UNIT II

METHODS:

- Methods are subroutines that manipulate the data defined by class and in some cases, control access to that data
- The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.
- Two of the components of a method declaration comprise the method signature—the method's name and the parameter types.
- Method declarations have six components, in order:
    1. Modifiers—such as public, private
    2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
    3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
    4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
    5. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.
- **General Form:**

```
type name(parameter-list) {

        // body of method

}
```

- The type of data returned by a method must be compatible with the return type specified by the method.
- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

```
return value;    //Here, value is the value returned.
```

Returning from a method:

- Two conditions can cause a method to return
    1. When the method's closing brace is encountered
    2. When a return statement is executed.
- There are 2 forms of return
    1. One for use in void methods (which do not return a value)
    2. Other for returning values
- In a void method, we can terminate a method by using the following form of return:

```
return;
```

- When this statement executes, program control returns to the caller, skipping any remaining code in the method.

  For ex:

```
void sample( ){
```

```
        for(int i=0; i<10; i++){
                if(i==5)
                        return;  //stop at 5
                System.out.println(i);
        }
}
```

- The for loop will only run from 0 to 5 because once i equals 5, the method returns.

Returning a Value:

- Methods return a value to the calling routing using this form of return:
  return value;
- This form of return can be used only with methods that have a non-void return type.

```
class Rectangle{
    int length,width;
    int area(int l, int w){
        length = l;
        width=w;
        int area=length*width;
        return area; //returns the area to the invoking method
    }
}
public class Method_return {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        int area = r.area(10, 5);
        System.out.println("The area of the rectangle is "+area);
    }
}
```

OUTPUT:

The area of the rectangle is 50

Using Parameters:

- Value passed to a method is called an argument. Variable that receives the argument is called a parameter.
- Parameters are declared inside the parentheses that follow the method's name.
  For ex:

```
class ckNum{
        //return true if x is even
        boolean isEven(int x){
                if((x%2)==0)  return true;
                else      return false;
        }
```

```
}
Class Demo{
        public static void main(String[ ] args){
                ckNum e=new ckNum();
                if(e.isEven(10))
                        System.out.println("10 is even");
                if(e.isEven(9))
                        System.out.println("9 is even");
        }
}
```

OUTPUT:

```
10 is even
```

- In the program isEven( ) is called twice, each time a different value is passed.
- When isEven( ) is called the first time, it is passed the value 10. Thus, when isEven( ) begins executing, the parameter x receives the value 10.
- In the second call, it is passed the value 9. Thus, when isEven( ) begins executing, the parameter x receives the value 9.

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through a special type of method called Constructor which is used to initialize the object
- A constructor initializes an object immediately upon creation

Rules for creating java constructor:

- It has the same name as the class in which it resides and is syntactically similar to a method.
- They have no explicit return type.

Types of java constructors:

- There are two types of constructors:
  - Default constructor (no-arg constructor)
  - Parameterized constructor

Default Constructor:

- Constructor which does not take any argument is called default constructor.

```
class Rectangle{
    int length,width;
    Rectangle(){  //default constructor
        length = 10;
        width = 5;
    }
    void area(){
        int area=length*width;
        System.out.println("The area of the rectangle is "+area);
    }
}
public class Default_Constructor {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area();
    }
}
```

Output:

The area of the rectangle is 50

- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor.
- However, once you define your own constructor, the default constructor is no longer used

Parameterized Constructors:

- Constructor which takes one or more arguments is called parameterized constructor.

```
class Rectangle{
    int length,width;
    Rectangle(int l, int w){  //parameterized constructor
        length = l;
        width = w;
    }
    void area(){
        int area=length*width;
        System.out.println("The area of the rectangle is "+area);
    }
}
public class Default_Constructor {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(10,5);
        r.area();
    }
}
```

Output:

The area of the rectangle is 50

- Each object is initialized as specified in the parameters to its constructor.
- The values of 10 and 5 are passed to Rectangle() constructor when new creates the object.
- r's copy of length and width will contain the values 10 and 5 respectively.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, you simply define the finalize( ) method.
- The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( ) method, actions that must be performed before an object is destroyed will be specified.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- The keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class
- It is important to understand that finalize( ) is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—finalize( ) will be executed

Ex:

```
class Myclass{
    protected void finalize(){
        System.out.println("This method is called "
                + "automatically when there is a need");
    }
    void generate(int i){
        Myclass obj = new Myclass();
        System.out.println("Object: "+i++);
    }
}
public class Constructor {
    public static void main(String[] args) {
        Myclass obj = new Myclass();
        for(int count =1; count<900000; count++){
            obj.generate(count);
        }
    }
}
```

Output:

```
Object: 383693
Object: 383694
This method is called automatically when there is a need
This method is called automatically when there is a need
Object: 383695
Object: 383696
Object: 383697
Object: 383698
Object: 383699
This method is called automatically when there is a need
This method is called automatically when there is a need
Object: 383700
Object: 383701
Object: 383702
Object: 383703
```

The this Keyword

- The this keyword can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked

```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

- The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class's instance variables
- When a local variable has the same name as an instance variable, the local variable hides the instance variable
- For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

- Use this to overcome the instance variable hiding

The new Operator

- new operator has this general form:
  class_var = new class_name(arg_list);
- class_var is a variable of the class type being created.
- The class_name is the name of the class that is being instantiated
- The class_name followed by a parenthesized argument list specifies the constructor for the class.
- If a class does not define its own constructor, new will use the default constructor supplied by Java
- The new can be used to create an object of any class type.
- It returns a reference to the newly created object, which is assigned to class-var

Controlling Access to Class Members

- As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control.
- How a member can be accessed is determined by the access specifier that modifies its declaration.
- Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.)
- Member access control is achieved through the use of three access specifiers/modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
  For example,

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}

class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

- Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**

Pass Objects to Methods

- So far, we have only been using primitive types such as **int** as parameters to methods. However, it is both correct and common to pass objects to methods.
  For example,

```
class Object_Method{
    int i,j;
    Object_Method(int i, int j){ //instance variable hiding
        this.i = i;
        this.j = j;
    }
    void check_equality(Object_Method obj){
        if(obj.i==i && obj.j==j)
            System.out.println("Objects are equal");
        else
            System.out.println("Objects are not equal");
    }
}
public class Pass_Objects {
    public static void main(String[] args) {
        Object_Method obj1 = new Object_Method(10,5);
        Object_Method obj2 = new Object_Method(10,5);
        Object_Method obj3 = new Object_Method(20,5);
        obj1.check_equality(obj2); // obj1 and obj2 are equal
        obj1.check_equality(obj3); // obj1 and obj3 are not equal
    }
}
```

Output:

Objects are equal

Objects are not equal

- The **check_equality( )** method inside **Object_Method** compares two objects for equality and displays the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method displays objects are equal. Otherwise, it displays objects are not equal

How Arguments are Passed

- There are two ways in which an argument can be passed to a subroutine/method
  1. **Call-by-value:**
     - Copies the value of an argument into the formal parameter of the subroutine.
     - Therefore, changes made to the parameter of the subroutine have no effect on the argument.
  2. **Call-by-reference:**

- Reference to an argument (not the value of the argument) is passed to the parameter
- Inside the subroutine, this reference is used to access the actual argument specified in the call. Changes made to the parameter will affect the argument used to call the subroutine.

- Example for **call-by-value**

```
class Test{
    void display(int i, int j){
        i = i+j;
        j= -j;
    }
}
public class Call_by_value {
    public static void main(String[] args) {
        Test obj = new Test();
        int a=10, b=20;
        System.out.println("Values of a and b before call "+a+" "+b);
        obj.display(a,b);
        System.out.println("Values of a and b after call "+a+" "+b);
    }
}
```

Output:

Values of a and b before call 10 20

Values of a and b after call 10 20

- The operations that occur inside **display( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and -20.

- Example for **call-by-reference**

```
class Test{
    int a,b;
    void display(Test obj){
        obj.a = this.a*2;
        obj.b = this.b*2;
    }
}
public class Call_by_reference {
    public static void main(String[] args) {
        Test o = new Test();
        o.a=10;
        o.b=20;
        System.out.println("The values of a and b before call "+o.a+" "+o.b);
        o.display(o);
        System.out.println("The values of a and b after call "+o.a+" "+o.b);
    }
}
```

Output:

Values of a and b before call 10 20

Values of a and b after call 20 40

- In this case, the actions inside **display( )** have affected and changed the actual object used as an argument, as copy of the **reference (not the value)** is passed as the argument. Hence, the reference variables o and obj are referring to the same object.

Returning Objects

- A method can return any type of data, including class types that you create.
- For example

```
class Ret_obj{
    int a,b;
    Ret_obj display(int i,int j){
       Ret_obj temp = new Ret_obj();
       temp.a=i;
       temp.b=j;
       return temp;  // returna an object of Ret_obj class
    }
}
public class Return_Object {
    public static void main(String[] args) {
        Ret_obj o = new Ret_obj();
        Ret_obj result = o.display(10, 20);
        System.out.println("The result is "+result.a+" "+result.b);
    }
}
```

Output:

The result is 10 20

- When **display( )** method is invoked, it returns an object of the class Ret_obj class

Method Overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*.
- Method overloading is one of the ways that **Java supports polymorphism**
- 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**.
- 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.

Example,

```
class Overload{
    int i;
    void initialize(){
        i=0;
        System.out.println("This method doesn't take any parameters");
    }
    void initialize(int a){   //overload initialize() for an integer parameter
        i=a;
        System.out.println("This method takes one integer parameter");
    }
    void initialize(int a, int b){ //overload initialize() for two integer parameters
        i=a;
        int j=b;
        System.out.println("This method takes two integer parameters");
    }
    double initialize(double a){ //overload initialize() for a double parameter
        double b=a;
        b*=a;
        System.out.println("This method takes one double parameter");
        return b;
    }

}

public class Method_Overloading {
    public static void main(String[] args) {
        Overload o = new Overload();
        //call all versions of initialize()
        o.initialize();
        o.initialize(10);
        o.initialize(10, 20);
        double result = o.initialize(12.5);
        System.out.println("The result is "+result);
    }
}
```

Output:

This method doesn't take any parameters

This method takes one integer parameter

This method takes two integer parameters

This method takes one double parameter

The result is 156.25

- **initialize( )** is overloaded four times

- In some cases, Java's automatic type conversions can play a role in overload resolution.

Example,

```
class Overload{
    int i;
    void initialize(int a, int b){ //overload initialize() for two integer parameters
        i=a;
        int j=b;
        System.out.println("This method takes two integer parameters");
    }
    double initialize(double a){ //overload initialize() for a double parameter
        double b=a;
        b*=a;
        System.out.println("This method takes one double parameter");
        return b;
    }

}
public class Method_Overloading {
    public static void main(String[] args) {
        Overload o = new Overload();
        o.initialize(10); // this invokes initialize(double)
        o.initialize(10, 20);
        double result = o.initialize(12.5f); //this invokes initialize(double)
        System.out.println("The result is "+result);
    }
}
```

Output:

```
This method takes one double parameter

This method takes two integer parameters

This method takes one double parameter

The result is 156.25
```

- This version of **Overload** does not define **initialize(int) and initialize(float)**. Therefore, when **initialize( )** is called with an integer / floating argument inside **Overload**, no matching method is found.
- Java can automatically convert an integer / a floating value into a **double**
- Java will employ its automatic type conversions only if no exact match is found.
- Overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Example,

```
// One ovlDemo(int) is OK.
void ovlDemo(int a) {
  System.out.println("One parameter: " + a);
}

/* Error! Two ovlDemo(int)s are not OK even though
   return types differ.
*/
int ovlDemo(int a) {
  System.out.println("One parameter: " + a);
  return a * a;
}
```

Return types cannot be used to differentiate overloaded methods.

Overloading Constructors

- Like methods, constructors can also be overloaded

For example,

```
class Cons_Overl{
    int i;
    Cons_Overl(){
        i=0;
        System.out.println("Its a default constructor");
    }
    Cons_Overl(int a){
        i=a;
        System.out.println("Its an integer parameterized constructor");
    }
    Cons_Overl(double a){
        i= (int)a;
        System.out.println("Its a double parameterized constructor");
    }
    Cons_Overl(int a, int b){
        i= a*b;
        System.out.println("Its a two integers parameterized constructor");
    }
}
public class Constructor_Overload {
    public static void main(String[] args) {
        Cons_Overl c1 = new Cons_Overl();
        Cons_Overl c2 = new Cons_Overl(10);
        Cons_Overl c3 = new Cons_Overl(10.2);
        Cons_Overl c4 = new Cons_Overl(10,20);
    }
}
```

Output:

Its a default constructor

Its an integer parameterized constructor

Its a double parameterized constructor

Its a two integers parameterized constructor

- **Cons_Overl( )** is overloaded in 4 ways, each constructing an object differently.
- The proper constructor is called based on the parameters specified when **new** is executed.
- Constructor overloading also allows one object to initialize another

For example,

//Initialize one object with another

```
class Copy_object{
    int a;
    Copy_object(int i){
        a=i;
    }
    Copy_object(Copy_object o){
        a = o.a;
    }
    void show(){
        System.out.println("The value of a is "+a);
    }
}
public class Constructor_Object {
    public static void main(String[] args) {
        Copy_object obj1 = new Copy_object(10);
        obj1.show();
        Copy_object obj2 = new Copy_object(obj1);
        obj2.show();

    }
}
```

Output:

The value of a is 10 //value of a of obj1

The value of a is 10 //value of a of obj2 initialized from obj1

- In the above example, constructor **Copy_object()** receives obj1 as parameter and initializes obj2.

For Your Knowledge and Understanding

Understanding static

- 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.
- You can declare both methods and variables to be 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.

static Variables

- 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.
- General form of accessing static variable is

  Classname.static_variablename;
- For example,

  MyTimer.count = 10;
- Example 1 shows differences between a static variable and an instance variable

```
// Use a static variable.
class StaticDemo {
  int x; // a normal instance variable
  static int y; // a static variable

  // Return the sum of the instance variable x
```

There is one copy of **y** for all objects to share.

```
  // and the static variable y.
  int sum() {
    return x + y;
  }
}

class SDemo {
  public static void main(String[] args) {
    StaticDemo ob1 = new StaticDemo();
    StaticDemo ob2 = new StaticDemo();

    // Each object has its own copy of an instance variable.
    ob1.x = 10;
    ob2.x = 20;
    System.out.println("Of course, ob1.x and ob2.x " +
                       "are independent.");
    System.out.println("ob1.x: " + ob1.x +
                       "\nob2.x: " + ob2.x);
    System.out.println();

    // Each object shares one copy of a static variable.
    System.out.println("The static variable y is shared.");
    StaticDemo.y = 19;
    System.out.println("Set StaticDemo.y to 19.");

    System.out.println("ob1.sum(): " + ob1.sum());
    System.out.println("ob2.sum(): " + ob2.sum());
    System.out.println();

    StaticDemo.y = 100;
    System.out.println("Change StaticDemo.y to 100");
    System.out.println("ob1.sum(): " + ob1.sum());
    System.out.println("ob2.sum(): " + ob2.sum());
    System.out.println();
  }
}
```

The output from the program is shown here:

```
Of course, ob1.x and ob2.x are independent.
ob1.x: 10
ob2.x: 20

The static variable y is shared.
Set StaticDemo.y to 19.
ob1.sum(): 29
ob2.sum(): 39

Change StaticDemo.y to 100
ob1.sum(): 110
ob2.sum(): 120
```

- **y** is accessed through its class name

Example 2

```
// Count instances.
class MyClass {
  // This static variable will be incremented each
  // time a MyClass object is made.
  static int count = 0;

  MyClass() {
    count++; // increment the count
  }
}

class UseStatic {
  public static void main(String[] args) {
    for(int i=0; i < 3; i++) {
      MyClass obj = new MyClass();
      System.out.println("Number of objects created: " + MyClass.count);
    }
  }
}
```

The output is shown here:

```
Number of objects created: 1
Number of objects created: 2
Number of objects created: 3
```

static Methods

- Methods declared as **static** are, essentially, global methods.
- They are called independently of any object. A **static** method is called through its class name
- For example,

// use a static method.

class StaticMeth {

static int val = 1024; // a static variable

```
  // A static method.
  static int valDiv2() {
    return val/2;
  }
}

class SDemo2 {
  public static void main(String[] args) {

    System.out.println("val is " + StaticMeth.val);
    System.out.println("StaticMeth.valDiv2(): " +
                       StaticMeth.valDiv2());

    StaticMeth.val = 4;
    System.out.println("val is " + StaticMeth.val);
    System.out.println("StaticMeth.valDiv2(): " +
                       StaticMeth.valDiv2());
  }
}
```

The output is shown here:

```
val is 1024
StaticMeth.valDiv2(): 512
val is 4
StaticMeth.valDiv2(): 2
```

- Methods declared as **static** have several restrictions:
  - They can only call other static methods.
  - They must only access static data.
  - They cannot refer to **this** or **super** in any way.

For example, in the following class, the **static** method **valDivDenom( )** is illegal.

```
class StaticError {
  int denom = 3; // a normal instance variable
  static int val = 1024; // a static variable

  /* Error! Can't access a non-static variable
     from within a static method. */
  static int valDivDenom() {


    return val/denom; // won't compile!
  }
}
```

Here, **denom** is a normal instance variable that cannot be accessed within a **static** method.

static Blocks

- Java allows to declare a **static** block.
- 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

For example,

```
// Use a static block
class StaticBlock {
  static double rootOf2;
  static double rootOf3;

  static {                                   // This block is executed when the class is loaded
    System.out.println("Inside static block.");
    rootOf2 = Math.sqrt(2.0);
    rootOf3 = Math.sqrt(3.0);
  }

  StaticBlock(String msg) {
    System.out.println(msg);
  }
}

class SDemo3 {
  public static void main(String[] args) {
    StaticBlock ob = new StaticBlock("Inside Constructor");

    System.out.println("Square root of 2 is " +
                       StaticBlock.rootOf2);
    System.out.println("Square root of 3 is " +
                       StaticBlock.rootOf3);

  }
}
```

The output is shown here:

```
Inside static block.
Inside Constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772
```

As you can see, the **static** block is executed before any objects are constructed.