Unit II
Arrays

An array is a collection of variables of the same type, referred to by a common name. An array organizes the data in such a way that it can be easily manipulated.

One-Dimensional Arrays

A one dimensional array is a list of related variables. It can be declared using following format:

```
type array-name[]=new type[size];
```

Here, type declares the element type of the array. The element type determines the data type of each element contained in the array. The number of elements that the array will hold is determined by size. Since arrays are implemented as objects, the creation of an array is a two-step process.

1. An array reference variable is declared
2. Memory is allocated to the array declared.

```
int sample[]=new int[10];
```

It is possible to break the declaration into two steps as follows:

```
int sample[];
sample=new int[10];
```

Individual elements can be accessed by the use of an index. An index describes the position of an element within an array. Array index starts from 0.

Example

```
//Demonstrate one dimensional array
class ArrayDemo{String args[]) {
int sample[]=new int[10];
int i;
for(i=0;i<10;i++)
 sample[i]=i;
for(i=0;i<10;i++)
System.out.println("This is Sample[ "+i+"] :"+sample[i]);
}  }
```

Output

```
This is Sample[0] : 0
This is Sample[1] : 1
This is Sample[2] : 2
This is Sample[3] : 3
This is Sample[4] : 4
This is Sample[5] : 5
This is Sample[6] : 6
This is Sample[7] : 7
This is Sample[8] : 8
This is Sample[9] : 9
```

An array can be initialized at the time of creation. The general format is

```
type array-name[] ={ val1, val2, val3, ....., valN}
```

Here, the initial values are specified bay val1 through valN. They are assigned in sequence left to right in index order. Java automatically allocates an array large enough to hold the initializers. There is no need to explicitly use new operator.

Example:

```
//Demonstrate Use of Array initializers
class Minimax {
public static void main(String args[]) {
int nums[]={ 99, 0, -10, 1023, 19, -99, 5678, 800, 784, -9 };
```

```
int min, max;
min=max=nums[0];
for(int i=;i<10;i++) {
if(max<nums[i]) max=nums[i];
if(min>num[i]) min=num[i]); }
System.out.println("Min and Max : " +min +" " +max); } }
```

Array boundaries are strictly enforced in Java; it is a run-time error to overrun or underrun the end of an array.

```
// Demonstrate Array overrun
class ArrayErr{
public static void main (String args[]) {
int sample[]=new int[10];
int i;
for(i=0;i<100;i++)  // This will generate
ArrayIndexOutOfBoundsException
 sample[i]=i; } }
```

Two-Dimensional Arrays

Simplest form of multi-dimensional array is two dimensional array. It can be declared as follows:

```
int table[][]=new int[10][10];
```

Example:

```
//Demonstrate two dimensional array
class TwoD{
public static void main(String args[]) {
int t, i;
int table[][]=new int[3][4];
for(t=0;t<3;t++) {
for(i=0;i<4;i++) {
table[t][i]=t*4+i+1;
System.out.print(table[t][i] + " "); } }
System.out.println(); } }
```

Irregular Arrays

When allocating memory for a multidimensional array, only the memory for the leftmost dimension needs to be specified. The remaining dimensions can be allocated separately. We need not allocate the same number of elements for each index. The length of each array can be controlled by the programmer.

Example

```
//Demonstrate irregular arrays
class ArrDemo2{
public static void main(String args[]) {
int riders[][]=new int[6][];
riders[0]= new int[10];
riders[1]= new int[10];
riders[2]= new int[10];
```

```
riders[3]= new int[10];
riders[4]= new int[5];
riders[5]= new int[5];
int i,j;
for(i=0;i<4;i++)
for(j=0;j<10;j++)
  riders[i][j]=i+j+10; // initializing with s
for(i=4;i<6;i++)
for(j=0;j<5;j++)
  riders[i][j]=i+j*10;
for(i=0;i<4;i++){
for(j=0;j<10;j++)
  System.out.print(riders[i][j]);
System.out.println(); }
for(i=4;i<6;i++) {
for(j=0;j<5;j++)
System.out.print(riders[i][j]);
System.out.println(); }  } }
```

Arrays of Three or more Dimensions

Java allows arrays with three or more dimensions. The general format is:

```
type name[][][] .....[]=new type[size1][size2].......[sizeN];
```

Example: `int muldim[][][]=new int[4][10][3];`

Alternate Array Declaration Syntax

There is a second form that can be used to declare an array:

```
type [] var-name;
```

The following statements are equivalent:

```
int count[] = new int[10];
int [] count = new int[10];  // Both are one and the same
```

The alternative declaration form offers convenience when declaring several arrays at the same time.

For example:

```
int[] num1, num2, num3; //  This statement declares 3 arrays.
```

Assigning Array References

When one array reference variable is assigned to other, we are changing what object that variable refers to.

Example

```
classArrRef{
public static void main(String args[]) {
int i;
int num1[]=new int[10];
int num2[]=new int[10];
for (i=0;i<10;i++)
  num1[i]=i;
for (i=0;i<10;i++)
  num2[i]= -i;
System.out.print("Here is num1 :");
for (i=0;i<10;i++)
  System.out.print (num1[i] + " ";
System.out.println();
```

Output
Here is num1 0 1 2 3 4 5 6 7 8 9
Here is num2 -1 -2 -3 -4 -5 -6 -7 -8 -9
Here is num2 after change
0 1 2 3 4 5 6 7 8 9
Here is num2 after change
0 1 2 -99 4 5 6 7 8 9

```
System.out.print("Here is num2: ");
for (i=0;i<10;i++)
  System.out.print (num2[i] + " ");
System.out.println();
num2=num1;
System.out.println("Here is num2 after change");
for (i=0;i<10;i++)
  System.out.print (num2[i] + " ");
System.out.println();
num2[3]=-99;
System.out.println("Here is num2 after change");
for (i=0;i<10;i++)
  System.out.print (num2[i] + " " ); }}
```

Using the Length Member

Because, arrays are implemented as objects, the length instance variable associate with each array contains the number of elements that the array can hold.

Example:

```
//Demonstrate Array Length
class LengthDemo{
public static void main(String args[]){
int list[]={ 1, 0, 99, 8, -7, 74};
for(int i=0; i<list.lenght;i++)
System.out.print(list[i] + " ") } }
```

The For-Each Style for Loop

When working with arrays, it is common to encounter situations in which each element in an array must be examined from start to finish. Java defines a second form of for loop that streamlines this operation. This form is "for-each" style loop. It cycles through an array in a strictly sequential manner from start to finish. The general form of this loop is :

```
for(type itr-var : collection) statement-block
```

Here, type specifies the type, and itr-var specifies the name of the iteration variable that will receive the elements from a collection, one at a time, from beginning to end. With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all the elements in the collection are obtained. The type must be same or compatible with that of array elements.

```
class ForEach{
public static void main(String args[]){
int list[]={ 1,2,3,4,5,6,78,9,10};
sum=0;
for(int x:list)
{ System.out.print(x + " ");
  sum+=x; }
  System.out.prinln("\n" + "Sum= "+sum); } }
```

Output

1 2 3 4 5 6 7 8 9 10

Sum= 55

Although the for-each loop iterates until all elements in the array have been examined, it is possible to terminate the loop early by using a break statement.

```
for(int x:list)
{ System.out.print(x + " ");
  sum+=x; }
  if(x==5) break; } }
```

The iteration variable of for-each loop is read only. We can't change the contents of the array by assigning the iteration variable a new value.

```
class Nochange{
public static void main(String args[]){
int list[]={ 1,2,3,4,5,6,78,9,10};
for(int x:list)
{ System.out.print(x + " ");
  x=x*10; }
Sustem.out.println();
for(int x: list)
System.out.print(x + " "); } }
```

Output
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

Strings

In Java, strings are objects. The syntax for creating strings is given below:
String str = new String("Hello");
String str2 = new String(str1);

Operating on Strings

boolean equals(str)	Returns true if the invoking string contains same character sequence as str
int length()	Obtains the length of the string
char charAt(index)	Obtains the character at specified index
int compareTo(str)	Returns less than zero if the invoking string is less than str, greater than zero if the invoking string is greater than str and zero if the strings are equal
int indexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure
int lastIndexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure

```
class StrOp {
public static void main(String args[]) {
String str1= " Java is #1 programming language";
String str2=new String(Str1);
String str3= "Java Strings are powerful";
int res, idx;
char ch;
System.out.println("Length of str1 = " +str1.lenght());
for(int i=0; i<str1.length(); i++)
  System.out.print(str1.charAt(i));
 System.out.println();

if(str1.equals(str2))
  System.out.println("str1 is equal to str2 ");
else
System.out.println("str1 is  not equal to str2");
if(str1.equals(str3))
  System.out.println("str1 is equal to str3 ");
else
System.out.println("str1 is  not equal to str3 ");
res=str1.compareTo(str3);
if(res==0)
 System.out.println(" str1 and str3 are equal");
```

Output
Length of str1 = 31
str1 is equal to str2
str1 is not equal to str3
srt1 is greater than str3
The first occurrence of One 0
The last occurrence of One 14

```
else if(result < 0)
System.out.println(srt1 is less than str3);
else
System.out.println(srt1 is greater than str3);
str2=”One Two Three One”;
idx=str2.indexOf(“One”);
System.out.println(“The first occurrence of One” +idx);
idx=str2.lastIndexOf(“One”);
System.out.println(“The last occurrence of One” +idx); } }
```

Using Command-Line Arguments

In certain cases, we may need to provide inputs at the time of execution. This is achieved in Java by using command line arguments. Command line arguments are parameters that are supplied to a program at the time of execution.

Example

```
// Demonstrate Command Line arguments
class ComLineTest {
public static void main(String args[]) {
int count;
count=args.length;
System.out.println(“There are  ” +count “ Number of arguments”);
for(int i=0;i<count;i++)
  System.out.println( i + “ : Java is ” +args[i]) + ” !”); } }
```

After compilation, we can supply arguments to the example above. Any arguments provided in the command line are passed to the array args[] as its elements. We can simply access the array elements the array elements and use them in the program as we wish.

The above program can be executed as follows:

```
java ComLineTest Simple Obejet_Oriented Distributed Robust Secure
Portable Multithreaded Dynamic
```

Upon execution, the command line arguments Simple, Object_Oriented etc are passed to the program through the array args. That is, args[0] contains Simple, args[1] contains Object_Oriented and so on. These elements are accessed using the loop variable i **as `name =args[i];`**

The output will be
There are 8 Number of arguments
1 : Java is Simple !
2: Java is Object_Oriented !
3: Java is Distributed !
4: Java is Robust !
5: Java is Secure !
6: Java is Portable !
7: Java is Multithreaded !
8: Java is Dynamic !

Classes Objects and Methods

Class Fundamentals

Java is a true Object-Oriented language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the state and behaviour of the basic program components known as objects. A class contains both data (referred to as attributes), and executable code (referred to as methods). Classes create objects and objects use methods to communicate between them.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called fields and the functions are called methods. Calling a specific method in an object is described as sending the object a message or message passing.

A class is a template that defines the form of an object. It specifies both the data and the code that will operate on the data. Java uses a class specification to construct objects. Objects are instances of a class. Class is a logical representation. The physical representation of that class will be created when an object is created.

Declaration

A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create 'variables' of that type using declarations that are similar to the basic type declarations. In Java, these variables are termed as *instances* of classes. Which are the actual *objects*. Unlike data structures in other languages which only contain data, Java classes consist of both *attributes* and *behaviors*. *Attributes* represent the data that is unique to an instance of a class, while *behaviors* are methods that operate on the data to perform useful tasks.

A class is created by using the keyword class. The general form is as follows:

```
class classname
{
  type var1;
  type var2; ......
  type var n;
  return-type method-name-1(arguments)
  {
  body of the method ;
  }
  return-type method-name-2(arguments)
  {
  body of the method ;
  } ......
  return-type method-name-n(arguments)
  {
  body of the method ;
  }
}
```

The class with only data fields has no life. The objects created by such a class cannot respond to any message. Methods are necessary for manipulating data contained in a class. Methods are declared inside the body of the class after declaring instance variables. Method declaration should have return type, method name list of parameters and body. The method name main() is reserved. A method contains one or more Java statements but performs only one task.

Example:

```
class Rectangle
{
   int length, width;
   int area()
   {
   int a=height * width;
                    return a;
   }
}
```

Creating Objects

An object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in Java are created using the ***new*** operator. The ***new*** operator creates an object of the specified class and returns a reference to that object. This reference is then stored in a variable.

Example:

```
Rectangle rect1;
rect1= new Rectangle();
```

Both statements can be combined in to a single statement as follows:

```
Rectangle rect1= new Rectangle();
```

The method Rectangle() is the default constructor of the class. We can created any number of objects of type Rectangle.

Accessing Class Members

The variables and methods inside a class can be accessed from outside using *dot* operator. The format is as shown below:

```
objectname.variablename;
objectname.methodname(parameter-list);
```

Example:

```
rect1.length=10;
rect1.width=50;
rect2.lenght=60;
rect2.width=50;
```

The two objects rect1 and rect2 store different values as shown below:

rect1	length	10
	width	50

rect2	length	60
	width	50

Example

```
class  Rectangle
{
int length, width;
void getData(int x, int y)
{
 length=x;
 width=y;
}
int rectArea()
{
int area=length *width;
return(area);
}}
class RectArea
{
public static void main(String args[])
{
int area1, area2;
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
rect1.lenght = 15;
rect1.width=10;
area1 = rect1.lenght * rect1.widht;
rect2.getData(20, 12);
area2=rect2.rectArea();
System.out.println("Area1  : " +area1);
System.out.println("Area2  : " +area2);
} }
```

Output
Area1 : 150
Area2 : 240

Constructors

In order to be used by a program, an object must first be *instantiated* from its class definition. A special type of method called a *constructor* is used to define how objects are created. Constructors are special methods which have same name as that of class. They do not specify return type, not even void. This is because they return the instance of the class itself.

Example:

```
class Rectangle
{
int length, width;
   Rectangle(int x, int y)
   {
   width=y;
   }
int rectArea()
{
int area=length *width;
return(area);
}}

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

Output
Area : 150

```
Rectangle rect1 = new Rectangle(15, 10);
int area= rect1.rectArea();
System.out.println("Area  : " +area1);
} }
```

Garbage collection and Finalizers

In java, objects are dynamically allocated from a pool of free memory by using the new operator. Memory is not infinite and the free memory can be exhausted. Thus, new operator may fail to allocate memory because there is insufficient free memory to create desired object. For this reason, the key component of any dynamic allocation scheme is the recovery free memory from unused objects. Java uses garbage collection mechanism to reclaim unused objects automatically. When no reference to an object exists, that object is assumed to be no longer needed and the memory occupied by the object is released. This recycled memory can be used for a subsequent allocation. For efficiency, the garbage collector will usually run only when two conditions are met.: there are object to recycle and there is a need to recycle them. Garbage collection requires time, so the Java run-time system does it only when it is appropriate.

The finalize() method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called finalize() method and it can be used to make sure that an object terminates cleanly. Syntax of adding a finalize() method is given below:

```
protected void finalize()
{
// Finalization code will be written here
}
```

The keyword protected prevents the access to finalize() by code defined outside the class.
The finalize() method is called just before garbage collection. It is not called when an object goes out of scope. We cannot come to know when finalize() gets executed.

Example:

```
class  FDemo
{
int x;
FDemo(int i) { x=i;
}
protected void finalize()
{
System.out.println(" Finalizing    " +x);
}
void generator(int i)
{
FDemo ob= new FDemo(i);
} }

class Finalize
{
public static void main(String args[])
{
int count;
FDemo ob = new FDemo(0);
```

```
//this keyword
public class Person {
 private String name;
 private int age;
 public Person(String name, int age) {
  this.name = name;
  this.age = age;
 }
       public void displayInfo() {
  System.out.println("Name: " + this.name);
  System.out.println("Age: " + this.age);
  this.greet();
       }
  public void greet() {
  System.out.println("Hello, I'm " + this.name);
       }
 public static void main(String[] args) {
  Person person1 = new Person("Alice", 30);
  person1.displayInfo();
      }
}
```

```
for (count=1; count < 100000; count++)
   ob.generator(count);
} }
```

The this Keyword

When a method is called, it is automatically passed an implicit argument that is a reference to the invoking object. This reference is called as **this**. Java syntax permits the name of the parameter or the local variable to be the same as the instance variable. When this happens, the local variable hides instance variable. The instance variable can be accessed by referring it through **this** keyword.

Example:

```
class Pwr
{
double b, val;
int e;
Pwr(double b, int e)
{
this.b=b;
this.e=e;
val=1; // can also be written as this.val=1
if(e==0) return
 for(; e>0; e--) val=val * b;
}
double get_pwr()
{ return this.val ;
} }
```

In this version, the names of parameters and names of instance variables are same. To uncover the instance variables, this keyword is used.

Controlling Access to Class Members

In its support for encapsulation, the class provides two major benefits. First, it links data with code that manipulates it. Second, it provides means by which access to members can be controlled. Restricting access to members of a class is a fundamental part of object-oriented programming as helps prevents the misuse of an object. By allowing access to data only through a well-defined set of methods, it is not possible for code outside the class to access and modify values of members directly.

Java's Access Modifiers

The member access control is achieved through the use of three access modifiers: *public, private and protected.* Protected will be applied only in case of inheritance. When a member of a class is modified public specifier, that member can be accessed by any other code in the program. When a member of a class is specified as private, that member can be accessed only by any other members in its class. The default access specifier is public. Access modifier precedes the rest of the type specification.

public

Any variable or method is visible to the entire class in which it is defined. We can make it visible to all the classes outside the class by simply declaring the variable or method as ***public***.

The 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.

Ex : ***public int*** number***;***

```
public void sum()
{        .....
}
```

friendly

When no access modifier is specified, the member defaults to a limited version of public visibility known as ***friendly***(default) 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 only in the same package, but not in other packages.

[Note : A package is a group of related classes stored separately.]

protected

The visibility level of protected field lies in between the public access and friendly access. That is, the 'protected' modifiers makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages. Note that non-subclasses in other packages cannot access the 'protected' members.

private

private fields enjoy the highest degree of protection. They are accessible only within 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 subclasses. Also note that we cannot override a non-private method in a subclass and then make it private.

private protected

This gives visibility level in between 'protected' access and 'private' access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package.

Rules of Thumb

- Use public if the field is to be visible everywhere in the current package and also subclasses in other packages
- Use default if the field is to be visible everywhere in the current package only
- Use private protected if the field is to be visible only in subclasses, regardless of packages
- Use private if the field is not to be visible anywhere except in its own class.

Access Modifier / Access Location	public	protected	friendly (default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same location	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclasses in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

```
class MyClass
{
  private int a; // a is private. Cannot be accessed outside the
class
  public int b; // b is public
  int c;  // c is public by default
  void seta(int a){ this.a=a;
}
   int getA()
{ return(a);
} }

class AccessDemo
{
public static void main(String args[])
{
MyClass ob= new MyClass();
Ob.setA(-99); // Allowed to modify a only through access function;
Ob.b=20; // allowed because b is public
Ob.c=34; // allowed because c is public
Ob.a=10; // Not allowed because a is private.
}}
```

This program will not compile because we access to the member ‘a’ outside the class which is not allowed.

Pass Objects to Methods

Till now only simple parameters are passed to methods. However objects can be passed to methods as parameters**.**

```
// Demonstrate objects as parameters to methods
class Block
{
 int a, b,c;
 int volume ;
Block(int i, int j, int k)
{
```

Output

```
ob1 is same dimension of ob2 true
ob1 is same dimension of ob3 false
ob1 is same volume as  ob3  true
```

```
  a=i; b=j; c=k;
  volume= a*b*c;
}
boolean sameBlock(Block ob)
{
   if ((ob.a==a) && (ob.b==b) && (ob.c==c)) return true;
else
return false;
}
boolean sameVolume(Block ob)
{
   if ((ob.volume==volume) return true;
else
return false;
} }

class PassOb
{
public static void main(String args[])
{
Block ob1=new Block(10, 2, 5);
Block ob2=new Block(10, 2, 5);
Block ob3=new Block(4,5, 5);
System.out.println( "ob1 is same dimension of ob2 "
+ob1.sameBlock(ob2);
System.out.println( "ob1 is same dimension of ob3 "
+ob1.sameBlock(ob3);
System.out.println( "ob1 is same volume as  ob3 "
+ob1.sameBlock(ob3);
} }
```

sameVolume

Pass-by Value and Pass-by Reference

Call by value or pass by value approach copies the vaue of an argument into the formal parameter of the subroutine. Therefore changes made to the parameter have no effect. When arguments are passed as primitive data types, the method is pass-by value.

```
class Test
{
void NoChange(int i, int j)
{
i=i+10;
 j=-j;
} }
class PassByVlaue
{
public static void main(String args[])
{
Test ob1 = new Test();
int a=15, b=20;
System.out.println("a and b before function call "+a + " " + b);
ob1.noChange(a,b);
System.out.println("a and b after function call "+a + " " + b);
} }
```

Output

```
a and b before function call 15 20
a and b after function  call 15 20
```

When an object is passed to a method, then changes made to the object by the method will be reflected. When a variable of type class is created, a reference to the object is made. This

method is called pass by reference and changes made to the object inside the method will affect the object used as an argument.

```
class Test
{
int a, b;
Test(int i, int j)
{
a=i;
b=j;
}
void change(Test ob)
{
  ob.a=ob.a+ob.b;
  Ob.b=-ob.b
} }
class PassByRef
{
public static void main(String args[])
{
Test ob1 = new Test(15, 20);
System.out.println("a and b before function call "+ob1.a + " " +
ob1.b);
ob1.change(ob1);
System.out.println("a and b after function call "+ob1.a + " " +
ob1.b);
} }
```

Output
```
a and b before function call 15 20
a and b after function  call 35 -20
```

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

// Returning an object.

```
class Test
{
int a;
Test(int i)
{
a = i;
}

Test incrByTen()
{
Test temp = new Test(a+10);
 return temp;
}  }

class RetOb
{
public static void main(String args[])
{
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
```

Output
```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

```
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: " + ob2.a);
}
}
```

Overloading Methods

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be *overloaded.* In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as ***polymorphism***. Methods are overridden on a signature-by-signature basis

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

The important points to note:

- A difference in return type only is not sufficient to constitute an overload and is illegal.
- You should restrict your use of overloaded method names to situations where the methods really are performing the same basic function with different data.
- The language treats methods with overloaded names as totally different methods and as such they can have different return types. However, since overloaded methods perform the same job with different data sets, they should produce same return type normally. There is one particular condition, however, under which it is sensible to define different return types. This is the situation where the return type is derived from the argument type and is exactly parallel with the arithmetic operators.
- Overloaded methods may call one another simply by providing a normal method call with an appropriately formed argument list.

```
class OverloadDemo
{
void test() {
System.out.println("No parameters");
}
void test(int a)
{
```

```
System.out.println("a: " + a);
}
void test(int a, int b)
{
System.out.println("a and b: " + a + " " + b);
}
double test(double a)
{
System.out.println("double a: " + a);
return a*a;
} }
class Overload
{
public static void main(String args[])
{
OverloadDemo ob = new OverloadDemo();
double result;

ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}  }
```

Output
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625

The method test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the parameter of the method. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

```
// Automatic type conversions apply to overloading.
class OverloadDemo
{
void test()
{
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b)
{
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a)
{
```

```
System.out.println("Inside test(double) a: " + a);
} }

class Overload
{
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
} }
```

Output

```
No parameters
a and b: 10 20
Inside test(double) a: 88
```

Inside test(double) a: 123.2

This version of **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.

Overloading Constructors

In addition to overloading normal methods, we can also overload constructor methods. In fact, for most real-world classes, overloaded constructors will be the norm, not the exception.

```
// Demonstrate Constructor Overloading
class MyClass
{
int x;
MyClass()
{
System.out.println("Inside MyClass()");
x=0;
}
MyClass(int i)
{
System.out.println("Inside MyClass(int)");
x=i;
}
MyClass(double d)
{
System.out.println("Inside MyClass(double)");
x= (int) d;
}
MyClass(int i, int j)
{
System.out.println("Inside MyClass(int, int)");
```

Output

```
Inside MyClass()
Inside MyClass(int)
Inside MyClass(double)
Inside MyClass(int,int)
ob1.x=0
ob2.x=88
ob3.x=17
ob4.x=8
```

```
x=i*j;
}}
class OvLoadConsDemo
{
public static void main(String args[])
{
MyClass ob1=new MyClass();
MyClass ob2=new MyClass(88);
MyClass ob3=new MyClass(17.23);
MyClass ob4=new MyClass(2,4);
System.out.printl("ob1.x : " +ob1.x);
System.out.printl("ob2.x : " +ob2.x);
System.out.printl("ob3.x : " +ob3.x);
System.out.printl("ob4.x : " +ob4.x);
}}
```

MyClass() is overloaded four ways, each constructing an object differently. The proper constructor is based upon the parameters specified when new is executed.

Recursion

In Java, a method can call itself. This process is called recursion and a method that calls itself is said to be recursive. When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative versions.

```
//Recursion Demo
class Factorial
{
int factR(int n)
{
if(n==1) return 1;
else return(n*factR(n-1));
} }
class Recursion{
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 " + f.factR(3));
System.out.println("Factorial of 4" + f.factR(4));
System.out.println("Factorial of 5 " + f.factR(5));
} }
```

Output

```
Factorial of 3 6
Factorial of 4 24
Factorial of 5 120
```

Understanding static

There will be times when we want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. both methods and variables can be declared **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

```
//Demonstrate static members
class StaticDemo
{
int x; // x is a normal variable
static int y; // y is a static variable
int sum()
{ return x+y; }
}

class SDemo
{
public static void main(String args[])
{
StaticDemo ob1 = new StaticDemo();
StaticDemo ob2 = new StaticDemo();
Ob1.x=10;
Ob2.x=20;
StaticDemo.y=50;
System.out.println("ob1. Sum ="+ob1.sum());
System.out.println("ob2. Sum ="+ob2.sum());
StaticDemo.y=100;
System.out.println("ob1. Sum ="+ob1.sum());
System.out.println("ob2. Sum ="+ob2.sum());
} }
```

Output

```
ob1.sum=60
ob2.sum=70
ob1.sum=110
ob2.sum=120
```

Static variable y is shared by both ob1 and ob2. Changing it affects the entire class.

Static Methods

Methods declared as static have several restrictions:

- They can directly call only other static methods.
- They can directly access only static data.
- They do not have a **this** reference.

The difference between the static method and a normal method is that the static method is called through its class name, without any object of that class being created.

```
// Demonstrate static method
class SMethod
{
static int val = 1024;
static int valDiv2()
{    return val/2;
} }
class SDemo
{
public static void main(String args[])
{
System.out.println("Value is "+SMethod.val);
System.out.println("After calling the function "+SMethod.valDiv2());
SMethod.val=4;
System.out.println("Value is "+SMethod.val);
System.out.println("After calling the function "+SMethod.valDiv2());
}}
```

Output

```
Value is 1024
After calling the function 512
Value is 4
After calling the function 2
```

Static methods can refer only static members. The following will not compile

```
class StaticError{
int denom=3;
static int val = 1024;
static int ValDiv() {
return val/denom;
}  }
```

This will generate a compile time error as the static method refers to a non static member.

Static Blocks

Static blocks are needed to initialize static variables before any of the static methods of a class are used. A static block is executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose.

```
//Demonstrate static block
class SBlock
{
static double r2;
static double r3;
static
{ System.out.println("Inside Static Block");
   r2=Math.sqrt(2.0);
   r3=Math.sqrt(3.0);
}
SBlock(String msg)
{
System.out.println(msg);
} }

class SDemo
{
public static void main(String args[]) {
SBlock ob = new SBlock("Inside Constructor ");
```

```
System.out.println("Root of 2 ="+SBlock.r2);
System.out.println("Root of 3 ="+SBlock.r3);
}}
```

Nested and Inner Classes

Nested class is a class that is declared within another class. A nested does not exist independently of its enclosing class. Thus the scope of a nested class is bounded by its outer class. It has access to all the variables and methods of its outer class.

```
// Nested Demo
class Outer
{
int nums[];
Outer(int n[])
{
nums = n;
}
void analyze()
{
Inner inob = new Inner();
System.out.println("Minimum  :"+inob.min());
System.out.println("Maximum  :"+inob.max());
System.out.println("Average :"+inob.avg());
}
class Inner
{
int min()
{
m=nums[0];
for(int I =1; i<nums.length;i++)
  if (nums[i]<m) m=nums[i];
return m ;
}


int max()
{
m=nums[0];
for(int I =1; i<nums.length;i++)
  if (nums[i]>m) m=nums[i];
return m ;
}
int avg()
{
int a=0;
for(int i=0;i<nums.length; i++)
  a+=nums[i];
return a/nums.length;
} } }

class NestDemo {
public static void main (String args[]) {
int x[]={3,2,1,5,6,9,7,8};
Outer outob = new Outer(x);
outob.analyze();
} }
```

Output
Minimum : 1
Maximum : 9
Average : 5

Varargs: Variable-Length Arguments

Sometimes it may be necessary to create a method that takes a variable number of arguments. This can be handled in two ways. If the maximum number of arguments is small and known, the methods can be overloaded. If the maximum number of arguments is large and unknown, we can construct a method that takes variable number of arguments.

A variable-length argument is specified by three periods (…)

```
//Demonstrate variable number of arguments
class VarArgs
{
static void vaTest(int … v) //
{
System.out.println("No. of args " +v.length());
  for(int i=0;i<v.length;i++)
   System.out.print(" arg " + i + " :" +v[i]);
}
public static void main(Sting args[])
{
vaTest(10);
vaTest(1,2,3);
vaTest();
} }
```

Output
No. of args : 1
arg 0 : 10
No. of args : 3
arg 0 : 1
arg 1 : 2
arg 2 : 3
No. of args 0

Inheritance

Inheritance Basics

Inheritance is one of the basic principles of object-oriented programming. A class that is inherited by is called superclass. The class that does the inheritance is called subclass. It inherits all the variables and methods defined by the superclass and adds its own unique elements.
Java supports inheritance by using the keyword extends. The general form of a class declaration that inherits a superclass is given below:

```
class subclass-name extends superclass-name {
// body of the class
}
//Demonstrate single inheritance
class Shape{
      int height, width;

      void showDim() {
      System.out.println("Width = " +width + "Height ="+height); } }

class Triangle extends Shape{
String style;
void showStyle(){
System.out.println("Triangle is " +style); }
int area() {
return(height*width /2); } }
```

```
class InheritDemo {
public static void main(String args[]) {
Triangle t1 = new Triangle();
Triangle t2 = new Triangle();
t1.height=4;
t1.width=6;
t1.style="filled";
t2.height=8;
t2.width=12;
t2.style="outlined";
t1.showDim();
t1.showStyle();
System.out.println("Area ="+t1.area());
t2.showDim();
t2.showStyle();
System.out.println("Area ="+t2.area()); } }
```

Output
Height = 4 Width = 6
Triangle is filled
Area =12
Height = 8 Width = 12
Triangle is outlined
Area =48

The class Shapes contains two members namely height and width and a method showDim(). The class triangle inherits all the members and methods of the class Shape and it has its own member named style and methods named area() and showStyle().

Member Access and Inheritance

All the public members of the base class are inherited by the derived classes where as the members declared with the modifier private cannot be accessed by the derived class. The following example program will not be compiled

```
class Shapes {
private int height;
private int width;
void showDim() {
      System.out.println("Width = " +width + "Height ="+height); } }

class Triangle extends Shape {
Sting style;
int area() {
return width * height /2; // Error !Private members cant be accessed
} }
```

This program will not compile because width and height can not be inherited by the derived class since they are private.

Constructors and Inheritance

Both superclasses and subclasses can have their own constructors. The constructor of the superclass constructs the superclass portion of the object and that of the subclass constructs the subclass part. When only the subclass defines a constructor, it just constructs the object of the subclass. The superclass portion of the object is constructed by calling the default constructor.

```
class Shape{
      int height, width;
      void showDim() {
      System.out.println("Width = " +width + "Height ="+height); } }

class Triangle extends Shape {
     String style;
```

```
Triangle(String s, int h, int w) {
style=s;
height =h;
width=w; }
void showStyle(){
System.out.println("Triangle is " +style); }
int area()
{return(height*width/2); } }

class ConDemo {
public static void main(String args[]) {
Triangle t1 = new Triangle("filled", 4, 6);
Triangle t2 = new Triangle("outlined", 8,12);
t1.height=4;
t1.width=6;
t1.style="filled";
t2.height=8;
t2.width=12;
t2.style="outlined";
t1.showDim();
t1.showStyle();
System.out.println("Area ="+t1.area());
t2.showDim();
t2.showStyle();
System.out.println("Area ="+t2.area()); } }
```

Output
Height = 4 Width = 6
Triangle is filled
Area =12
Height = 8 Width = 12
Triangle is outlined
Area =48

The constructor Triangle() can be used to initialize objects of the derived class but it cannot be used to initialize the objects of the base class.

Use of super()

When both superclass and subclass define constructors, the process is bit complicated. The subclass constructor uses the keyword super to invoke the constructor of the superclass. The keyword super can be used only within the derived class and if included, it must always be the first statement executed inside a subclass constructor.

```
//To demonstrate super()
class Room {
int length, breadth;
Room(int x, int y) {
length=x;
breadth=y; }
int area() {
return(length*breadth); }}

class BedRoom extends Room{
int height;
BedRoom(int x, int y, int z) {
super(x,y);
height=z; }
int volume() {
return(length * breadth*height); } }
```

Output
Area =168
Volume =1680

```
class InhTest{
public static void main(String args[]) {
BedRoom r1= new BedRoom(14,12,10);
System.out.println("Area =", +r1.area());
System.out.println("Volume =", +r1.volume());     } }
```

Using super to Access Superclass Members

We can access the superclass member with the help of super key word. This has the following form:

super*.*member

Here member is either a method or an instance variable.

```
// using super to access member of superclass
class A {
 int i; }

class B extends A {
int i; // This i  belongs to class B which hides the i of A
B(int a, int b) {
super.i=a;
i=b; }
void show() {
System.out.println("i in superclass "+super.i);
System.out.println("i in subclass "+ i); } }

class UseSuper {
public static void main(String args[]) {
B ob1=new(1,2); B ob1=new B(1,2);
ob1.show(); } }
```

Output
i in superclass 1
i in subclass 2

Although variable i of B is hidden by variable i of A, ***super*** allows access to the i defined in the superclass. The keyword ***super*** can also be used to call methods that are hidden by a subclass.

Multilevel Inheritance

It is possible to build hierarchies that contain many layers of inheritance. This phenomenon is called multilevel inheritance. In this situation there will be two or more levels of inheritance. If there are classes A B and C, in such a way that B is the subclass of A and C is the subclass of B we call it as multilevel inheritance.

```
//Demonstrate Multilevel Inheritance
class Shape{
   int height, width;

   void showDim() {
   System.out.println("Width = " +width + "Height ="+height); }

Shape()
{ height=width=0; }
```

```
Shape(int x, int y)
{ height=x; width=y; }

void showDim()
Sustem.out.println("Height = "+height+" Width = "+width);}}

class Triangle extends Shape {
Triangle() {
super();
style="none"; }

class Triangle extends Shape {
Triangle(String s, int h, int w) {
super(h, w);
style=s;}
void showStyle(){
System.out.println("Triangle is " +style); }
int area()
{return(height*width/2); } }

class colTriangle extends Triangle {
String colr;
colTriangle(String s1, String s2, int x, int y)
{ super(s2, x,y);
  colr=s1; }
  void showColor(){
System.out.println("Color is " +colr);} }

class MulDemo {
public static void main(String args[]) {
colTriangle t1 = new Triangle("blue", "filled", 4, 6);
Triangle t2 = new Triangle("red", "outlined", 8,12);
t1.showDim();
t1.showStyle();
t1.showColor();
System.out.println("Area ="+t1.area());
t2.showDim();
t2.showStyle();
t1.showColor();
System.out.println("Area ="+t2.area()); } }
```

Output
Height = 4 Width = 6
Triangle is filled
Color is blue
Area =12
Height = 8 Width = 12
Triangle is outlined
Color is red
Area =48

Method Overriding

In a class hierarchy when a method in a subclass has the same return type and signature as a method in it's superclass, then the method in the subclass is said to override the method in the superclass. when a overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass**.** The version of the method defined by the superclass will be hidden.

//Demonstrate method overriding

```
class A {
int i,j;
A(int x, int y) {
i=x;
```

```
j=y; }
void show() {
System.out.println("Values of i and j are " +i+ "  " +j); } }

class B extends A {
int k;
B(int x, int y, int z) {
super(x,y);
k=z; }
void show() {
System.out.println(""k= "+k); } }

class overrideDemo{
public static void main(String args[]){
B ob1=new B(10,20,30);
Ob1.show() //refers t the function written in the subclass
} }
```

Output
k= 30

The show() in the subclass overrides the show() in the superclass. If we want to access the show() of superclass we can use the super() as follows:

```
B(int x, int y, int z) {
super(x,y);
k=z; }
void show() {
super();
System.out.println(""k= "+k); } }
```

Output
Values of i and j 10 20
k= 30

Final variables and methods

All methods and variables can be overridden by the subclasses. If we wish to prevent overriding the members if superclass we can declare them as final using the keyword ***final*** as modifier.

Example
final int size = 100;
final void showStatus()

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 of the class.

```
class A{
final void meth() {
System.out.println("This is a final method"); }

Class B extends A {
void meth() { // Error can't override!!! } }
```

Abstract Methods and Classes

If we modify a method as final we ensure that the method is not redefined in a subclass. Java allows us to do something that is exactly opposite to this. That is, we can indicate a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword ***abstract***.

Example:

```
abstract class Shape {
........
........
abstract void show() {
........  }   }
```

While using abstract classes, we must satisfy the following conditions:

- Abstract classes cannot be used to instantiate objects directly. For example:
  Shape s1=new Shape() is illegal because Shape is an abstract class according to the above example.
- The abstract methods of an abstract class must be redefined in its subclass
- We cannot declare abstract constructors or abstract static methods.

Multilevel Inheritance Example

```
class Shape
{
   int height, width;
   Shape()
   {
      height=width=0;
   }
   Shape(int x, int y)
   {
      height=x; width=y;
   }
   void showDim()
   {
      System.out.println("Height = "+height+"\nWidth = "+width);
   }
}

class Triangle extends Shape
{
   String style;
   Triangle(String s, int h, int w)
   {
      super(h, w);
      style=s;
   }
   void showStyle()
   {
      System.out.println("Triangle is  "+style);
   }
   int area()
   {
      return(height*width/2);
   }
}
```

```
class colTriangle extends Triangle
{
   String colr;
   colTriangle(String s1, String s2, int x, int y)
   {
      super(s2, x,y);
      colr=s1;
   }
   void showColor()
   {
      System.out.println("Color is "+colr);
   }
}

class Main
{
   public static void main(String args[])
   {
      colTriangle t1 = new colTriangle("blue", "filled", 4, 6);
      Triangle t2 = new Triangle("outlined", 8,12);
      t1.showDim();
      t1.showStyle();
      t1.showColor();
      System.out.println("Area ="+t1.area());
      t2.showDim();
      t2.showStyle();
      t1.showColor();
      System.out.println("Area ="+t2.area());
   }
}
```