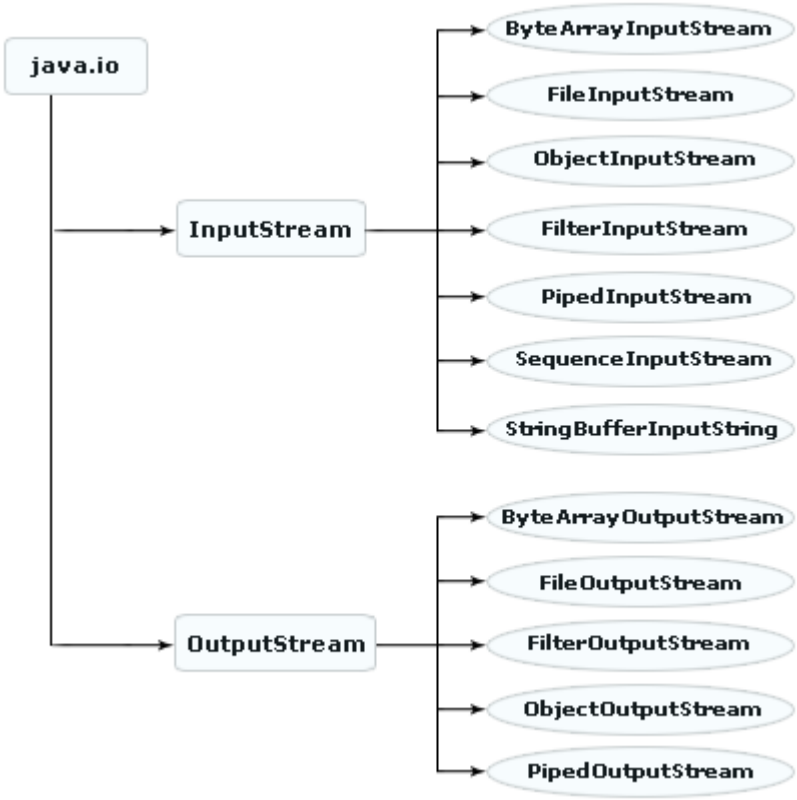
5) throw transfers control to caller, while throws is suggest for information and compiler checking.

6) Both Checked and Unchecked Exception can be declared to be thrown using throws clause in Java.

4.7 Java I/O

The Java Input/Output (I/O) is a part of **java.io** package.
The **java.io** package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The **InputStream** and**OutputStream** are central classes in the package which are used for reading from and writing to byte streams, respectively.

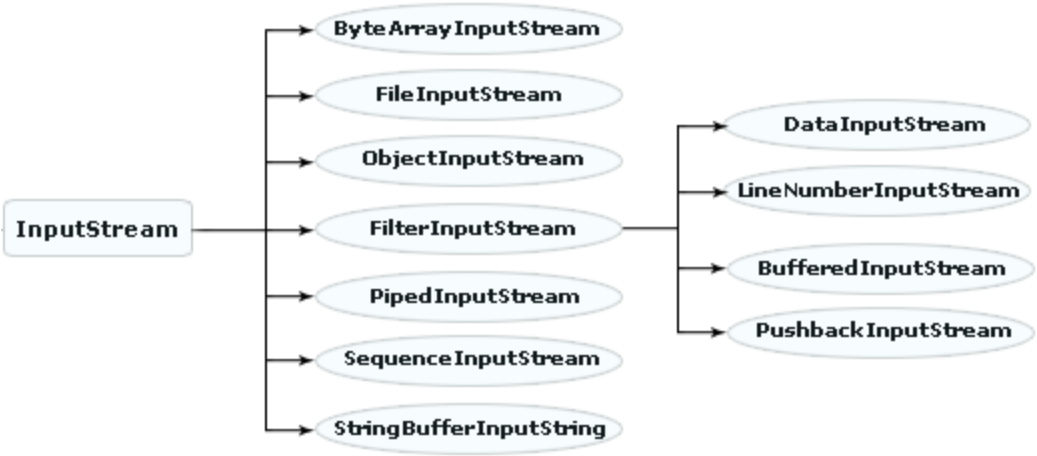
The java.io package can be categories along with its stream classes in a hierarchy structure shown below:



InputStream:

The **InputStream** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a **file**, a **string**, or **memory** that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when you create it. You cans explicitly close a stream with the **close( )**method, or let it be closed implicitly when the object is found as a garbage.

The subclasses inherited from the **InputStream** class can be seen in a hierarchy manner shown below:

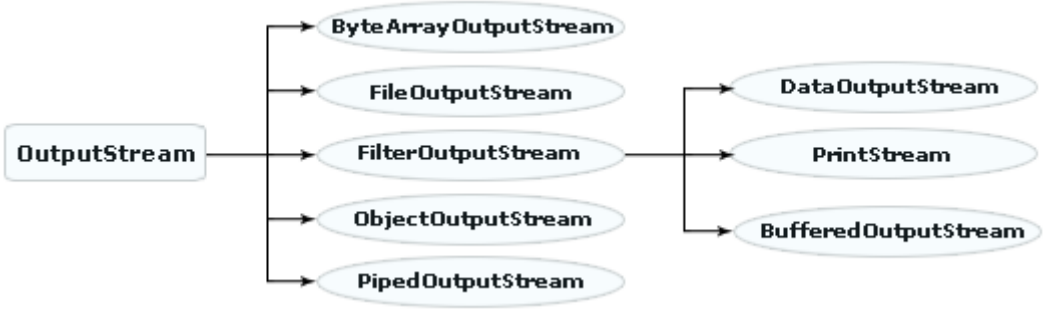


InputStream is inherited from the Object class. Each class of the InputStreams provided by the java.io package is intended for a different purpose.

OutputStream:

The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the **close( )** method, or let it be closed implicitly when the object is garbage collected.

The classes inherited from the **OutputStream** class can be seen in a hierarchy structure shown below:

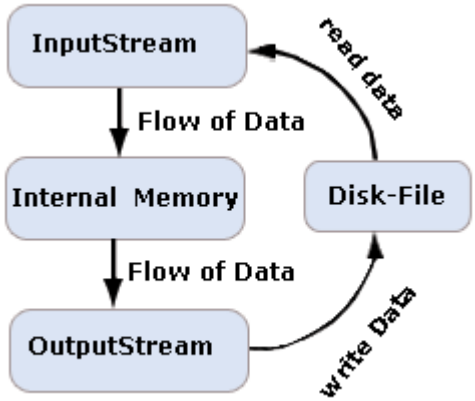


OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.

How Files and Streams Work:

Java uses **streams** to handle I/O operations through which the data is flowed from one location to another. For example, an **InputStream** can flow the data from a disk file to the internal memory and an**OutputStream** can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file. When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream.

The working process of the I/O streams can be shown in the given diagram.



4.8 Java file handling

One of the most frequently used task in programming is writing to and reading from a file. To do this in Java there are more possibilities. At this time only the most frequently used text file handling solutions will be presented.

Java is one of the most popular programming languages that people use. It is so popular because it can be used so widely from application software to web applications. As a language it is class-based, concurrent, general purpose and object-orientated. This allows it to have as few implementation dependencies as possible.

Filename handling

To write anything to a file first of all we need a file name we want to use. The file name is a simple string like like this:

- *String fileName = "test.txt";*

If you want to write in a file which is located elsewhere you need to define the
complete file name and path in your fileName variable:

- *String fileName = "c:\\filedemo\\test.txt";*

However if you define a path in your file name then you have to take care the path separator. On windows system the '\' is used and you need to backslash it so you need to write '\\', in Unix,Linux systems
the separator is a simple slash '/'.

To make your code OS independent you can get the separator character as follows:

- *String separator = File.separator;*

Open a file

To open a file for writing use the FileWriter class and create an instance from it.
The file name is passed in the constructor like this:

- *FileWriter writer = new FileWriter(fileName);*

This code opens the file in overwrite mode. If you want to append to the file then
you need to use an other constructor like this:

- *FileWriter writer = new FileWriter(fileName,true);*

Besides this the constructor can throw an IOException so we put all of the code inside
a try-catch block.

Write to file

At this point we have a writer object and we can send real content to the file. You can do this using the **write()** method, which has more variant but the most commonly used requires a string as input parameter.

Calling the **write()** method doesn't mean that it immediately writes the data into the file. The output is maybe cached so if you want to send your data immediately to the file you need to call the **flush()** method.

As last step you should close the file with the **close()** method and you are done.

The basic write method looks like this:

Code:

```
1.  public void writeFile() {
2.      String fileName = "c:\\test.txt";
3.
4.      try {
5.          FileWriter writer = new FileWriter(fileName,true);
6.          writer.write("Test text.");
7.          writer.close();
8.      } catch (IOException e) {
9.          e.printStackTrace();
10.     }
11. }
12.
```

However in a real world situation the FileWriter usually not used directly. Instead of FileWriter the BufferedWriter or from Java 1.5 the PrintWriter are used. These writer objects gives you more flexibility to handle your IO. Here is a simple BufferedWriter example:

Code:

```
1.  public void writeFile() {
2.      String fileName = "c:\\test.txt";
3.
4.      try {
5.          BufferedWriter writer = new BufferedWriter(new
    FileWriter(fileName,true));
6.          writer.write("Test text.");
7.          writer.newLine();
8.          writer.write("Line2");
9.          writer.close();
10.     } catch (IOException e) {
11.         e.printStackTrace();
12.     }
13. }
```

Step 2 - Reading from a file

If everything is alright, you have successfully written some basic text into a file. Now it's time to read the file content back. Not surprisingly reading from a file is very similar to writing. We only need to use *Reader objects instead of *Writer objects. It means that you can use FileReader or BufferedReader. As a simple FileReader can handle only a single character or a character array it is more convenient to use the BufferedReader which can read a complete line from a file as a string. So using a BufferedReader we can read

a text file line by line with the **readln()** method as you can see in this example:

Code:

```
1.  public String readFile() {
2.      String fileName = "c:\\test.txt";
3.
4.      StringBuffer fileContent = new StringBuffer();
5.
6.      try {
7.          FileReader fr = new FileReader(fileName);
8.          BufferedReader reader = new BufferedReader(new
    FileReader(fileName));
9.
10.             String line;
11.             while ((line = reader.readLine()) != null) {
12.                 fileContent.append(line).append(LS);
13.             }
14.
15.     } catch (IOException e) {
16.         e.printStackTrace();
17.     }
18.
19.     return fileContent.toString();
20. }
```

Copy Paste Program

```
import java.io.*;
class copypaste
{
public static void main(String args[])
{
FileReader fr= null;
BufferedReader br= null;
FileWriter fw= null;
BufferedWriter bw= null;

try
{
fr= new FileReader("arr.txt");
br= new BufferedReader(fr);
fw= new FileWriter("b.txt");
```

```
bw= new BufferedWriter(fw);

String text=null;

while((text=br.readLine())!=null)
{
bw.write(text);
bw.newLine();

}

System.out.println("pasting done");


}catch(FileNotFoundException e)
{
System.out.println("file name is not valid");
}catch(IOException e)
{
System.out.println("some probem occurs");
}
finally
{
bw.close();
br.close();

}
}
}
```

Summary for Java I/O

1. Java I/O operations can done in two modes , character based mode and binary based mode
2. In Binary based , we can't read file contents and binary recognizes data types.
3. Want to store objects in File , then our class must implements Serializable interface.
4. Serailizable is an empty interface (also called marker and tag interface) , means it don't provide any methods , but it is indicate to JVM that object of class is serialize in files.
5. During serialization , object instance is stored in file with information about the class , so when we deserialize it , it can be restored as a class attributes on heap.

Chapter Summary :

1. Exception means some unwanted event occurs while execution (during run time) which we can handle .
2. Exception Handling is a way to separate normal flow of code from error rectification code.
3. All the exceptions in java are subclass of Exception class , which is subclass of Throwable .(see Exception hierarchy)
4. Exception can be handled by using try,catah , finally mechanism or by using throws keyword.
5. Normal flow of code is put in the try block ,if any exception occurs in try block, then execution moves to catch block.
6. There can be more then one catch block , each catch block is used to handled specific type error.
7. In catch block , we can mention the reference of exception type or its superclass reference.
8. Finally is a piece of code which is guaranteed to execute regardless exception occurs or not.
9. Throws indicates that the function can have the exception and all these exceptions are handled by the function which calls this function.
10. Throws keyword is used in the function prototype and if there are more then one exception , then it is separated by comma
11. Generally throws is used to throw Checked exeception(which is not subclass of rum time exception) ,
12. Throw keyword is used to throw the exception from try block to catch block .
13. If the exception is not handled in the function where it is raised, then exception is swanned to the function from where this function is called , if no function handled the exception , then finally JVM stops the exection of your code due to exception.

Chapter 5 : Concurrency and Applet

5.1 Multitasking :

Multitasking allows us to perform more than one activities concurrently.

We can acheive multitasking by using

- Multiprocessing
- MultiThreading

At the coarse grain , there is MultiProcessing (process based multitasking) which allows process to run concurrently on the computer , where each process has its own memory , and in a single processor enviourment , switching between processes requires the current running process to save its state in its process table , this switching is called context switching . So Multi processing exists between two application .

At the fine grain level , there is multithreading which allows part of the same program to run concurrently on the computer , a separate path of execution is used to run different parts of the same application , these separate path of execution in process is called threads which are independent of each others . For Example , in a word processor , during printing of some pages , printing and formatting are done simuntenouly.

Java supports multithreading .

5.2 Threads :

Threads is a path of execution within a program that shares code and data among other threads , that's why threads are called light weighted process .In java , there are two types of threads

- Daemon threads or system defined threads
- User defined threads

Threads which is created by the system is called daemon threads which are used to create user defined threads means all the user defined threads are spawned from Daemon threads . whenever all the user defined threads are executing in a program , your program didn't terminate .

We can set the state of threads by defining it as daemon or user defined threads , setDaemon(Boolean value) is used to change the status of the thread.

5.3 Creating Threads

Threads can be created using

- Implements the java.lang.Runnable interface
- Extending the java.lang.Thread class

Implementing the Runnable interface :

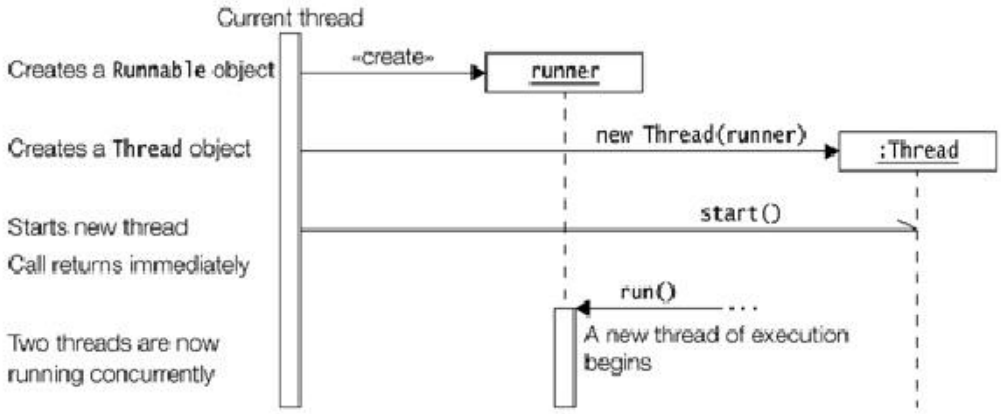
Runable interface have the following specification.

```
public interface Runable
{
public void run();
}
```

Procedure for creating threads based on the runnable interface is as follows:

1. A class implements the Runnable interface provides the run() method which will be executed by the threads.
2. An Object of the thread class is created. An object of a class implementing the RUnnable interface is passed as an argument to a constructor of the thread class.
3. The start() method is invoked on the thread object created in the previous step.

Creation of thread using Runnable interface



The following is a summary of important constructors and methods from the java.lang.Thread class:

Thread(Runnable threadTarget)

Thread(Runnable threadTarget, String threadName)

The argument threadTarget is the object whose run() method will be executed when the thread is started. The argument threadName can be specified to give an explicit name for the thread, rather than an automatically generated one. A thread's name can be retrieved by using the getName() method.

static Thread currentThread()

This method returns a reference to the Thread object of the currently executing thread.

final String getName()
final void setName(String name)

The first method returns the name of the thread. The second one sets the thread's name to the argument.

Implementing the Runnable Interface Example

```
class Counter implements Runnable {
// variables declaration
  private int currentValue;

  private Thread worker;

// constructor of the counter class

 public Counter(String threadName) {
    currentValue = 0;
    worker = new Thread(this, threadName);   // (1) Create a new thread.
    System.out.println(worker);
    worker.start();                          // (2) Start the thread.
   }

          public int getValue()
          {
          return currentValue;
          }

   public void run() {                       // (3) Thread entry point
      try {
         while (currentValue < 5) {
            System.out.println(worker.getName() + ": " + (currentValue++));
            Thread.sleep(250);              // (4) Current thread sleeps.
         }
      } catch (InterruptedException e) {
         System.out.println(worker.getName() + " interrupted.");
```

```
        }
        System.out.println("Exit from thread: " + worker.getName());
    }
}

public class Client {
    public static void main(String[] args) {
        Counter counterA = new Counter("Counter A"); // (5) Create a thread.

        try {
            int val;
            do {
                val = counterA.getValue();      // (6) Access the counter value.
                System.out.println("Counter value read by main thread: " + val);
                Thread.sleep(1000);             // (7) Current thread sleeps.
            } while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
        }

        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

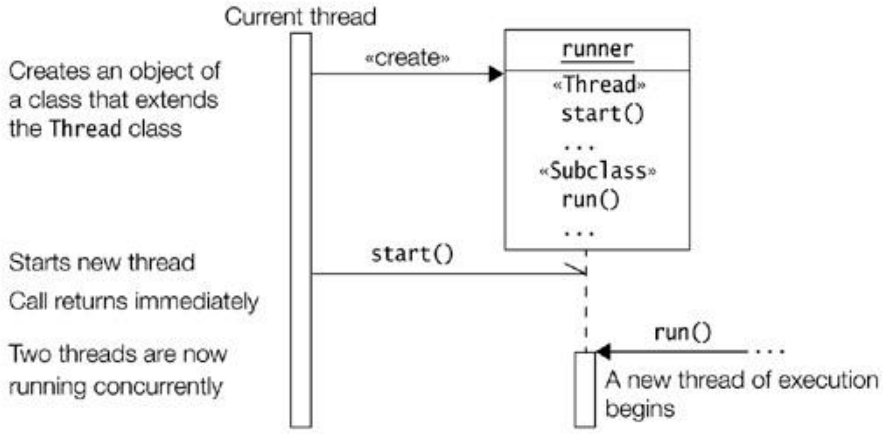
```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

The Client class uses the Counter class. It creates an object of class Counter at (5) and retrieves its value in a loop at (6). After each retrieval, it sleeps for 1,000 milliseconds at (7), allowing other threads to run.

Extending the Thread Class

A class can also extend the Thread class to create a thread. A typical procedure for doing this is as follows :

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.



the Counter class from Example has been modified to illustrate extending the Thread class. Note the call to the constructor of the superclass Thread at (1) and the invocation of the inherited start() method at (2) in the constructor of the Counter class. The program output shows that the Client class creates two threads and exits, but the program continues running until the child threads have completed. The two child threads are independent, each having its own counter and executing its own run() method.

Extending the Thread Class

```
class Counter extends Thread {

   private int currentValue;

   public Counter(String threadName) {
      super(threadName);                  // (1) Initialize thread.
      currentValue = 0;
      System.out.println(this);
      start();                            // (2) Start this thread.
   }

   public int getValue() { return currentValue; }

public void run() {                       // (3) Override from superclass.
      try {
         while (currentValue < 5)
```

```
                {
            System.out.println(getName() + ": " + (currentValue++));
            Thread.sleep(250);              // (4) Current thread sleeps.
          }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + getName());
    }
}

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

        System.out.println("Method main() runs in thread " +
            Thread.currentThread().getName());  // (5) Current thread

                    Counter counterA = new Counter("Counter A"); // (6) Create a
thread.
                    Counter counterB = new Counter("Counter B"); // (7) Create a
thread.

        System.out.println("Exit from main() method.");
    }
}

Possible output from the program:

Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

5.4 Synchronization :

Threads shares the same memory and resources . However, there are situations where it is desirable that only one thread at a time to access the shared resources. Foe Example, creditng and debiting a shared bank account concurrently amongst several users without proper discipline will endanger the integrity of data , so in that scenario , we are forcing that only one thread have access the shared resources . This synchronization can be achieved by using

1. Synchronized Methods
2. Synchronized Blocks

Synchronized Methods :

If the method of an object should be executed by one thread at a time , then the definitions of such methods should be modified with the keyword *synchronied*.

While a thread is in the synchronized method of an object , all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait. This restriction doesn't apply to the thread that already in the synchronized method of the object and such a method can invoke other synchronized methods of the object without being blocked.

Synchronized blocks

Whereas synchronized Methods of a class are synchronized on the monitor of an object of the class , the synchronized blocks allows the arbitrary code to be synchronizes on the monitor of an arbitrary objects. The general form of synchronized blocks are

```
Synchronized (<Object reference>)
{
// code block
}
```

5.5 Thread States

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because a thread's start()

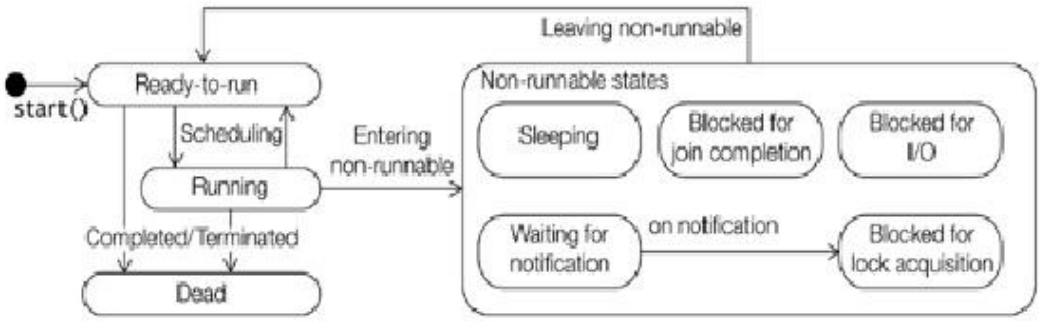
method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed.

Figure shows the states and the transitions in the life cycle of a thread.

- *Ready-to-run state* : A thread starts life in the Ready-to-run state
- *Running state* : If a thread is in the Running state, it means that the thread is currently executing
- *Dead state* : Once in this state, the thread cannot ever run again
- *Non-runnable states :* A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.

  The non-runnable states can be characterized as follows:

  - Sleeping: The thread sleeps for a specified amount of time .
  - Blocked for I/O: The thread waits for a blocking operation to complete.
  - Blocked for join completion: The thread awaits completion of another thread Waiting for notification: The thread awaits notification from another thread).
  - Blocked for lock acquisition: The thread waits to acquire the lock of an object



Various methods from the Thread class are presented next. Examples of their usage are presented in subsequent sections.

final boolean isAlive()

This method can be used to find out if a thread is alive or dead. A thread is alive if it has been started but not yet terminated, that is, it is not in the Dead state.

final int getPriority()
final void setPriority(int newPriority)

The first method returns the priority of the current thread. The second method changes its priority. The priority set will be the minimum of the two values: the specified newPriority and the maximum priority permitted for this thread.

static void yield()

This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute.

static void sleep (long millisec) throws InterruptedException

The current thread sleeps for the specified time before it takes its turn at running again.

final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively.

void interrupt()

The method interrupts the thread on which it is invoked. In the Waiting-for-notification, Sleeping, or Blocked-for-join-completion states, the thread will receive an InterruptedException.

5.6 Thread Priorities

Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Priorities are integer values from 1 (lowest priority given by the constant Thread. MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).

A thread inherits the priority of its parent thread. Priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class. The following code sets the

priority of the thread myThread to the minimum of two values: maximum priority and current priority incremented to the next level:

```
myThread.setPriority(Math.min(Thread.MAX_PRIORITY,
myThread.getPriority()+1));
```

Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.

  If a thread with a higher priority than the current running thread moves to the Ready-to-run state, then the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.

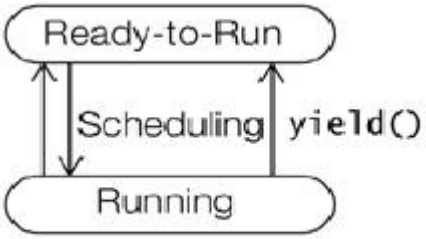
- Time-Sliced or Round-Robin scheduling.

  A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

It should be pointed out that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Running and Yielding

After its start() method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, that is, it waits for its turn to get CPU time. The thread scheduler decides which thread gets to run and for how long. Figure illustrates the transitions between the Ready-to-Run and Running states. A call to the static method yield(), defined in the Thread class, will cause the current thread in the Running state to transit to the Ready-to-run state, this relinquishing the CPU. The thread is

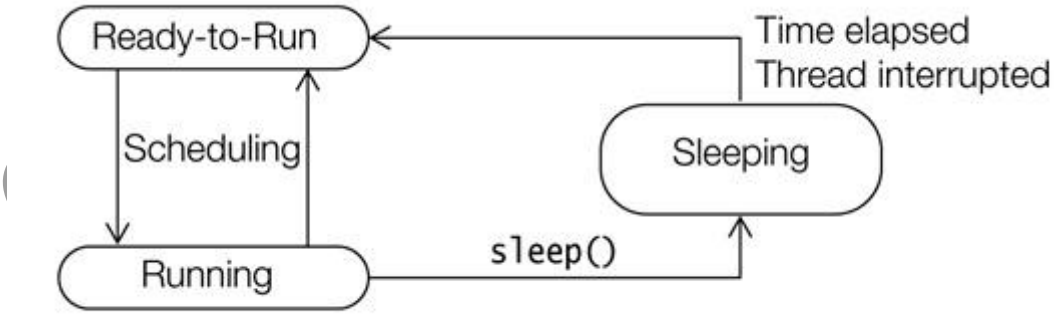


then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute.

By calling the static method yield(), the running thread gives other threads in the Ready-to-run state a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of canceling it by clicking on a Cancel button. If the computation thread hogs the CPU and the user clicks the Cancel button, chances are that it might take a while before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following run() method:

```
public void run() {
    try {
        while (!done()) {
            doLittleBitMore();
            Thread.yield();          // Current thread yields
        }
    } catch (InterruptedException e) {
        doCleaningUp();
    }
}
```

Sleeping and Waking up



A call to the static method sleep() in the Thread class will cause the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before transitioning to the Ready-to-run state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an InterruptedException when it awakes and gets to execute.

There are serveral overloaded versions of the sleep() method in the Thread class.

Waiting and Notifying

Waiting and notifying provide means of communication between threads that synchronize on the same object. The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose. These final methods are defined in the Object class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException.

final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException

A thread invokes the wait() method on the object whose lock it holds. The thread is added to the wait set of the object.

final void notify()
final void notifyAll()

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object

5.7 Applets : An Applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page. When you use a Java technology-enabled browser to view a page that contains an Applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM). In java , there are two types of applet.

- The first are those based directly on the applet class , use the Abstract window toolkit components for graphical user interface
- The second type of applets care based on the swing class called JApplet . Swing applets uses the swing components for graphic user interface .

JApplets inherits from Applets , All the features of Applet are also available in JApplet.

5.8 Difference between remote and local applet:

Local Applet are written in local system and stored in the file structure of the local system , when this applets are included in the java enabled

browser , there is no requirement for the internet connectivity . Browser search for the Applet in the local system path as defined in the HTML page code.

Remote Applets are stored on remote system , whenever browser is requested for Remote Applets , then it queries for the applet from remote system , so internet connectivity is an essential while executing the remote Applets. For Remote Applets , web page should have an address of the remote system , and to run this remote applet , these are downloaded in local system from the remote path.

5.9 Applet Life Cycle

Applet runs in the browser and its lifecycle method are called by JVM when it is loaded and destroyed. Here are the lifecycle methods of an Applet:

init(): This method is called to initialized an applet

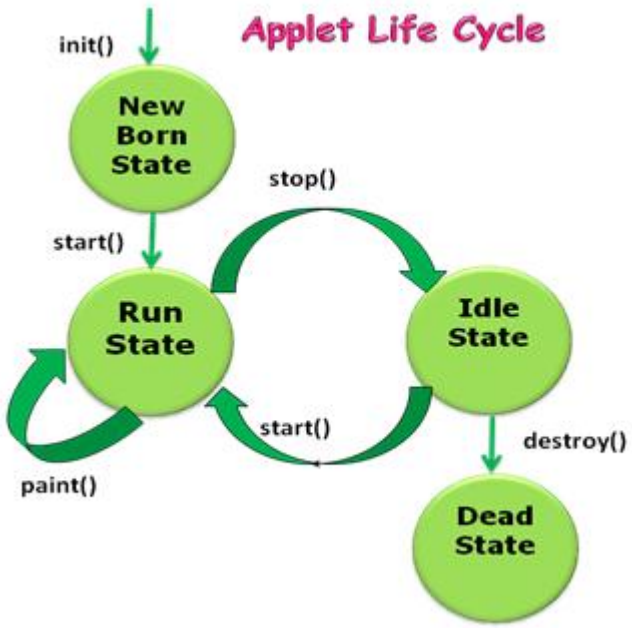
start(): This method is called after the initialization of the applet.

stop(): This method can be called multiple times in the life cycle of an Applet.

destroy(): This method is called only once in the life cycle of the applet when applet is destroyed.

init () method: The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method. The init() method is called only one time in the life cycle on an applet. The init() method is basically called to read the PARAM tag in the html file. The init () method retrieve the passed parameter through the PARAM tag of html file using get Parameter() method All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method .After the initialization of the init() method user can interact with the Applet and mostly applet contains the init() method.

Start () method: The start method of an applet is called after the initialization method init(). This method may be called multiples time when the Applet needs to be started or restarted.



For Example if the user wants to return to the Applet, in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.

Stop () method: The stop() method can be called multiple times in the life cycle of applet like the start () method. Or should be called at least one time. There is only miner difference between the start() method and stop () method. For example the stop() method is called by the web browser on that time When the user leaves one applet to go another applet and the start() method is called on that time when the user wants to go back into the first program or Applet.

destroy() method: The destroy() method is called only one time in the life cycle of Applet like init() method. This method is called only on that time when the browser needs to Shut down.

Demo Applet Program

```
import java.applet.Applet;
import java.awt.Graphics;


public class AppletProgram extends Applet{

        String s = "Applet example" + "\n";
```

```
    public void destroy() {

        s= s+ "I am in destroy() function"+ "\n";
    }

    public void init() {

        s= s+ "I am in init() function \n";

    }

    public void start() {

        s= s+ "I am in start() function \n ";
    }

    public void stop() {

        s= s+ "I am in stop() function \n";
    }


    public void paint(Graphics g) {
        g.drawString(s, 50, 50);
    }


}
```

HTML and Applet

When you put an applet on your page you will need to save the applet on your server as well as the HTML page the applet is embedded in. When the page is loaded by a visitor the applet will be loaded and inserted on the page where you embedded it.

Applets have the file extension "class". An example would be "myapplet.class". Some applets consist of more than just one class file, and often other files need to be present for the applet to run (such as JPG or GIF images used by the applet). Make sure to check the documentation for the applet to see if you have all files for it to run.
Before embedding an applet on your page you need to upload the required files to your server.

Below is a short example showing how simple it is to embed an applet on a page.

```
<Html>
<Head>
<Title>Java Example</Title>
```

```
</Head>

<Body>
This is my page<br>
Below you see an applet<br>
<br>
<Applet Code="MyApplet.class" width=200 Height=100>
</Applet>
</Body>
</Html>
```

Two HTML tags are relevant according to applets: <Applet> and <Param>.

The <Applet> tag embeds the applet in your HTML page.
The <Param> tag is used to enter parameters for the applet.

The following attributes can be set for the <Applet> tag:

Attribute	Explanation	Example
Code	Name of class file	Code="myapplet.class"
Width=n	n=Width of applet	Width=200
Height=n	n=Height of applet	Height=100
Codebase	Library where the applet is stored. If the applet is in same directory as your page this can be omitted.	Codebase="applets/"
Alt="Text"	Text that will be shown in browsers where the ability to show applets has been turned off.	alt="Menu Applet"
Name=Name	Assigning a name to an applet can be used when applets should communicate with each other.	Name="starter"
Align= Left Right Top Texttop Middle Absmiddle Baseline Bottom Absbottom	Justifies the applet according to the text and images surrounding it. A full explanation of the individual parameters is given **here**.	Align=Right
Vspace=n	Space over and under the applet.	Vspace=20

Hspace=n	Space to the left and right of applet.	Hspace=40

The <Param> tag has the general syntax:

```
<Param Name=NameOfParameter
Value="ValueOfParameter">
```

Each applet has different parameters that should be set.
Typical parameters for an applet would be:

- color used by the applet
- font and font size to be used on text in the applet
- name of an image file to be inserted in the applet

Chapter 6 : Java and Database

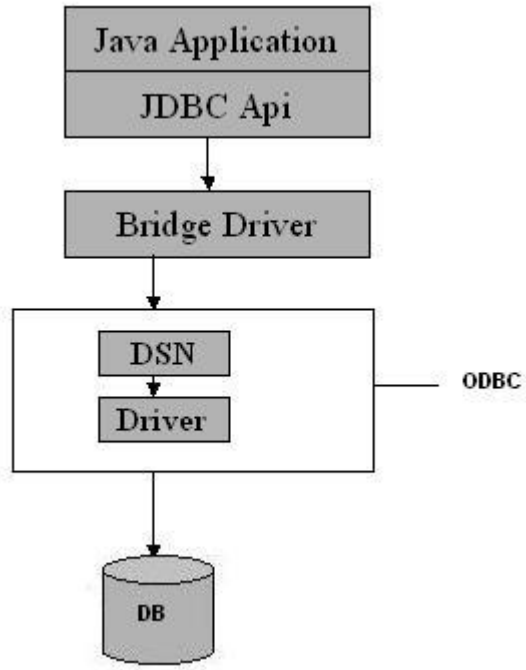
6.1 Drivers : Drivers are used to connect database and the java application.

Request generated by any language is not DBMS specific calls , so it is required to convert these calls according to specific database and vice versa is applicable to process the result. So there are different types of Driver like:

Type 1 JDBC Driver : JDBC-ODBC Bridge driver

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver

is recommended only for experimental use or when no other alternative is available.



Type 1: JDBC-ODBC Bridge

Advantage

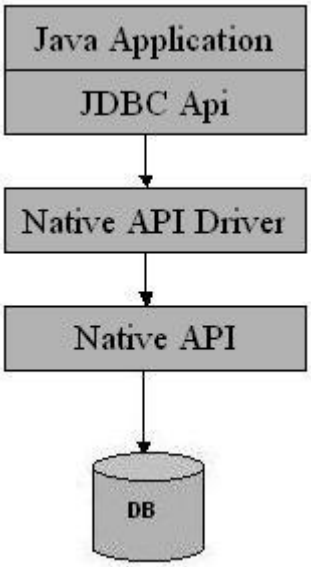
The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Type 2 JDBC Driver : Native-API/partly Java driver

The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.



Type 2: Native api/ Partly Java Driver

Advantage

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type
1 and also it uses Native api which is Database specific.

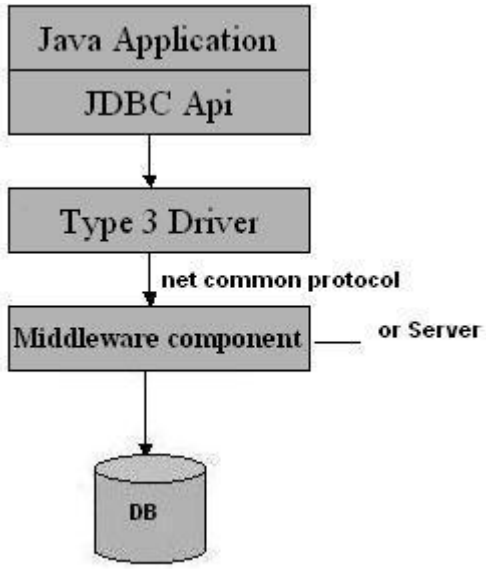
Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

Type 3 JDBC Driver : All Java/Net-protocol driver

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the

middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



Type 3: All Java/ Net-Protocol Driver

Advantage

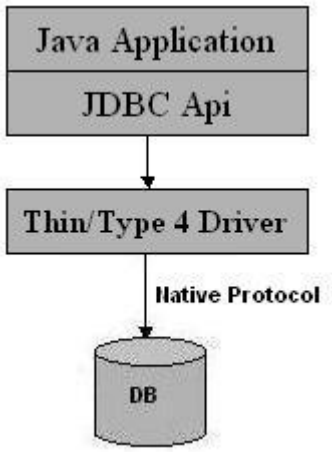
1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

Type 4 JDBC Driver : Native-protocol/all-Java driver

The Type 4 uses java networking libraries to communicate directly with the database server.



Type 4: Native-protocol/all-Java driver

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

6.2 Steps for Connectivity between java program and database

There are seven standard steps in querying databases:

1. Import the required classes
2. Load the JDBC driver.
3. Establish the connection.
4. Create a statement object.
5. Execute a query or update.
6. Process the results.
7. Close the connection.

1. **Importing required classes** : The very first step that is required while connecting java code to database is that we have to import the required class for connection . As an example , for connection mysql , the required classes are present inside java.sql package .

Import the java.sql package

Ex: import java.sql.* ;

* indicates all classes of sql package are imported.

2. **Load the JDBC driver**: Once you import the required classes , next step is to load the driver for connectivity to database. Java provides a class with the name “Class” which is used to load the driver by using its static method forName . So we load the driver by calling Class.forName() method.

```
try {
Class.forName( "com.mysql.jdbc.Driver" );
}
catch (Exception e)
{
System.out.println( “Unable to establish the connection” );
}
```

• A client can connect to Database server by JDBC driver. Connector/j driver is commonly used driver for mysql database.

• The newInstance() method is used to create new instance of driver class.

3. **Establish the connection**

• Driver Manager Class defines object which is used to create connection between database and application.

• The getConnection() method uses user name, password and URL to establish connection.

• With this connection we can execute SQL, PL/SQL statement.

Java provides Connection interface which hold the connection reference for executing different queries .

Connection ref= DriverManger.getConnection(“url”,”username”,”password”);

4. **Create a statement object**.

• Once establish the connection with database server we can interact with database using statement object.

• Statement object is instantiated from connection object by calling createStatement() method.

• Statement object is to send and execute sql statements to database.

Three kinds of statements objects are available

• Statement – create simple SQL statement without parameters.

Interface provides basic functionality for executing and retrieving results.

• Prepared statement – which inherits from Statement, Object used to executes precompiled sql statements with or without parameter. This interface adds method to deal with IN parameter.

5. **Executing statement object**

Statement interface defines method to interact with database by executing sql quires.

• Statement has three method

- executeQuery() is used for SELECT statement.
- executeUpdate() is used for create, alter or drop table.
- execute() is used when sql statement written as string object.

```
String query = "SELECT col1, col2, col3 FROM sometable";
ResultSet resultSet = statement.executeQuery(query);
```

6. **Process the result :**

The simplest way to handle the results is to process them one row at a time, using the ResultSet's next method to move through the table a row at a time. Within a row, ResultSet provides various getXxx methods that take a column index or column name as an argument and return the result as a variety of different Java types. For instance, use getInt if the value should be an integer, getString for a String, and so on for most other data types. If you just want to display the results, you can use getString regardless of the actual column type. However, if you use the version that takes a column index, note that columns are indexed starting at 1 (following the SQL convention), not at 0 as with arrays, vectors, and most other data structures in the Java programming language.

Note that the first column in a ResultSet row has index 1, not 0. Here is an example that prints the values of the first three columns in all rows of a ResultSet.

```
while(resultSet.next()) {
      System.out.println(results.getString(1) + " " +
      results.getString(2) + " " +
      results.getString(3));
}
```

7.Close the Connection

To close the connection explicitly, you should do:

```
connection.close();
```

6.3 Select Query Program

This session is used to for selection the particular rows according to DBMS select query .

```
// Import the required Drivers

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SelectQuery {

	public static void main(String[] args) {
		// Create the reference of Connection Interface
		Connection con =null;
	try {

// load the drivers, com.mysql.jdbc.Driver is DriverClass name

		Class.forName("com.mysql.jdbc.Driver");

	// jdbc:mysql://ip of database machine:port number/databasename

String url= "jdbc:mysql://127.0.0.1:3306/shopping";

	// Establish the connection

con = DriverManager.getConnection(url,"root","shishav");
String query="select * from login_table where username=?";

	// Pass the query

PreparedStatement ps = con.prepareStatement(query);
ps.setString(1, "adi");

//Execute the select query using ExecuteQuery() method
```

```
ResultSet rs = ps.executeQuery();

// Process the result

if(rs.next())
{
        // fetch the rows until the empty row , rs.next() returns false if row
        //don't have any data , else it returns true

        do
        {
        // USERNAME and PASS are the column names in table of Database
        // we can also used the number of column starting from 1.

                String username= rs.getString("USERNAME");
                String pass= rs.getString("PASS");

        // print the username and password in console window

                System.out.println("username is :" + username   + ": password
is :" + pass + "\n");

        }while(rs.next());
                        }
else
{
        System.out.println("no row selected");
}

        }  catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }finally
        {
                try {
//Close the connection
                        con.close();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }


}
```

}

6.4 Insert Query Program

This session is used for learning the insertion of data from java program to dbms table.

```
// Import the required Drivers

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SelectQuery {

	public static void main(String[] args) {
		// Create the reference of Connection Interface
		Connection con =null;
	try {
	// load the drivers, com.mysql.jdbc.Driver is DriverClass name

		Class.forName("com.mysql.jdbc.Driver");

	// jdbc:mysql://ip of database machine:port number/databasename

		String url= "jdbc:mysql://127.0.0.1:3306/shopping";

	// Establish the connection

con = DriverManager.getConnection(url,"root","shishav");
String query="insert into [table name] [columns name comma separated]
values (values for the columns in comma separated ) ";

	// Pass the query

PreparedStatement ps = con.prepareStatement(query);
// if we want to insert the values according to some class attributes or some
//variable , then we can add ? in the query and fill the ? like this , 1st ? in
// the query have 1 number as shown below
// insert into emp(empid) values(?);
// above query have single ? . so its value is filled by adi as shown in below
//line

ps.setString(1, "adi");
```

```
//Execute the select query using ExecuteUpdate() method

int rs = ps.executeUpdate();
if(rs >0)
{
System.out.println("rows successfully inserted");

}
}
        }  catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }finally
        {
                try {
//Close the connection
                        con.close();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

6.5 Delete Query Program

This session is used for learning the deletion of data from Database table using java program .

```
// Import the required Drivers

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class SelectQuery {

        public static void main(String[] args) {
                // Create the reference of Connection Interface
                Connection con =null;
        try {
        // load the drivers, com.mysql.jdbc.Driver is DriverClass name

                Class.forName("com.mysql.jdbc.Driver");
```

```
        // jdbc:mysql://ip of database machine:port number/databasename

                String url= "jdbc:mysql://127.0.0.1:3306/shopping";

        // Establish the connection

con = DriverManager.getConnection(url,"root","shishav");
String query="delete from [tablename] where [condition] ";

        // Pass the query

PreparedStatement ps = con.prepareStatement(query);
// if we want to insert the values according to some class attributes or some
//variable , then we can add ? in the query and fill the ? like this , 1st ? in
// the query have 1 number as shown below
// delete from emp where empid=?
// above query have single ? . so its value is filled by adi as shown in below
//line

ps.setString(1, "adi");


//Execute the select query using ExecuteUpdate() method

int rs = ps.executeUpdate();
if(rs >0)
{
// in case of deletion , number of rows deleted is returned by
//executeUpdate(),if  value is less then 0, then no deletion
System.out.println("rows successfullydeleted");

}
}
        } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }finally
        {
                try {
//Close the connection
                        con.close();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

6.6 Update Query Program :

Updation in the database table is similar like insertion and deletion , only the query syntax is changed according to requirement . In case of updation , query will be

Update [tablename] set columnname=[value which we want] where [condition];

Summary :

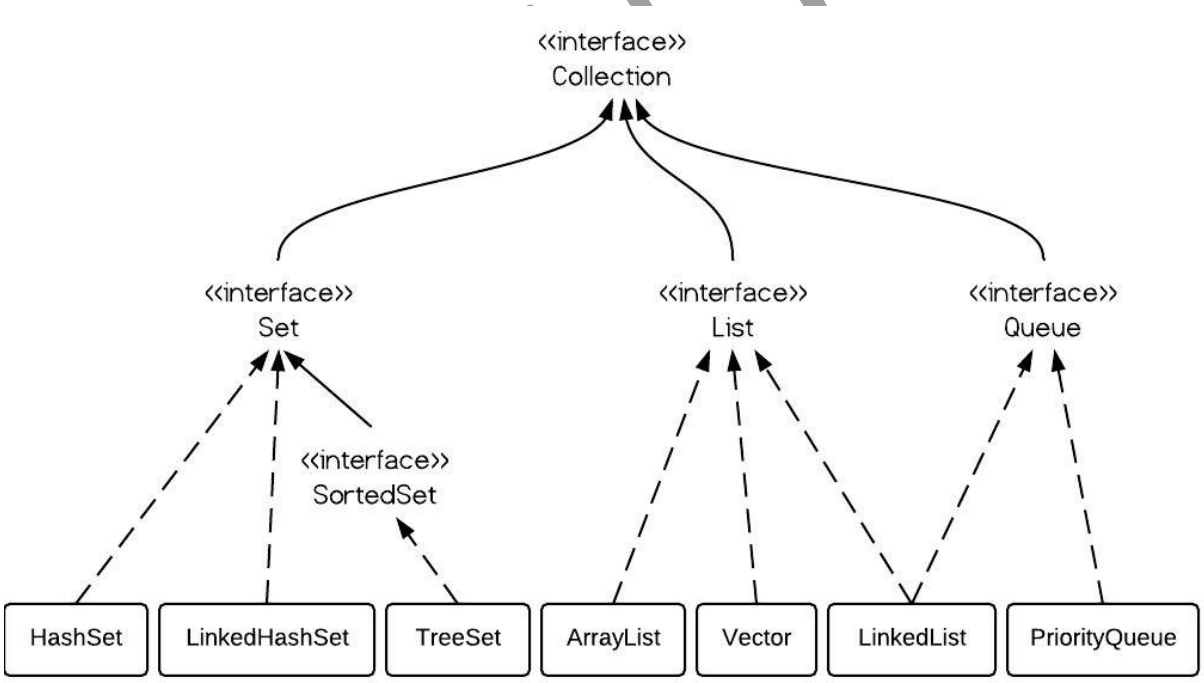
1. We require Drivers to convert the Java specific calls to DBMS specific calls .
2. Class.forName() ,static method of the class Class is used to load the driver class. Each database has specific Drivers Class
3. We have to pass the username ,password and ip and port while establishing a connection with database.
4. We used executeUpdate() method for insertion,deletion and updation query . This method returns the int value which is the number of rows that are affected from query.
5. We used executeQuery() method for select query , which returns the table accoding to our condition in the query and this result is hold by the ResultSet Interface.

Chapter 7: Collections

Collection is used to create group of object that can be treated like a single unit. Objects can be stored , manipulated or deleted in the group of objects , means collection is used as a data structure on which we can perform different data operations like adding an object , retrieval and deletion etc.

The collections framework presents as a set of standard utility classes for managing such collections. This framework is provided in the java.util package and comprises three main parts

- The ***core interfaces*** that allow collections to be manipulated independently of their implementation .These interfaces define the common functionality exhibited by collections and facilitate data exchange between collections.
- A small set of ***implementations*** , that are specific implementations of the core interfaces, providing data structures that a program can use readily.
- An assortment of static utility methods that can be used to perform various operations on collections, such as sorting and searching, or creating customized collections



Core Interfaces

The Collection interface is a generalized interface for maintaining collections, and is the top of the interface inheritance hierarchy for collections shown in above Figure . These interfaces are summarized in below Table .

Table . Core Interfaces in the Collections Framework

Interface	Description	Concrete Classes
Collection	A basic interface that defines the normal operations that allow a collection of objects to be maintained or handled as a single unit.	
Set	The Set interface extends the Collection interface to represent its mathematical namesake: a set of unique elements.	HashSet LinkedHashSet
SortedSet	The SortedSet interface extends the Set interface to provide the required functionality for maintaining a set in which the elements are stored in some sorted order.	TreeSet
List	The List interface extends the Collection interface to maintain a sequence of elements that need not be unique.	ArrayList Vector LinkedList
Map	A basic interface that defines operations for maintaining mappings of keys to values.	HashMap Hashtable LinkedHashMap
SortedMap	Extends the Map interface for maps that maintain their mappings sorted in key order.	TreeMap

The elements in a Set must be unique, that is, no two elements in the set can be equal. The order of elements in a List is retained, and individual elements can be accessed according to their position in the list.

The Map interface does not extend the Collection interface because conceptually, a map is not a collection. A map does not contain elements. It contains mappings (also called entries) from a set of key objects to a set of value objects. A key can, at most, be associated with one value. As the name implies, the SortedMap interface extends the Map interface to maintain its mappings sorted in key order.

Sets

Unlike other implementations of the Collection interface, implementations of the Set interface do not allow duplicate elements. This also means that a set can contain at most one null value. The Set interface does not define any new methods, and its add() and addAll() methods will not store duplicates. If an element is not currently in the set, two consecutive calls to the add() method to insert the element will first return true, then false. A Set models a mathematical set, that is, it is an unordered collection of distinct objects.

HashSet and LinkedHashSet

The HashSet class implements the Set interface. Since this implementation uses a hash table, it offers near constant-time performance for most operations. A HashSet does not guarantee any ordering of the elements. However, the LinkedHashSet subclass of HashSet guarantees insertion-order. The sorted counterpart is TreeSet, which implements the SortedSet interface and has logarithmic time complexity .

As mentioned earlier, the LinkedHashSet implementation is a subclass of the HashSet class. It works similarly to a HashSet, except for one important detail. Unlike a HashSet, a LinkedHashSet guarantees that the iterator will access the elements in insertion order, that is, in the order in which they were inserted into the LinkedHashSet.

The LinkedHashSet class offers constructors analogous to the ones in the HashSet class. The initial capacity (i.e., the number of buckets in the hash table) and its load factor (i.e., the ratio of number of elements stored to its current capacity) can be tuned when the set is created. The default values for these parameters will under most circumstances provide acceptable performance.

HashSet()

Constructs a new, empty set.

HashSet(Collection c)

Constructs a new set containing the elements in the specified collection. The new set will not contain any duplicates. This offers a convenient way to remove duplicates from a collection.

HashSet(int initialCapacity)

Constructs a new, empty set with the specified initial capacity.

HashSet(int initialCapacity, float loadFactor)

Constructs a new, empty set with the specified initial capacity and the specified load factor.

Lists

Lists are collections that maintain their elements in order, and can contain duplicates. The elements in a list are ordered. Each element, therefore, has a position in the list. A zero-based index can be used to access the element at the position designated by the index value. The position of an element can change as elements are inserted or deleted from the list.

In addition to the operations inherited from the Collection interface, the List interface also defines operations that work specifically on lists: position-based access of the list elements, searching in a list, creation of customized iterators, and operations on parts of a list (called open range-view operations). This additional functionality is provided by the following methods in the List interface:

ArrayList, LinkedList, and Vector

Methods Specified for the List Interface

// Element Access by Index

Object get(int index)

Returns the element at the specified index.

Object set(int index, Object element) Optional

Replaces the element at the specified index with the specified element. It returns the previous element at the specified index.

void add(int index, Object element) Optional

Inserts the specified element at the specified index. If necessary, it shifts the element previously at this index and any subsequent elements one position toward the end of the list. The inherited method add(Object) from the Collection interface will append the specified element to the end of the list.

Object remove(int index) Optional

Deletes and returns the element at the specified index, contracting the list accordingly. The inherited method remove(Object) from the Collection interface will remove the first occurrence of the element from the list.

boolean addAll(int index, Collection c) Optional

Inserts the elements from the specified collection at the specified index, using the iterator of the specified collection. The method returns true if any elements were added.

In a non-empty list, the first element is at index 0 and the last element is at size()-1. As might be expected, all methods throw an IndexOutOfBoundsException if an illegal index is specified.

// Element Search

int indexOf(Object o)
int lastIndexOf(Object o)

These methods respectively return the index of the first and the last occurrence of the element in the list if the element is found; otherwise, the value –1 is returned.

// List Iterators
ListIterator listIterator()
ListIterator listIterator(int index)

The iterator from the first method traverses the elements consecutively, starting with the first element of the list, whereas the iterator from the second method starts traversing the list from the element indicated by the specified index.

```
interface ListIterator extends Iterator {

    boolean hasNext();
    boolean hasPrevious();

    Object next();          // Element after the cursor
    Object previous();      // Element before the cursor

    int nextIndex();        // Index of element after the cursor
    int previousIndex();    // Index of element before the cursor

    void remove();          // Optional
    void set(Object o);     // Optional
    void add(Object o);     // Optional
}
```

The ListIterator interface is a bidirectional iterator for lists. It extends the Iterator interface and allows the list to be traversed in either direction. When traversing lists, it can be helpful to imagine a cursor moving forward or backward between the elements when calls are made to the next() and the previous() method, respectively. The element that the cursor passes over is returned. When the remove() method is called, the element last passed over is removed from the list.

// Open Range-View
List subList(int fromIndex, int toIndex)

This method returns a view of the list, which consists of the sublist of the elements from the index fromIndex to the index toIndex-1. A view allows the range it represents in the underlying list to be manipulated. Any changes in

the view are reflected in the underlying list, and vice versa. Views can be used to perform operations on specific ranges of a list.

ArrayList, LinkedList, and Vector

Three implementations of the List interface are provided in the java.util package: ArrayList, LinkedList, and Vector.

The ArrayList class implements the List interface. The Vector class is a legacy class that has been retrofitted to implement the List interface. The Vector and ArrayList classes are implemented using dynamically resizable arrays, providing fast random access and fast list traversal—very much like using an ordinary array. Unlike the ArrayList class, the Vector class is thread-safe, meaning that concurrent calls to the vector will not compromise its integrity.

The LinkedList implementation uses a doubly-linked list. Insertions and deletions in a doubly-linked list are very efficient—elements are not shifted, as is the case for an array. The LinkedList class provides extra methods that implement operations that add, get, and remove elements at either end of a LinkedList:

void addFirst(Object obj)
void addLast(Object obj)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()

The ArrayList and Vector classes offer comparable performance, but Vector objects suffer a slight performance penalty due to synchronization. Position-based access has constant-time performance for the ArrayList and Vector classes. However, position-based access is in linear time for a LinkedList, owing to traversal in a doubly-linked list. When frequent insertions and deletions occur inside a list, a LinkedList can be worth considering. In most cases, the ArrayList implementation is the over-all best choice for implementing lists

Map :

Map is a two column data structure , advantage of a map if we give the key, it tells us the value.

How to add elements in map

Map m = new HashMap();
m.put(key , value);

put method is used to insert an key-value pair in the array.

m.get(key) : it returns the value for the key , if key is not there , then it returns null value , else it returns the value for the key.

m.remove(key) : this method is used to delete the key value pair if the key passed is present in the Map ,
else it returns null . it returns Object type value.

Smart way to run thru the map

```
Map m =new HashMap();
        m.put("pen",3);
        m.put("pen",4);
        m.put("abc", 1);
        Set s = m.entrySet();
        Iterator i = s.iterator();

        Map.Entry x = null;
        while(i.hasNext())
        {
            x = (Entry) i.next();
            System.out.println(x.getKey() + " -- " + x.getValue());
            }
```

Dump way to run thru the map

```
    Map m =new HashMap();
        m.put("pen",3);
        m.put("pen",4);
        m.put("abc", 1);
        Set s = m.keySet();
        Iterator i = s.iterator();
        // good thing we are running thru the map
        //but what is pathetic about it is
        // we get the key from iterator
        // and then run to the map to get the value
        while(i.hasNext())
        {
            Object key = i.next();
            Object value = m.get(key);
            System.out.println(key + "---" + value);
```

}