

# Chapter Two

## Objects and Classes

# Defining a class

- From chapter one it is recalled that a class is a template that defines the form of an object.
- It specifies both the data and the code that will operate on that data.
- Java uses a class specification to construct objects.
  - Objects are instances of a class.
- One other point: recall that the methods and variables that constitute a class are called members of the class.
- The data members are also referred to as instance variables.

## The General Form of a Class

- When you define a class, you declare its exact form and nature.
- You do this by specifying the instance variables that it contains and the methods that operate on them.
- Although very simple classes might contain only methods or only instance variables, most real-world classes contain both.
- A class is created by using the keyword **class**.

- The general form of a **class** definition is shown here:

```
class classname {  
    // declare instance variables  
    type var1;  
    type var2;  
    // ...  
    type varN;  
    // declare methods  
    type method1(parameters) {  
        // body of method  
    }  
    type method2(parameters) {  
        // body of method  
    }  
    // ...  
    type methodN(parameters) {  
        // body of method  
    }  
}
```

# Example: Simple class

```
class Vehicle {  
    int passengers; // number of passengers  
    int fuelcap; // fuel capacity in gallons  
    int mpg; // fuel consumption in miles per  
    gallon  
    public static void main(String args[]) {  
        Vehicle minivan = new Vehicle();  
        int range; //local variable  
        // assign values to fields in minivan  
        minivan.passengers = 7;  
        minivan.fuelcap = 16;  
        minivan.mpg = 21;  
        // compute the range assuming a full tank of  
        gas  
        range = minivan.fuelcap * minivan.mpg;  
        System.out.println("Minivan can carry " +  
            minivan.passengers + " with a range of " +  
            range);  
    }  
}
```

# Instantiating and using objects

- An object is comprised of data and operations that manipulate these data. To define an object we use the following format.

**Class Name object\_identifier= *new* Class Name ()**

- If there is constructor with an argument we have to pass during object instantiation otherwise we have to leave the bracket () empty.
- From the example above

**Vehicle minivan = new Vehicle();**

- is a typical example of defining object.

# Constructors

- In Java, constructor is a block of codes similar to method.
- It is called when an instance of object is created and memory is allocated for the object.
- It is a special type of method which is used to initialize the object.

– **Note:**

- *It is called constructor because it constructs the values at the time of object creation.*
- *It is not necessary to write a constructor for a class.*
- *Java compiler creates a default constructor if your class doesn't have any.*

## Rules for creating java constructor

- There are basically two rules defined for the constructor.
  - Constructor name must be same as its class name
  - Constructor must have no explicit return type
- There are two types of constructors in java:
  - Default constructor (no-arg constructor)
  - Parameterized constructor



# Java Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- *Syntax of default constructor:*

```
<class_name>(){} 
```

- **Example of default constructor**
- In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
class Bike1{  
    Bike1(){System.out.println("B  
ike is created");}  
    public static void main(String  
args[]){  
        Bike1 b=new Bike1();  
    }  
}
```

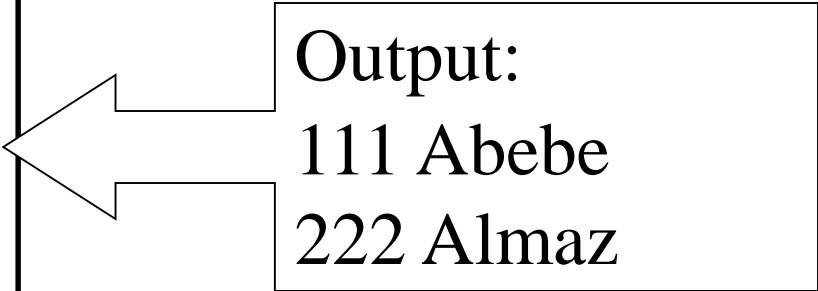


**Output: Bike is  
created**

# Java parameterized constructor

- A constructor which has a specific number of parameters is called parameterized constructor.
- Parameterized constructor is used to provide different values to the distinct objects.
- **Example of parameterized constructor**
  - In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student4{
    int id;
    String name;
    Student4(int i,String n){
        id = i;
        name = n;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
        Student4 s1 = new Student4(111,"Abebe");
        Student4 s2 = new Student4(222,"Almaz");
        s1.display();
        s2.display();
    }
}
```



Output:  
111 Abebe  
222 Almaz

# Difference between constructor and method in java

- There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

## Multiple classes

- A Java application is implemented by one or more classes.
- Small applications can be accommodated by a single class, but larger applications often require multiple classes.
- In that case one of the classes is designated as the main class and contains the `main()` entry-point method.

- Typical structure of a Java application with multiple classes

```
class A
{
}

class B
{
}

class C
{
    public static void main(String[] args)
    {
        System.out.println("Application C entry point");
    }
}
```

# Example:

```
class Account {  
    // Data Members  
    private String ownerName;  
    private double balance;  
    //Constructor  
    public Account( ) {
```



```
ownerName = "Unassigned";
balance = 0.0;
}
//Adds the passed amount to the balance
public void add(double amt) {
    balance = balance + amt;
}
//Deducts the passed amount from the balance
public void deduct(double amt) {
    balance = balance - amt;
}
//Returns the current balance of this account
public double getCurrentBalance( ) {
    return balance;
}
//Returns the name of this account's owner
public String getOwnerName( ) {
    return ownerName;
}
//Sets the initial balance of this account
public void setInitialBalance(double bal) {
    balance = bal;
}
```

```
//Assigns the name of this account's owner
public void setOwnerName(String name) {
    ownerName = name;
}
}

class Bicycle {
    // Data Member
    private String ownerName;
    //Constructor: Initializes the data member
    public Bicycle( ) {
        ownerName = "Unknown";
    }
    //Returns the name of this bicycle's owner
    public String getOwnerName( ) {
        return ownerName;
    }
    //Assigns the name of this bicycle's owner
```

```
public void setOwnerName(String name) {  
    ownerName = name;  
}  
class SecondMain {  
    //This sample program uses both the Bicycle and Account classes  
    public static void main(String[] args) {  
        Bicycle bike;  
        Account acct;  
        String myName = "Jon Java";  
        bike = new Bicycle( );  
        bike.setOwnerName(myName);  
        acct = new Account( );  
        acct.setOwnerName(myName);  
        acct.setInitialBalance(250.00);  
        acct.add(25.00);  
        acct.deduct(50);  
        //Output some information  
        System.out.println(bike.getOwnerName() + " owns a bicycle and");  
        System.out.println("has $ " + acct.getCurrentBalance() +  
            " left in the bank");  
    }  
}
```

# Access modifiers

- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - private
  - default
  - protected
  - public

# private access modifier

- The private access modifier is accessible only within class.
- **Example:** In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

# Role of Private Constructor

- If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{  
    private A(){}//private constructor  
    void msg(){System.out.println("Hello java");}  
}  
  
public class Simple{  
    public static void main(String args[]){  
        A obj=new A();//Compile Time Error  
    } }  

```

# default access modifier

- If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.
- **Example** : In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
    void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

- In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.



# protected access modifier

- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- **Example:** In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

- `//save by A.java`
- `package pack;`
- `public class A{`
- `protected void msg(){System.out.println("Hello");}`
- `}`
- `//save by B.java`
- `package mypack;`
- `import pack.*;`
- 
- `class B extends A{`
- `public static void main(String args[]){`
- `B obj = new B();`
- `obj.msg();`
- `}`
- `}`

# public access modifier

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
- **Example of public access modifier**

```
//save by A.java
```

```
package pack;
```

```
public class A{
```

```
public void msg(){System.out.println("Hello");}
```

```
}
```

```
//save by B.java
```

```
package mypack;
```

```
import pack.*;
```

```
class B{
```

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

```
    A obj = new A();
```

```
    obj.msg();
```

```
}}
```

# Table summary of access modifiers

<b>Access Modifier</b>	<b>within class</b>	<b>within package</b>	<b>outside package subclass only</b>	<b>outside by package</b>
<b>Private</b>	YES	NO	NO	NO
<b>Default</b>	YES	YES	NO	NO
<b>Protected</b>	YES	YES	YES	NO
<b>Public</b>	YES	YES	YES	YES

## Pass Objects to Methods

- As with many other primitive data types a method argument can be an object.
- **Example:** The following program defines a class called **Block** that stores the dimensions of a three-dimensional block and passes objects to the two methods in the class.

**Example:**

// Objects can be passed to methods.

```
class Block {
```

```
int a, b, c;
```

```
int volume;
```

```
Block(int i, int j, int k) {
```

```
  a = i;
```

```
  b = j;
```

```
  c = k;
```

```
  volume = a * b * c;
```

```
}
```

// Return true if ob defines same block.

```
boolean sameBlock(Block ob) {
```

```
  if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
```

```
  else return false;
```

```
}
```

// Return true if ob has same volume.

```
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 dimensions as ob2:  
            " +  
            ob1.sameBlock(ob2));  
        System.out.println("ob1 is same dimensions  
            as ob3: " +  
            ob1.sameBlock(ob3));  
        System.out.println("ob1 same volume as ob3: " +  
            ob1.sameVolume(ob3));  
    }  
}
```



# Returning an Object from a Method

- Like other return type in java we can also have a method returning an object.
- Example: The following Fraction class uses method return objects.

### Example

```
class Fraction {  
    private int numerator;  
    private int denominator;  
    public Fraction(int num, int denom) {  
        setNumerator(num);  
        setDenominator(denom);  
    }  
    public int getDenominator( ) {  
        return denominator;  
    }  
    public int getNumerator( ) {  
        return numerator;  
    }  
    public void setDenominator(int denom) {  
        if (denom == 0) {  
            //Fatal error  
            System.err.println("Fatal Error");  
            System.exit(1);  
        }  
        denominator = denom;  
    }  
}
```

```
public void setNumerator(int num) {  
    numerator = num;  
}  
  
public String toString( ) {  
    return getNumerator() + "/" + getDenominator();  
}  
  
public Fraction simplify( ) {  
    int num = getNumerator();  
    int denom = getDenominator();  
    int gcd = gcd(num, denom);  
    Fraction simp = new Fraction(num/gcd, denom/gcd);  
    return simp;  
}
```

```
public int gcd (int m, int n) {  
    //assume m, n >= 1  
    int last = Math.min(m, n);  
    int gcd;  
    int i = 1;  
    while (i <= last) {  
        if (m % i == 0 && n % i == 0) {  
            gcd = i;  
        }  
        i++;  
    }  
    return gcd;  
    }}
```

```
Public TestFrac{  
Public static void main(String args[]){  
Fraction f1, f2;  
f1 = new Fraction(24, 36);  
f2 = f1.simplify( );  
System.out.println(f1.toString() + "can be  
    reduced to " +  
f2.toString());  
}}
```

# The Reserved Word *this*

- The reserved word *this* is called a *self-referencing pointer* because it is used to refer to the receiving object of a message from within this object's method.
- Let's start with the add method that adds two fractions:

```
public Fraction add( Fraction frac) {  
    int a, b, c, d;  
    Fraction sum;  
    a = this.getNumerator(); //get the receiving  
    b = this.getDenominator(); //object's num and denom  
    c = frac.getNumerator(); //get frac's num  
    d = frac.getDenominator(); //and denom  
    sum = new Fraction(a*d + b*c, b*d);  
    return sum;  
}
```

- Let's first look at how this add method is used. The following code adds two
- fractions f1 and f2 and assigns the sum to f3:
- Fraction f1, f2, f3;
- f1 = **new** Fraction(1, 2);
- f2 = **new** Fraction(1, 4);
- f3 = f1.add(f2);
- System.out.println("Sum of " + f1.toString() + " and " +
- f2.toString() + " is " +
- f3.simplify().toString() );
- This code, when executed, will produce the following output:
- Sum of 1/2 and 1/4 is 6/8
- In the statement
- f3 = f1.add(f2);
- We are calling the add method of f1 and passing the argument f2. So in this case, the receiving object is f1.
- Because f2 is also a Fraction object, we can write the statement as



- `f3 = f2.add(f1);`
- and get the same result (since the operation is addition). In this case, the receiving object is `f2`, and the argument is `f1`.
- The use of the reserved word `this` is actually optional. If we do not include it explicitly, then the compiler will insert the reserved word for us. For example, if we write
- **`class Sample {`**
- **`public void m1( ) {`**
- `...`
- `}`
- **`public void m2( ) {`**
- `m1();`
- `}}`
- then the compiler will interpret the definition as
- **`class Sample {`**
- **`public void m1( ) {`**
- `...`
- `}`
- **`public void m2( ) {`**
- **`this.m1();`**
- **`}}`**

- The methods for the other three arithmetic operations are defined in a similar manner:

```
public Fraction divide(Fraction frac) {  
    int a, b, c, d;  
    Fraction quotient;  
    a = this.getNumerator();  
    b = this.getDenominator();  
    c = frac.getNumerator();  
    d = frac.getDenominator();  
    quotient = new Fraction(a*d, b*c);  
    return quotient;  
}
```

```
public Fraction multiply(Fraction frac) {  
    int a, b, c, d;  
    Fraction product;  
    a = this.getNumerator();  
    b = this.getDenominator();  
    c = frac.getNumerator();  
    d = frac.getDenominator();  
    product = new Fraction(a*c, b*d);  
    return product;  
}
```

```
public Fraction subtract(Fraction frac) {  
    int a, b, c, d;  
    Fraction diff;  
    a = this.getNumerator();  
    b = this.getDenominator();  
    c = frac.getNumerator();  
    d = frac.getDenominator();  
    diff = new Fraction(a*d - b*c, b*d);  
    return diff;}  

```

# Another Use of this

- Consider the following class declaration:
- **class** MusicCD {
- **private** String artist;
- **private** String title;
- **private** String id;
- **public** MusicCD(String name1, String name2) {
- artist = name1;
- title = name2;
- id = artist.substring(0,2) + "-" +
- title.substring(0,9);
- }
- ...
- }

- The constructor has two String parameters, one for the artist and another for the title.
- An id for a MusicCD object is set to be the first two letters of the artist name followed by a hyphen and the first nine letters of the title.
- Now, consider what happens if we include (say, inadvertently) a local declaration for the identifier id like this:

```
public MusicCD(String name1, String name2) { String id;  
artist = name1;  
title = name2;  
id = artist.substring(0,2) + "-" +  
title.substring(0,9);  
}
```

- Because there is a matching local declaration for `id`, the identifier refers to the local variable, not to the third data member anymore.

- It would actually be more meaningful to use the conflicting identifiers artist and title. To do so, we rewrite the method by using the reserved word this as follows.

```
public MusicCD(String artist, String title) {  
this.artist = artist;  
this.title = title;  
id = artist.substring(0,2) + "-" +  
title.substring(0,9);  
}
```



- ***Rules for associating an identifier to a local variable, a parameter, and a data member***
  1. If there's a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
  2. Otherwise, if there's a matching data member declaration, then the identifier refers to the data member.
  3. Otherwise, it is an error because there's no matching declaration.

## Overloaded Methods and Constructors

- Multiple methods can share the same name as long as one of the following rules is met:
  1. They have a different number of parameters.
  2. The parameters are of different data types when the number of parameters is the same.
    - This methods with same name are called overloaded methods.

- More formally, we say two methods can be overloaded if they do not have the same signature.
  - The ***method signature*** refers to the name of the method and the number and types of its parameters.
- Two methods cannot be overloaded just by the different return types because two such methods would have the same signature.
- For example, the following two methods cannot be defined in the same class:

```
public double getInfo(String item) { ... }  
public int getInfo(String item) { ... }
```

- **Example: in our fraction class in the previous topics example we can add a second add method that helps add an integer.**

```
public Fraction add(int number) {  
    Fraction frac = new Fraction(number, 1);  
    Fraction sum = add(frac); //calls the first  
    add method  
    return sum;  
}
```

```
public Fraction add( Fraction frac)  
{  
    int a, b, c, d;  
    Fraction sum;  
    a = this.getNumerator(); //get the  
    receiving  
    b = this.getDenominator();  
    //object's num and denom  
    c = frac.getNumerator(); //get  
    frac's num  
    d = frac.getDenominator(); //and  
    denom  
    sum = new Fraction(a*d + b*c,  
    b*d);  
    return sum;  
}
```

# Constructor Overloading

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

# Example of Constructor Overloading

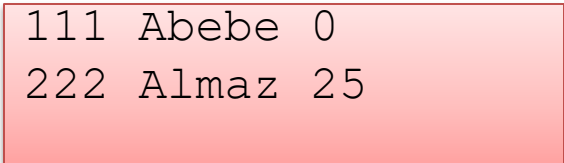
```
class Student5{
    int id,age;
    String name;

    Student5(int i,String n){
        id = i;
        name = n;
    }
    Student5(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }

    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
        Student5 s1 = new Student5(111,"Abebe");
        Student5 s2 = new Student5(222,"Almaz",25);
        s1.display();
        s2.display(); }}

```



```
111 Abebe 0
222 Almaz 25
```

## Calling a Constructor From Another Constructor by Using this

- The last use of the reserved word ***this*** is to call a constructor from another constructor of the same class.
- Here's how we can rewrite the four constructors of the Fraction class by using the reserved word this:

```
public Fraction( ) { //creates 0/1
this(0, 1);
}
public Fraction(int number) { //creates number/1
this(number, 1);
}
public Fraction(Fraction frac) { //copy constructor
this( frac.getNumerator(),frac.getDenominator() );
}
public Fraction(int num, int denom) {
    setNumerator(num);
    setDenominator(denom);
}
```



- The syntax for calling a constructor from another constructor of the same class is

**this( <parameter-list> );**

- The constructor that matches the parameter list will be called.
- We can add more statements after the this statement in a constructor, but not before it.
- In other words, the call to this in a constructor must be the first statement of the constructor.

# Class variables and methods

- The **static keyword** in java is used for memory management mainly.
- We can apply java static keyword with variables, methods, blocks and nested class.
- The static keyword belongs to the class than instance of the class.

The static can be:

- 1) variable (also known as class variable)
- 2) method (also known as class method)

# 1) Java static variable

- If you declare any variable as static, it is known static variable.
- The static variable can be used to refer the common property of all objects (that is not unique for each object)
  - e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.
- ***Advantage of static variable***
  - It makes your program **memory efficient** (i.e it saves memory).

# Understanding problem without static variable

- Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.
- All student have its unique rollno and name so instance data member is good.
- Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

```
class Student8{
    int rollno;
    String name;
    static String college ="ITS";

    Student8(int r,String n){    rollno = r;
    name = n;
    }
    void display (){System.out.println(rollno+" "+name+" "+college);}

    public static void main(String args[]){
        Student8 s1 = new Student8(111,"Abebe");
        Student8 s2 = new Student8(222,"Almaz");

        s1.display();
        s2.display();
    }
}
```

## 2) Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

```
class Student9{  
    int rollno;  
    String name;  
    static String college = "ITS";  
        static void change(){  
college = "BBDIT";  
    }  
  
    Student9(int r, String n){  
        rollno = r;  
        name = n;  
    }  
}
```

```
void display (){System.out.println(rollno+" "+name+" "+college);}
```

```
public static void main(String args[]){  
Student9.change();
```

```
Student9 s1 = new Student9 (111,"Almaz");  
Student9 s2 = new Student9 (222,"Abebe");  
Student9 s3 = new Student9 (333,"kebede");
```

```
s1.display();  
s2.display();  
s3.display();  
}
```

```
}
```



# Enumerated Types

- The Enum in Java is a data type which contains a fixed set of constants.
- It can be used for:
  - days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) ,
  - directions (NORTH, SOUTH, EAST, and WEST),
  - season (SPRING, SUMMER, WINTER, and AUTUMN or FALL),
  - colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.
- According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly.

- We can define an enum either inside the class or outside the class.
- Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces.
- We can have fields, constructors, methods, and main methods in Java enum.
- **Java Enumerated types**
  - Enum improves type safety
  - Enum can be easily used in switch
  - Enum can be traversed
  - Enum can have fields, constructors and methods
  - Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

# Example 1

- **class** EnumExample1{
- `//defining the enum inside the class`
- **public enum** Season { WINTER, SPRING, SUMMER, FALL }
- `//main method`
- **public static void** main(String[] args) {
- `//traversing the enum`
- **for** (Season s : Season.values())
- `System.out.println(s);`
- `}}`

**Output :**  
WINTER  
SPRING  
SUMMER  
FALL

## Exampe2

- `class EnumExample1{`
- `//defining enum within class`
- `public enum Season { WINTER, SPRING, SUMMER, FALL }`
- `//creating the main method`
- `public static void main(String[] args) {`
- `//printing all enum`
- `for (Season s : Season.values()){`
- `System.out.println(s);`
- `}`
- `System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));`
- `System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());`
- `System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());`
- 
- `}}`

### **Output :**

WINTER

SPRING

SUMMER

FALL

Value of WINTER is: WINTER

Index of WINTER is: 0

Index of SUMMER is: 2

# Defining Java Enum

- The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional. For example:

```
1.enum Season { WINTER, SPRING, SUMMER, FALL }
LL }
```

```
2.class EnumExample2{
```

```
3.public static void main(String[] args) {
```

```
4.Season s=Season.WINTER;
```

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

```
6.}}
```

```
1.class EnumExample3{
```

```
2.enum Season { WINTER, SPRING, SUMMER, FALL; }//
semicolon(;) is optional here
```

```
3.public static void main(String[] args) {
```

```
4.Season s=Season.WINTER;//enum type is required to access WINTER
```

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

```
6.}}
```

# Example 3

- **class** EnumExample4{
- **enum** Season{
- WINTER(5), SPRING(10), SUMMER(15), FALL(20);
- 
- **private int** value;
- **private** Season(**int** value){
- **this.value**=value;
- }
- }
- **public static void** main(String args[]){
- **for** (Season s : Season.values())
- System.out.println(s+" "+s.value);
- 
- }}

**Output :**  
WINTER 5  
SPRING 10  
SUMMER 15  
FALL 20

# Encapsulation

- **Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.
- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.
- To achieve encapsulation in Java –
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.

- Benefits of Encapsulation

- **Increased Flexibility**: The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- **Reusability**: Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy**: Encapsulated code is easy to test for unit testing.



# Comparing and Identifying Objects

- In java both == and equals() method is used to check the equality of two variables or objects.
- Following are the important differences between == and equals() method
- ==
  - == is an operator.
  - == should be used during reference comparison. == checks if both references points to same location or not.

- Equal()
  - equals() is a method of Object class.
  - equals() method should be used for content comparison. equals() method evaluates the content to check the equality.
  - equals() method if not present and Object.equals() method is utilized, otherwise it can be overridden.

- public class compare {
- 
- public static void main(String args[]) {
- String s1 = new String("Hello world");
- String s2 = new String("Hello world");
- //Reference comparison
- System.out.println(s1 == s2); //false
- //Content comparison
- System.out.println(s1.equals(s2)); //true
- // integer-type
- System.out.println(10 == 10); //true
- // char-type
- System.out.println('a' == 'a'); //true
- }
- }