**Inheritance**

**Inheritance:-**

Reusability is yet another aspect of OOP paradigm. It is always nice if we could reuse something that already exists rather than creating the same all over again. Java supports this concept. Java classes can be reused in several ways. This is basically done by creating new classes reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is known as the base class or super class or parent class and the new one is called the subclass or derived class or child class.

The inheritance allows subclasses to inherit all the variables and methods of their parent classes inheritance may take different forms:

* Single inheritance(only one super class)
* Multi level inheritance(derive form a derived class)
* Multiple inheritance(several super classes)
* Hybrid inheritance (several super classes one derived class)
* Hierarchical inheritance (one super class many subclasses)

**Note:** java does not directly implement multiple inheritances. However this concept is implemented using a secondary inheritance path in the from of interfaces.

**Defining a subclass:-**

The keyword extends signifies that the properties of the superclassname are extended to the subclassname. The subclass will now contain its own variables and methods as well those of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

Syntax:

class sub-class\_name extends super-class\_name

{

Variables declaration;

Methods declaration;

}

**Single inheritance**

A derived class with one base class is known as single inheritance.In single inheritance only one super class and one subclass

**A**

**B**

Example:

class Person

{

int pno;

String pna;

void showPerson ( )

{

System.out.println ( "\n Person number : "+pno );

System.out.println ( "\n Person name : "+pna );

}

}

class Student extends Person

{

int m1, m2, tot;

void showStudent ( )

{

tot = m1 + m2;

System.out.println ( "\n Marks : "+m1+" "+m2 );

System.out.println ( "\n Total : "+tot );

}

}

class SingleInheritEx1

{

public static void main ( String args [ ] )

{

Student S = new Student ( );

S.pno = 101;

S.pna = "Malli";

S.m1 = 90;

S.m2 = 95;

S.showPerson ( );

S.showStudent ( );

}

}

**Multilevel Inheritance:**

The mechanism of deriving a new class from derived class is called multilevel inheritance. Java supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes.

**B**

**A**

**C**

Example:

class Person

{

int pno;

String pna;

void show ( )

{

System.out.println ( "\n Person Number : "+pno );

System.out.println ( "\n Person Name : "+pna );

}

}

class Student extends Person

{

int m1, m2,tot;

void show ( )

{

super.show ( ); // show ( ) of super class i.e., Person class got invoked

tot = m1+m2;

System.out.println ( "\n Marks : "+m1+" "+m2 );

System.out.println ( "\n Total : "+tot );

}

}

class Result extends Student

{

double avg;

String res;

void show ( )

{

super.show ( ); // show ( ) of super class i.e., Student class got invoked

avg = (double)tot/2;

if ( avg > 75 )

res = "Distinction";

else if ( avg > 60 )

res = "I class";

else if ( avg > 50 )

res = "II class";

else if ( avg > 35 )

res = "III class";

else

res = "Fail";

System.out.println ( "\n Average = "+avg );

System.out.println ( "\n Result = "+res );

}

}

class MultilevelInheritEx3

{

public static void main ( String args [ ] )

{

Result R = new Result ( );

R.pno = 101;

R.pna = "Malli";

R.m1 = 90;

R.m2 = 90;

R.show ( ); // show ( ) of class Result is in active state and hence it is invoked

}

}

**Hierarchical inheritance:**

Deriving two (or) more classes from one base class is called hierarchical inheritance i.e., one super class and many sub classes.

**C**

**B**

**A**

Example:

class Person

{

int pno;

String pna;

void show ( )

{

System.out.println ( "\n\n Number : "+pno );

System.out.println ( "\n Name : "+pna );

}

}

class Student extends Person

{

int m1, m2, tot;

void show ( )

{

tot = m1 + m2;

super.show ( ); // show ( ) of super class i.e., Person class gets invoked

System.out.println ( "\n Marks : "+m1+" "+m2 );

System.out.println ( "\n Total : "+tot );

}

}

class Employee extends Person

{

int sal, hra, tax, nsal;

void show ( )

{

nsal = sal + hra - tax;

super.show ( ); // show ( ) of super class i.e., Person gets invoked

System.out.println ( "\n Net salary : "+nsal );

}

}

class HierarchialInheritEx4

{

public static void main ( String args [ ] )

{

Student S = new Student ( );

Employee E = new Employee ( );

S.pno = 101;

S.pna = "Malli";

S.m1 = 90;

S.m2 = 95;

E.pno = 102;

E.pna = "Arjun";

E.sal = 20000;

E.hra = 5000;

E.tax = 3000;

S.show ( ); // show ( ) of sub class i.e., Student class gets invoked

E.show ( ); // show ( ) of sub class i.e., Employee class gets invoked

}

}

**Multiple Inheritance:**

A derived class with more than one base class is known as multiple inheritance.

**A**

**B**

**C**

**Overriding Methods:-**

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behavior when method is called. That means, we should override the method defined in the super class. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the super class. Then when that method is called, the method defined in the subclass is invoked and executed of the super class. This is known as overriding. Illustrates the concept of overriding. The method display ( ) is overridden.

Example:

class super

{

int x;

super(int x)

{

this.x=x;

}

Final void display( )

{

System.out.println(“super x=”+x);

class sub extends super

{

int y;

sub(int x,int y)

{

super(x);

this.y=y;

}

void display( )

{

System.out.println(“super x=”+x);

System.out.println(“sub y=”+y);

}

}

class overridetest

{

public static void main(String args[ ])

{

sub.s1=new sub(100,200)

s1.display();

}

}

**Final variables and method :-**

All methods and variables can be sub overridden by default in -classes. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword final as a modifier. Whenever we want to declare a variable as a constant, we use final.

Ex:

final double PI=3.14;

final void show( )

{

------

------

}

Making a method final ensures that the functionality defined in this method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects the class.

Example:

class FinalEx

{

public static void main(String arg[])

{

int a;

final int b=100;

a=50;

a=a+100;

//b=75;

//b=b+100;

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

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

}

}

**Final Classes:-**

Sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be sub classed is called a final class. This is achieved in java using the keyword final as follows.

Ex:

final class Aclass

{

-----

-----

}

final class Bclass extends Aclass

{

----

----

}

Any attempt to inherit these classes will cause an error and will the compiler not allow it.

Declaring a class final prevents any unwanted extensions to the class. It also allows the compiler to perform some optimizations when a method of a final class is invoked.

**Finalizer Methods:-**

We have seen that a constructor method is used to initialize an object when it is declared. This process is known as initialization. Similarly java supports a concept called finalization which is just opposite to initialization. We know that java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources. In order to free these resources we must use a finalizer method. This is similar to destructors in C++.

The finalizer method is simply finalize ( ) and can be added to any class. Java calls that method wherever it is about to reclaim the space for that object. The finalize method should explicitly define the tasks to be performed.

**Abstract Methods and classes:-**

The method final we ensure that the method is not redefined in a subclass. That is the method can never be subclassed. Java allows us to do something that is exactly opposite to this. That is we can indicate that a method must always be redefined in a subclass thus making overriding compulsory. This is done using the modifier keyword abstract in the method definition.

Syntax:

abstract class class-name

{

---------

---------

abstract void method-name(parameter list);

}

Ex:

abstract class shape

{

……………….

abstract void draw ( );

……………….

}

When a class contains one or more abstract methods is should also be declared abstract.

While using abstract classes we must satisfy the following conditions:

* We cannot use abstract classes to instantiate objects directly. For example

Shape s=new shape ( );

Is illegal because shape is an abstract class.

* The abstract methods of an abstract class must be defined in its subclass.
* We cannot declare abstract constructors or abstract static methods.

Example1:

abstract class Person

{

int pno;

String pna;

abstract void putdata ( );

void putPerson ( )

{

System.out.println ( "\n Person Number : "+pno );

System.out.println ( "\n Person Name : "+pna );

}

}

class Student extends Person

{

int m1, m2, tot;

Student ( int pno, String pna, int m1, int m2 )

{

this.pno = pno;

this.pna = pna;

this.m1 = m1;

this.m2 = m2;

}

void putdata ( )

{

tot = m1 + m2;

System.out.println ( "\n Student Number : "+pno );

System.out.println ( "\n Student Name : "+pna );

System.out.println ( "\n Marks : "+m1+" "+m2 );

System.out.println ( "\n Total : "+tot );

}

void putResult ( )

{

double avg = ( double )tot/2;

String res = "";

if ( avg > 75 )

res = "Distinction";

else if ( avg > 60 )

res = "I class";

else if ( avg > 50 )

res = "II class";

else if ( avg > 35 )

res = "III class";

else

res = "Fail";

System.out.println ( "\n Result : "+res );

}

}

class AbstractEx1

{

public static void main ( String args [ ] )

{

Student S = new Student ( 101, "Praveen", 90, 95 );

Person P = S; // Super class instance can refer to sub class instance

P.putdata ( );

S.putResult ( );

}

}

Example2:

abstract class Person

{

int pno;

String pna;

abstract void putdata ( );

}

class Student extends Person

{

int m1, m2, tot;

Student ( int pno, String pna, int m1, int m2 )

{

this.pno = pno;

this.pna = pna;

this.m1 = m1;

this.m2 = m2;

}

void putdata ( )

{

tot = m1 + m2;

System.out.println ( "\n\n Student Number : "+pno );

System.out.println ( "\n Student Name : "+pna );

System.out.println ( "\n Marks : "+m1+" "+m2 );

System.out.println ( "\n Total : "+tot );

}

}

class Employee extends Person

{

int sal, hra, tax, nsal;

Employee ( int pno, String pna, int sal, int hra, int tax )

{

this.pno = pno;

this.pna = pna;

this.sal = sal;

this.hra = hra;

this.tax = tax;

}

void putdata ( )

{

nsal = sal + hra - tax;

System.out.println ( "\n\n Employee Number : "+pno );

System.out.println ( "\n Employee Name : "+pna );

System.out.println ( "\n Salary : "+sal );

System.out.println ( "\n Net salary : "+nsal );

}

}

class AbstractPolyEx2

{

public static void main ( String args [ ] )

{

Person P;

Student S = new Student ( 101, "Praveen", 90, 95 );

Employee E = new Employee ( 105, "Krishna", 1500, 500, 300 );

P = S; // P got binded to S

P.putdata ( ); // putdata ( ) of Student invoked

P = E; // P got binded to E

P.putdata ( ); // putdata ( ) of Employee invoked

} }

* **Visibility control:-**

We stated earlier that it possible to inherit all the members of a class by a subclass using the keyword extends. We have also seen that the variables and methods of a class are visible everywhere in the program. However it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in java by applying visibility modifiers to the instance variables and methods. The visibility modifiers are also known as access modifiers. Java provides three types of visibility modifiers: **public, private and protected**. They provide different levels of protection as described below.

**Public Access:-**

Any variable or method is visible to the entire class in which it is defined. What if we want to make it visible to all the classes outside this class? This is possible by simply declaring the variable or method as public.

Example:

public int number:

public void sum( );

{

…………………

…………………

}

A variable or method declared as public has the widest possible visibility and accessible everywhere. In fact this is what we would like to prevent in many programs. This takes us to the next levels of protection.

**Friendly Access :-**

In many of our previous examples, we have not used Public modifier, yet they were still accessible in other classes in the program. When no access modifier is specified the member defaults to a limited version of public accessibility known as ” friendly“ level of access.

The difference between the public access and the “friendly” access is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages (A package is a group of related classes stored separately.). A package in java is similar to a source file in C.

**Protected Access:-**

The visibility level of a “protected” filed lies in between the public access and friendly access. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that not –subclasses in other packages cannot access the “protected” members.

**Private Access:-**

Private fields enjoy the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as private behaves like a method declared as final. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

**Private Protected Access :-**

A field can be declared with two keywords private and protected together like:

private protected int codenumber;

This gives a visibility level in between the “protected” access. This modifier makes the filed visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package. Summarizes the visibility provided by various access modifiers.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Visibility of field in a class | | | | | |
| **Access modifier**  *Access location* | Public | Protected | Friendly(default) | Private protected | Private |
| Same class | Yes | Yes | Yes | Yes | Yes |
| Subclass in same package | Yes | Yes | Yes | Yes | No |
| Other classes in same package | Yes | Yes | Yes | No | No |
| Subclass in other package | Yes | Yes | No | Yes | No |
| Non-subclasses in other packages | Yes | No | No | No | No |

**Rules of Thumb:-**

The details discussed so far about field visibility may be quite confusing and seem complicated. Given below are some simple rules for applying appropriate access modifiers.

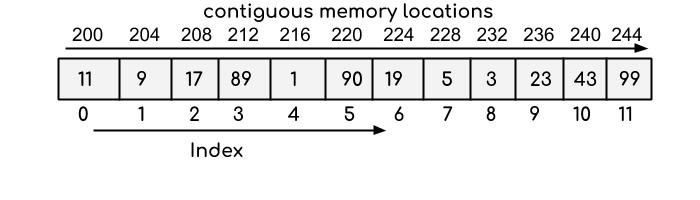
1. Use public if the field is to be visible everywhere.
2. Use protected if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use “default” if the field is to be visible everywhere in the current package only.
4. Use private protected if the field is to be visible only in subclasses regardless of packages.
5. Use private if the field is not to be visible any where except in its own class.

**Arrays, Strings and Vectors**

**Introduction**

An array is a group of contiguous or related data items that share a common name (or) An array is a collection of similar elements. For instance, we can define an array name **salary** to represent a set of salaries of a group of employees. A particular value is indicated by writing a number called *index* number or *subscript* in brackets after the array name. For example,

salary[10]



represents the salary or the 10th employee. While the complete set of values is referred to as an array,the individual values are called elements of array. Different types of arrays are

1) Single dimensional array 2) Multi-dimensional array.

Arrays can be of any variable type.

**One dimensional Arrays**

list of items can be given one variable name using only one subscript and such a variable is called a *single-subscripted* variable or a *one-dimensional* array.

For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable number, then we may create the variable number as follows

int number [ ] = new int[5];

and the computer reserves five storage locations as shown below :

|  |  |
| --- | --- |
|  | number[0] |
|  | number[1] |
|  | number[2] |
|  | number[3] |
|  | number[4] |

The values to the array elements can be assigned as follows:

number[0]= 35;

number[1]=40;

number[2]=20;

number[3]=57;

number[4]=19;

This would *cause* the array number to store the values shown as follows:

|  |  |  |
| --- | --- | --- |
| number[0] | 35 |  |
| number[1] | 40 |  |
| number[2] | 20 |  |
| number[3] | 57 |  |
| number[4] | 19 |  |

**Note** : In Java, subscripts starts with the value 0.

These elements may be used in programs just like any other Java variable. for example, the following are the valid statements :

int res=number[0]+10;

number[4]=number[1]+number[2];

**Creating an Array**

Like any other variables, arrays must be declared and created in the computer memory before they are used. Creation of an array involves three steps;

1. Declaring the array

2. Creating memory locations

3. Putting values into the memory locations.

**Declaration of Arrays**

Arrays in Java may be declared in two forms

|  |  |
| --- | --- |
| Form 1 | type arrayname[ ] ; |
|  |  |
| Form 2 | type [ ] arrayname ; |

Examples :

int number[ ];

float average[ ];

int[ ] counter;

float[ ] marks;

Remember, we do not enter the size of the arrays in the declaration.

**Creation of Arrays**

After declaring an array, we need to create it in the memory. Java allows us to create arrays using new operator only, as shown below:

|  |
| --- |
| arrayname= new arrayname[size] ; |

Examples :

number= new int[5] ;

average=new float[10] ;

It is also possible to combine the two steps --declaration and creation -- into one as shown below

int number= new int[5] ;

**Initialization of Arrays**

The final step is to put values into the array created. This process is known as *initialization.* This is done using the array subscripts as shown below.

|  |
| --- |
| arrayname[subscript]= value ; |

Example :

number[0]=35;

number[1]=40;

….. ……..

number[4]=19;

Note That Java creates arrays starting with the subscriptof 0 and ends with a value one less, than the size specified.

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared as shown below:

|  |
| --- |
| type arrayname[ ]= { list of values } ; |

The array initializer is a list of values separated by commas and surrounded by curly braces. Note that no size is given. The compiler allocates enough space for all the elements specified in the list.

Example :

int number [ ]={35,40,20,57,19 };

It is possible to assign an array object to another.

Example:

int a[ ]={1,2,3};

int b[ ];

b=a;

are valid in Java. Both the arrays will have the same values.

**Array Length**

In Java, all arrays store the allocated size in a variable named length. We can obtain the length of the array **a** using **a.length**

**Example:**

class Average

{

public static void main(String args[])

{

double marks[] = {10.1, 11.2, 12.3, 13.4, 14.5};

double result =0;

int i;

for(i=0; i<5; i++)

result = result + marks[i];

System.out.println("Average is " + result / 5);

}

}

**Two Dimensional Arrays**

So for we have discussed the array variables that can store a list of values. There will be situations where a table of values will have to be stored.

Java allows us to define such tables of items by using *two ­dimensional* arrays.

Two Dimensional array are stored in the memory as follows for a 5 row and 5 column table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Column  0 | Column  1 | Column  2 | Column  3 | Column  4 |
| Row 0 | [0][0] | [0][1] | [0][2] | [0][3] | [0][4] |
| Row 1 | [1][0] | [1][1] | [1][2] | [1][3] | [1][4] |
| Row 2 | [2][0] | [2][1] | [2][2] | [2][3] | [2][4] |
| Row 3 | [3][0] | [3][1] | [3][2] | [3][3] | [3][4] |
| Row 4 | [4][0] | [4][1] | [4][2] | [4][3] | [4][4] |

As with the single dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

For creating two-dimensional arrays. we must fallow the same steps as that of simple arrays. We may create a two-dimensional array like this:

int myArray[ ][ ];

myArray=new int[3][4];

or

int myArray[ ][ ]=new int[3][4];

This creates a table that can store 12 integer values, four across and three down.

Like the one-dimensional arrays, two-dimensional arrays may he initialized by following their declaration with a list of initial values enclosed in braces. For example,

int table[2][3]= { 0,0,0,1,1,1};

initializes the elements of the first row to zero and the second few to one. The above statement can be equivalently written as

int table[ ][ ]={ {0,0,0},{1,1,1} };

by surrounding the elements of each row by braces.

**Example:**

class DDA

{

public static void main(String args[])

{

int arr[][]={{1,2,3},{4,5,6},{7,8,9}};

for(int i=0;i<3;i++)

{

for(int j=0;j<3;j++)

{

System.out.print(arr[i][j]+" ");

}

System.out.println();

}

}

}

**Variable Size Arrays**

Java treats multidimensional array as “arrays of arrays". It is possible to declare a two-dimensional array as follows:

int x[ ][ ]=new int[3][ ];

x[0]=new int[2];

x[1]=new int[4];

x[2]= new int[3];

These statements create a two-dimensional array as having different lengths foreach row as shown in the following figure.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x[0] |  |  |  | X[0][1] |  |  |
| x[1] |  |  |  |  |  | X[1][3] |
| x[2] |  |  |  |  | X[2][2] |  |

**Strings**

Strings represent a sequence of characters. The easiest way to represent a sequence of characters in Java is by using a character array.

Example:

char charArray[ ] =new char[4];

charArray[0] =’J’;

charArray[1] =’A’;

charArray[2] =’V’;

charArray[3] ='A';

Although character arrays have the advantage of being able to query their length, they themselves are not good enough to support the range of operations we may like to perform on strings.

In Java strings are class objects end implemented using two classes, namely, String and StringBuffer. A Java string is an instantiated object of the String class. A Java string is not a character array and is not NULL terminated. Strings may be declared and created as follows:

|  |
| --- |
| String stringName;  StringName= *new String(“string”);* |

Example :

String stuName;

stuName= new String(“Ajay”);

These two statements may be combined as follows

String stuName=new String(“Ajay”);

**String Arrays**

We can also create and use arrays that contain strings. The statement

String itemArray[ ]= new String[3];

will create an itemArray of size 3 to hold three string constants. We can assign the strings to the hem Army element by element using three different statements or more efficiently using a for loop.

**String Methods**

The String class defines a number of methods (hat allow us to accomplish a variety of string manipulation tasks.The following list are some of most commonly used string methods.

|  |  |
| --- | --- |
| Method Call | Task performed |
| s2=s1.toLowerCase(); | Converts the string ‘s1’ to lower case |
| s2=s1.toUpperCase(); | Converts the string ‘s1’ to Upper case |
| s2=s1.replace(‘x’,’y’); | Replaces all appearances of ‘x’ with ‘y’ |
| s1.equals(s2) | Returns “true” if ‘s1’ is equals to ‘s2’ |
| s1.equalsIgnoreCase(s2) | Returns “true” if s1=s2,ignoring the case of characters |
| s1.length(); | Gives the length of ‘s1’ |
| s1.charAt(n); | Gives ‘n’th character or ‘s1’ |
| s1.compareTo(s2) | Returns –ve if s1<s2, +ve if s1>s2 and 0 if s1=s2 |
| s1.concat(s2) | Concatenates ‘s1’ and ‘s2’ |
| s1.substring(n) | Gives the substring starting from ‘n’ th character |
| p.toString() | Creates a string representation of object ‘p’ |
| s1.indexOf(‘x’) | Gives the position of first occurrence of ‘x’ |

**Example**

class StringOrdering

{

public static void main(String args[ ])

{

String name[] = {"HYDERABAD", “VIJAYAWADA", "KHAMMAM", "Guntur" ,"NIZAMABAD"} ;

int size = name.length;

String temp = null;

for (int i=0;i<size; i++)

{

for (int j= i+1; j <size; j++ )

{

if (name[j] .compareTo(name[i]) < 0)

{

temp =name[i];

name[i] =name[j];

name[j] = temp;

}

}

}

for(int i=0;i<size;i++)

{

System.out.println(name[i]);

}

}

}

**String Buffer Class**

StringBuffer is a peer class of String. While String creates strings of fixed length, StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end. The following table lists some of the methods that are frequently used in string manipulations.

|  |  |
| --- | --- |
| Method Call | Task performed |
| s1.setCharAt(n,’x’) | Modifies the ‘n’th character to ‘x’ |
| s1.append(s2) | Appends the string ‘s2’ to ‘s1’ at the end |
| s1.insert(n,s2); | Inserts the string ‘s2’ at the position ‘n’ |

**Example :**

class StringManipulation

{

public static void main(String args[])

{

StringBuffer str = new StringBuffer("Object language");

System.out.println("Original String : "+ str);

System.out.println("Length of String : "+ str.length());

String aString = new String(str.toString());

int pos =aString.indexOf("language" );

str.insert(pos, " Oriented ");

System.out.println("Modified string : " + str);

str.setCharAt(6 , '-');

System.out.println("String now : " +str);

str. append(" JAVA");

System.out.println("Appended string : "+ str);

}

}

**Vectors**

Vector class contained in the **java.util** package can be used to create a generic dynamic array known as *vector* that can hold objects of any type and any number*.* The objects do not have to be homogeneous. Arrays can be easily implemented as vectors. Vectors are created like arrays as follows

Vector v1=new Vector( ); --- declaring without size

Vector v2=new Vector(3); -- declaring with size

Note that a vector cam be declared without specifying any size explicitly. A vector without size can accommodate unknown number of items. Even, when a size is specified ,this can be overlooked and a different number of items may be put into the vector. Remember, in contrast ,an array must always have its size specified.

Vectors possess a number of advantages over arrays

1.It is convenient to use vectors to store objects.

2.A vector can be used to store a List of objects that may vary in size.

3.We can add and delete objects from the List as and when required.

A major constraint in using vectors is that we cannot directly store simple data type in a vector. we can only store objects. Therefore we need to convert simple types to objects. This can be done using the **wrapper classes**. The vector class supports a number of methods that can be used to manipulate the vectors.

|  |  |
| --- | --- |
| Method Call | Task performed |
| list.addElement(item) | Adds the item specified to the list at the end |
| list.elementAt(10) | Gives the name of the ‘10’th object |
| list.size() | Gives the number of objects present |
| list.removeElement(item) | Removes the specified item from the list |
| list.removeElementAt(n) | Removes the item stored in the ‘n’th position |
| list.removeAllElements() | Removes all the elements of the list |
| list.copyInto(array) | Copies all items from list to array |
| list.insertElementAt(item,n) | Inserts the item at ‘n’th position |

**Example :**

import java.util.\*;

class LanguageVector

{

public static void main(String args[])

{

Vector list = new Vector();

int length = args.length;

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

{

list.addElement(args[i]);

}

list.insertElementAt("COBOL",2);

int size = list.size();

String listArray[] =new String[size];

list.copyInto(listArray);

System.out.println( "List of Languages..");

for (int i=0; i< size; i++)

{

System.out.println(listArray[i]);

}

}

}

CommandLine input and outpur are…

java LanguageVector C C++ JAVA DBMS

List of Languages..

C

C++

COBOL

JAVA

DBMS

**Wrapper Classes**

As pointed out earlier, vectors cannot handle primitive data types like int, float, long. char, and double. Primitive data types may be converted into object types by using the wrapper classes contained in the java.lang package. The following Table shows the simple data types and their corresponding wrapper class

|  |  |
| --- | --- |
| Simple DataType | Wrapper Class |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |

The wrapper classes have a number of unique methods for handling primitive data types and objects. They are listed in the following tables.

Converting primitive numbers to objects using Constructor methods

|  |  |
| --- | --- |
| Constructor calling | Conversion action |
| Integer intVal=new Integer(i) ; | Primitive integer to Integer object |
| Float floatVal=new Float(f); | Primitive float to Float Object |
| Double doubleVal=new Double(d) ; | Primitive double to Double object |
| Long longVal=new Long(l); | Primitive long to Long object. |

Converting primitive numbers to objects using typeValue( ) method

|  |  |
| --- | --- |
| Method calling | Conversion action |
| int i=intVal.intValue( ); | Object to primitive integer |
| float f=floatVal.floatValue( ); | Object to primitive float |
| long l=longVal.longValue( ); | Object to primitive long |
| double d=doubleVal.doubleValue( ); | Object to primitive double |

Converting numbers to string using toString( ) method

|  |  |
| --- | --- |
| Method calling | Conversion action |
| str=Integer.toString(i ); | Primitive integer to String |
| str=Float.toString(f); | Primitive float to String |
| str=Double.toString(d); | Primitive double to String |
| str=Long.toString(l); | Primitive long to String |

Converting String objects to numeric objects using static method valueOf( )

|  |  |
| --- | --- |
| Method calling | Conversion action |
| intVal=Integer.valueOf(str) ; | Converts string to Integer object |
| floatVal= Float.valueOf(str); | Converts string to Float object |
| doubleVal= Double.valueOf(str) ; | Converts string to Double object |
| longVal= Long.valueOf(str); | Converts string to Long object |

Converting Numeric Strings to primitive numbers using parsing methods

|  |  |
| --- | --- |
| Method calling | Conversion action |
| int i=Integer.parseInt(str); | Converts string to primitive integer |
| long j=Long.parseLong(str); | Converts string to primitive long |

**Interfaces: Multiple Inheritance**

Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

class A extends B extends C

{  
 ---------

---------

}

is not permitted in Java. However the designers of Java could not overlook the importance of multiple Inheritance. Java provided an alternate approach known as *interface*  to support the concept of multiple inheritance. Although a Java class cannot be a subclass of more than one superclass, it can implement more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance,

**Defining Interfaces**

An interface is basically a kind of class. Like classes, interlaces contain method.\* and variables but with *a* major difference, The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implement an interface to define the code for implementation of these methods

The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is;

interface InterfaceName

{

Variable Declarations;

Method Declarations;

}

Here, interface is the key word and InterfaceName is any valid Java variable (just like class names). Variables are declared as follows:

|  |
| --- |
| final type variableName=value; |

Note that all variables are deflated as constants. Methods declaration will contain only a list of parameters without any body statements. Example:

|  |
| --- |
| return-type methodname1(parameter list); |

Here is an example of an interface definition that contains one variable and two methods:

interface Area

{

final float PI=3.14f;

float compute( float x, float y) ;

void show();

}

Note that the code for the method is not included in the interface and the method declaration simply ends with a Semicolon. The class that implements this interface must define the code for the method.

**Extending Interfaces**

Like classes. Interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subInterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword extends as shown below;

interface name2 extends name1

{

Body of name2;

}

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces, After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements derived interface to define all the methods.

**Implementing Interfaces**

Interfaces are used as "super classes" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows**:**

|  |
| --- |
| class classname implements interfacename  {  body of class name;  } |

Example :

interface Area

{

final float PI=3.14f;

float compute(float x, float y);

}

class Rectangle implements Area

{

public float compute(float a, float b)

{

return(a\*b);

}

}

class Circle implements Area

{

public float compute( float a, float b)

{

return(PI\*a\*a);

}

}

class InterfaceTest

{

public static void main(String args[ ])

{

Rectangle rect=new Rectangle();

Circle cir=new Circle();

float res1=rect.compute(20,30);

float res2= cir.compute(10,0);

System.out.println("Area of rectangle..."+res1);

System.out.println("Area of circle..."+res2);

}

}

**Accessing Interface Variables**

interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header tiles in C++ to contain a large number of constants. Since such interfaces do not contain methods,there is no need to worry about implementing any methods. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value. Example:

interface A

{

int x=20;

int y=50;

}

class B implements A

{

int a=x;

void method( int size)

{

--------

--------

if(size >y)

----------

}

}

**Example :**Program for implementing multiple inheritance by using Interfaces.

class student

{

int rollNumber;

void getNumber(int n)

{

rollNumber=n;

}

void printNumber()

{

System.out.println("RollNo is " +rollNumber);

}

}

  class test extends student

{

float part1,part2;

void getMarks(float a, float b)

{

part1=a;

part2=b;

}

void putMarks()

{

System.out.println("Marks Part1 "+part1);

System.out.println("Marks Part2 "+part2);

}

}

interface sports

{

float sportwt=6.0F;

void putwt();

}

class results extends test implements sports

{

float total;

public void putwt()

{

System.out.println("Sports Marks "+ sportwt);

}

void display()

{

total=part1+part2+sportwt;

System.out.println("Total marks of " +rollNumber+" is "+total);

}

}

class mainClass

{

public static void main(String srgs[])

{

results a=new results();

a.getNumber(10);

a.printNumber();

a.getMarks(10.0F,25.5F);

a.putMarks();

a.putwt();

a.display();

}

}