Introducing Classes, Objects, and Methods

1. Class Fundamentals

Una clase es una plantilla que define la forma de un objeto. Especifica los datos y el código que operará en esos datos. Java usa una especificación de clase para construir objetos. Los objetos son instancias de una clase. Por lo tanto, una clase es esencialmente un conjunto de planes que especifican cómo construir un objeto.

***Otro punto:** recuerde que los métodos y las variables que constituyen una clase se llaman miembros de la clase. Los miembros de datos también se conocen como variables de instancia.*

2. The General Form of a Class

```
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
   }
}
```

3. Defining a Class

Para ilustrar las clases, desarrollaremos una clase que encapsule información sobre vehículos, como automóviles, camionetas y camiones. Esta clase se llama Vehículo y almacenará tres elementos de información sobre un vehículo: la cantidad de pasajeros que puede transportar, su capacidad de combustible y su consumo promedio de combustible (en millas por galón).

```
class Vehicle {
```

```
   int passengers; // number of passengers
   int fuelcap; // fuel capacity in gallons
   int mpg; // fuel consumption in miles per gallon
}
```

Una definición de clase crea un nuevo tipo de datos. En este caso, el nuevo tipo de datos se llama Vehículo. Utilizará este nombre para declarar objetos de tipo Vehículo. Recuerde que una declaración de clase es solo una descripción de tipo; no crea un objeto real. Por lo tanto, el código anterior no provoca la existencia de ningún objeto de tipo Vehículo.

To actually create a **Vehicle** object, you will use a statement like the following:

```
Vehicle minivan = new Vehicle(); // crea un objeto de la clase Vehiculo llamado minivan
```

Cada objeto de vehículo contendrá sus propias copias de las variables de instancia pasajeros, fuelcap y mpg. Para acceder a estas variables, utilizará el operador de punto (.): minivan.fuelcap = 16;

/* A program that uses the Vehicle class. Call this file **VehicleDemo.java** */

```
class Vehicle {

   int passengers; // number of passengers
   int fuelcap; // fuel capacity in gallons
   int mpg; // fuel consumption in miles per gallon
}

class VehicleDemo {

   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      int range;
// 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);
   }
}
```

Debe llamar al archivo que contiene este programa VehicleDemo.java porque el método main () está en la clase llamada VehicleDemo, no en la clase llamada Vehicle. Cuando compile este programa, encontrará que se han creado dos archivos .class, uno para Vehicle y otro para VehicleDemo. **El compilador de Java coloca automáticamente cada clase en su propio archivo .class.** No es necesario que las clases Vehicle y VehicleDemo estén en el mismo archivo fuente. Podría poner cada clase en su propio archivo, llamado Vehicle.java y VehicleDemo.java, respectivamente.

Each object has its own copies of the instance variables defined by its class.

```
class TwoVehicles {

   public static void main(String args[]) {
      Vehicle minivan = new Vehicle();
      Vehicle sportscar = new Vehicle();
      int range1, range2;

// assign values to fields in minivan
      minivan.passengers = 7;
      minivan.fuelcap = 16;
      minivan.mpg = 21;

// assign values to fields in sportscar
      sportscar.passengers = 2;
      sportscar.fuelcap = 14;
      sportscar.mpg = 12;

// compute the ranges assuming a full tank of gas
      range1 = minivan.fuelcap * minivan.mpg;
      range2 = sportscar.fuelcap * sportscar.mpg;
      System.out.println("Minivan can carry " + minivan.passengers
            + " with a range of " + range1);
      System.out.println("Sportscar can carry " + sportscar.passengers
            + " with a range of " + range2);
   }
}
```

4. How Objects Are Created

```
Vehicle minivan; // declare reference to object
minivan = new Vehicle(); // allocate a Vehicle object
```

Esta declaración realiza dos funciones. Primero, declara una variable llamada minivan del tipo de clase Vehículo. Esta variable no define un objeto. En su lugar, es simplemente una variable que puede referirse a un objeto. En segundo lugar, la declaración crea una copia física del objeto y asigna a minivan una referencia a ese objeto. Esto se hace utilizando el operador **new** .
El operador **new** asigna dinámicamente (es decir, asigna en tiempo de ejecución) la memoria para un objeto y le devuelve una referencia. Esta referencia es, más o menos, la dirección en memoria del objeto asignado por **new.** Esta referencia se almacena en una variable. Por lo tanto, en Java, todos los objetos de clase deben asignarse dinámicamente.

```
Vehicle minivan; // declare reference to object
minivan = new Vehicle(); // allocate a Vehicle object
```

5. Methods

Los métodos son subrutinas que manipulan los datos definidos por la clase y, en muchos casos, proporcionan acceso a esos datos. En la mayoría de los casos, otras partes de su programa interactuarán con una clase a través de sus métodos.

The general form of a method is shown here:

```
return-type nameOfMethod( parameter-list ) {
    // body of method
}
```

```
// Add range to Vehicle.

class Vehicle {
   int passengers; // number of passengers
   int fuelcap; // fuel capacity in gallons
   int mpg; // fuel consumption in miles per gallon

// Display the range.

   void range() {
      System.out.println("Range is " + fuelcap * mpg);
   }
}
```

Returning from a Method

Hay dos formas de devolución: una para uso en métodos nulos (aquellos que no devuelven un valor) y otra para valores devueltos. La primera forma se examina aquí.
En un método nulo, puede causar la terminación inmediata de un método usando return;

Returning a Value

Los valores de retorno se utilizan para una variedad de propósitos en la programación. En algunos casos, el valor de retorno contiene el resultado de algunos cálculos. En otros casos, el valor de retorno puede simplemente indicar éxito o fracaso. En otros, puede contener un código de estado.

```
// Use a return value.
class Vehicle {
   int passengers; // number of passengers
   int fuelcap; // fuel capacity in gallons
   int mpg; // fuel consumption in miles per gallon

// Return the range.
   int range() {
      return mpg * fuelcap;
   }
```

```
    }
```

Using Parameters

Es posible pasar uno o más valores a un método cuando se llama al método. Recuerde que un valor pasado a un método se llama un argumento. Dentro del método, la variable que recibe el argumento se llama parámetro. Los parámetros se declaran dentro de los paréntesis que siguen al nombre del método. La sintaxis de la declaración de parámetros es la misma que la utilizada para las variables. Un parámetro está dentro del alcance(ambito) de su método y, aparte de su tarea especial de recibir un argumento, actúa como cualquier otra variable local.

```
class Factor {
   boolean isFactor(int a, int b) {
      if ((b % a) == 0) {
         return true;
      } else {
         return false;
      }
   }
}
```

Adding a Parameterized Method to Vehicle

```
class Vehicle {

   int passengers; // number of passengers
   int fuelcap; // fuel capacity in gallons
   int mpg; // fuel consumption in miles per gallon
// Return the range.

   int range() {
      return mpg * fuelcap;
   }
// Compute fuel needed for a given distance.

   double fuelneeded(int miles) {
      return (double) miles / mpg;
   }
}
```

6. Constructors

Un constructor inicializa un objeto cuando se crea.
Tiene el mismo nombre que su clase y es sintácticamente similar a un método. **Sin embargo, los constructores no tienen un tipo de retorno explícito.**
Normalmente, utilizará un constructor para asignar valores iniciales a las variables de instancia definidas por la clase, o para realizar cualquier otro procedimiento de inicio requerido para crear un objeto completamente formado.
Todas las clases tienen constructores, ya sea que defina uno o no, porque Java proporciona automáticamente un constructor predeterminado que inicializa todas las variables miembro a sus valores predeterminados, que son cero, nulo y falso, para tipos numéricos, tipos de referencia y valores booleanos, respectivamente. Sin embargo, una vez que define su propio constructor, el constructor predeterminado ya no se usa.

Here is a simple example that uses a constructor:

```
class MyClass {
   int x;
   MyClass() {
      x = 10;
   }
}

class ConstructorDemo {
   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

Parameterized Constructors

La mayoría de las veces necesitarás un constructor que acepte uno o más parámetros. Los parámetros se agregan a un constructor de la misma manera que se agregan a un método: simplemente declararlos dentro de los paréntesis después del nombre del constructor.

```
class MyClass {
   int x;

   MyClass() {
      x = 10;
   }

   MyClass(int i) {
      x = i;
   }
}
```

Adding a Constructor to the Vehicle Class

```
class Vehicle {

  int passengers; // number of passengers
  int fuelcap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon


// Return the range.

  Vehicle(int p, int f, int m) {
     passengers = p;
```

```
        fuelcap = f;
        mpg = m;
    }

    int range() {
        return mpg * fuelcap;
    }
// Compute fuel needed for a given distance.

    double fuelneeded(int miles) {
        return (double) miles / mpg;
    }
}
```

7. Garbage Collection

Como ha visto, los objetos se asignan dinámicamente desde un grupo de memoria libre mediante el uso del operador ***new*** *. Como se explicó, la memoria no es infinita y la memoria libre puede agotarse. Por lo tanto, es posible que fallen(los new) porque no hay suficiente memoria libre para crear el objeto deseado.*
Por esta razón, un componente clave de cualquier esquema de asignación dinámica es la recuperación de la memoria libre de los objetos no utilizados, haciendo que esa memoria esté disponible para su posterior reasignación.
En algunos lenguajes de programación, el lanzamiento de la memoria previamente asignada se maneja manualmente. Sin embargo, Java utiliza un enfoque diferente, más libre de problemas: la recolección de basura.
El sistema de recolección de basura de Java recupera los objetos automáticamente, se produce de manera transparente, detrás de la escena, sin la intervención del programador.
Funciona así: cuando no existen referencias a un objeto, se supone que ese objeto ya no es necesario y se libera la memoria ocupada por el objeto. Esta memoria reciclada se puede utilizar para una asignación posterior.

8. The finalize( ) Method and The *this* Keyword

Es posible definir un método que será llamado justo antes de la destrucción final de un objeto por el recolector de basura. Este método se llama finalize (), y se puede usar para asegurar que un objeto termine de forma limpia. Por ejemplo, puede usar finalize () para asegurarse de que un archivo abierto que es propiedad de ese objeto está cerrado.

Para agregar un finalizador a una clase, simplemente defina el método finalize (). El sistema Java en tiempo de ejecución llama a ese método cuando está a punto de reciclar un objeto de esa clase. Dentro del método finalize (), especificará las acciones que deben realizarse antes de que se destruya un objeto.

The finalize( ) method has this general form:

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

Ejemplos:

```
class FDemo {

    int x;

    FDemo(int x) {
        this.x = x;
        System.out.println("Creado objeto "+x);
```

```
    }

  // called when object is recycled
    protected void finalize() {
      System.out.println("Finalizing " + x);
    }
}

class Finalize {

    public static void main(String args[]) {
      int count;

        /* Now, generate a large number of objects. At some point, garbage collection
        will occur.
       Note: you might need to increase the number of objects generated in order to
       force garbage collection. */

      for (count = 1; count < 100000; count++) {
        FDemo ob = new FDemo(count);

      }
    }
}
```

9. Controlling Access to Class Members

Hay dos tipos básicos de miembros de la clase: públicos y privados. Se puede acceder libremente a un miembro público mediante un código definido fuera de su clase. Este es el tipo de miembro de clase que hemos estado usando hasta este momento. Solo se puede acceder a un miembro privado mediante otros métodos definidos por su clase. Es a través del uso de miembros privados que el acceso está controlado.

Java's Access Modifiers

El control de acceso de los miembros se logra mediante el uso de tres modificadores de acceso: público, privado y protegido.

Cuando un miembro de una clase es cualificado por el especificador público, se puede acceder a ese miembro por cualquier otro código en su programa. Esto incluye métodos definidos dentro de otras clases.

Cuando un miembro de una clase se especifica como privado, solo se puede acceder a ese miembro por otros miembros de su clase. Por lo tanto, los métodos en otras clases no pueden acceder a un miembro privado de otra clase.

La configuración de acceso predeterminada (en la que no se usa un modificador de acceso) es la misma que la de público, a menos que su programa se divida en paquetes. Un paquete es, esencialmente, una agrupación de clases (veremos más adelante).

```
class MiClase {

   private int alpha; // private access
   public int beta; // public access
   int gamma; // default access

   void setAlpha(int a) {
      alpha = a;
   }

   int getAlpha() {
      return alpha;
   }
}

class Acceso {
   public static void main(String args[]) {
      MiClase ob = new MiClase();
      ob.setAlpha(-99);
      System.out.println("ob.alpha is " + ob.getAlpha());
      ob.beta = 88;
      ob.gamma = 99;
   }
}
```

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	TODOS
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
No especificado	Sí	Sí	No	No
private	Sí	No	No	No

10. Pass Objects to Methods and returning objects

```
class Block {
   int a, b, c;
   int volume;

   Block(int a, int b, int c) {
```

```
        this.a=a;
        this.b=b;
        this.c=c;
        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;
        }
    }

    Block initializeBlock()
    {
        return new Block(0,0,0);
    }
}

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 same dimensions as ob2: "
                + ob1.sameBlock(ob2));
        System.out.println("ob1 same dimensions as ob3: "
                + ob1.sameBlock(ob3));
        System.out.println("ob1 same volume as ob3: "
                + ob1.sameVolume(ob3));
        ob3=ob3.initializeBlock();

    }
}
```

Ejercicio 1

Crea una clase llamada *Vehículo:*

- Con los **atributos privados** *numRuedas* (número de ruedas), *velMax* (velocidad máxima) y *peso*.
- Implementa los **métodos** necesarios para acceder a estos atributos de manera **pública** (*getters* y *setters*).

- La clase dispondrá de un **constructor** que necesitará como parámetros los valores iniciales para todos sus atributos.

- Crea un **método** público *boolean esIgual(Vehiculo)* que sirva para comparar dos vehículos, de manera que devuelva true o false dependiendo de si son iguales o no (tienen todos sus atributos el mismo valor o no). El método recibirá como parámetro un objeto de la clase *Vehículo*.

- Crea un **método** público void *copia(Vehiculo)* que copiará un vehículo en otro. El método recibirá como parámetro un objeto de la clase *Vehículo* del cual se copiarán sus valores.

- Crea una clase aparte con el método *main* para probar todas las funcionalidades de la clase *Vehículo*.

Ejercicio 2

Crea una clase llamada *Nombres*, capaz de gestionar una lista de nombres de un tamaño determinado:

El **constructor** recibirá como parámetro el número máximo de nombres que albergará.

Tendrá los siguientes **métodos**:
int anadir(String): Añade a la lista el nombre pasado como parámetro. Devuelve -1 si la lista está llena, 0 si se añade con éxito y 1 si ya existía el nombre (no admite repetidos).

boolean eliminar(String): Elimina de la lista el nombre pasado como parámetro. Devuelve *true* en caso de éxito y *false* si no encuentra el nombre a eliminar.

void vaciar(): Elimina todos los nombres de la lista.

String mostrar(int): Devuelve el nombre que se encuentra en la posición pasada como parámetro (la primera es la posición 0).

int numNombres(): Devuelve el número de nombres que hay actualmente.

int maxNombres(): Devuelve el número máximo de nombres que puede albergar.

boolean estaLlena(): Devuelve *true* si la lista está llena y *false* en caso contrario.

11. Method Overloading

En Java, dos o más métodos dentro de la misma clase pueden compartir el mismo nombre, siempre y cuando sus declaraciones de parámetros sean diferentes. Cuando este es el caso, se dice que los métodos están sobrecargados, y el proceso se denomina sobrecaraa de métodos. La sobrecarga de métodos es una de las formas en que Java implementa el polimorfismo.

```
class Overload {
   void ovlDemo() {
      System.out.println("No parameters");
   }
   void ovlDemo(int a) {
      System.out.println("One parameter: " + a);
   }
   int ovlDemo(int a, int b) {
      System.out.println("Two parameters: " + a + " " + b);
      return a + b;
   }
   double ovlDemo(double a, double b) {
      System.out.println("Two double parameters: " + a + " " + b);
      return a + b;
   }
}

public class OverloadDemo {

   public static void main(String args[]) {
      Overload ob = new Overload();
      int resI;
      double resD;
      ob.ovlDemo();
      System.out.println();
      ob.ovlDemo(2);
      System.out.println();
      resI = ob.ovlDemo(4, 6);
      System.out.println("Result of ob.ovlDemo(4, 6): " + resI);
      System.out.println();
      resD = ob.ovlDemo(1.1, 2.32);
      System.out.println("Result of ob.ovlDemo(1.1, 2.32): " + resD);
   }
}
```

Overloading Constructors

```
class OverConstructor {
   int x;
  OverConstructor() {
      System.out.println("Inside MyClass().");
      x = 0;
   }
   OverConstructor(int i) {
      System.out.println("Inside MyClass(int).");
```

```
        x = i;
    }
    OverConstructor(double d) {
        System.out.println("Inside MyClass(double).");
        x = (int) d;
    }
    OverConstructor(int i, int j) {
        System.out.println("Inside MyClass(int, int).");
        x = i * j;
    }
}
```

12. Understanding static

Habrá ocasiones en las que querrá definir un miembro de la clase que se utilizará independientemente de cualquier objeto de esa clase. Normalmente, se debe acceder a un miembro de la clase a través de un objeto de su clase, pero es posible crear un miembro que se pueda usar solo, sin hacer referencia a una instancia específica. Para crear dicho miembro, preceda su declaración con la palabra clave static. Cuando un miembro se declara estático, se puede acceder a él antes de que se cree cualquier objeto de su clase y sin hacer referencia a ningún objeto. Puedes declarar que tanto los métodos como las variables son estáticos. El ejemplo más común de un miembro estático es main (). main () se declara como estático porque debe ser llamado por la JVM cuando comienza su programa. Fuera de la clase, para usar un miembro estático, solo necesita especificar el nombre de su clase seguido del operador punto. Ningún objeto necesita ser creado.

Las variables declaradas como estáticas son, esencialmente, variables globales. Cuando se declara un objeto, no se realiza ninguna copia de una variable estática. En su lugar, todas las instancias de la clase comparten la misma variable estática.

```
class StaticDemo {
    int x; // a normal instance variable
    static int y; // a static variable
    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();
    }
}
```

Ejercicio 3

Crea una clase llamada *Empleado*:

Con los **atributos privados** *nombre*, y *teléfono*.

Implementa los **métodos** necesarios para acceder a estos dos atributos de manera **pública** (*getters* y *setters*).

Añade el **atributo estático y privado** *numEmpleados*. Este atributo almacenará el número de instancias que se han creado de la clase *Empleado*.

Implementa un **método público** para conocer el valor de *numEmpleados*.

Crea una clase aparte con el método *main* para probar todas las funcionalidades de la clase *Empleado*.

Ejercicio 4

Crea una clase llamada *MiString*, sin atributos y con los siguientes **métodos públicos y estáticos**:

String alReves(String): Devuelve la cadena pasada como parámetro pero al revés.

String limpiaCaracteres(String, String): Devuelve la cadena pasada como primer parámetro pero eliminando los caracteres pasados en la cadena como segundo parámetro.

String quitaTildes(String): Devuelve la cadena pasada como parámetro pero sin tildes.

boolean esPalindromo(String): Devuelve *true* o *false*, dependiendo de si es o no palíndromo la frase pasada como parámetro.

boolean esNumero(String): Devuelve *true* o *false*, dependiendo de si la cadena pasada como parámetro es un número o no.

Crea una clase aparte con el método *main* para probar todas las funcionalidades de la clase *MiString*.

Ejercicio 5

Crea una clase llamada *MiString*, sin atributos y con los siguientes **métodos públicos y estáticos**:

String alReves(String): Devuelve la cadena pasada como parámetro pero al revés.
String limpiaCaracteres(String, String): Devuelve la cadena pasada como primer parámetro pero eliminando los caracteres pasados en la cadena como segundo parámetro.

String quitaTildes(String): Devuelve la cadena pasada como parámetro pero sin tildes.

boolean esPalindromo(String): Devuelve *true* o *false*, dependiendo de si es o no palíndromo la frase pasada como parámetro.

boolean esNumero(String): Devuelve *true* o *false*, dependiendo de si la cadena pasada como parámetro es un número o no.

Crea una clase aparte con el método *main* para probar todas las funcionalidades de la clase *MiString*.

13. Recursion o recursividad

En Java, un método puede llamarse a sí mismo. Este proceso se llama recursión, y se dice que un método que se llama a sí mismo es recursivo.

```
class Factorial {
// This is a recursive function.
   int factR(int n) {
      int result;
      if (n == 1) {
         return 1;
      }
      result = factR(n - 1) * n;
      return result;
   }
// This is an iterative equivalent.
   int factI(int n) {
      int t, result;
      result = 1;
      for (t = 1; t <= n; t++) {
         result *= t;
      }
      return result;
   }
}

class Recursion {
```

```
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Factorials using recursive method.");
        System.out.println("Factorial of 3 is " + f.factR(3));
        System.out.println("Factorial of 4 is " + f.factR(4));
        System.out.println("Factorial of 5 is " + f.factR(5));
        System.out.println();
        System.out.println("Factorials using iterative method.");
        System.out.println("Factorial of 3 is " + f.factI(3));
        System.out.println("Factorial of 4 is " + f.factI(4));
        System.out.println("Factorial of 5 is " + f.factI(5));
    }
}
```

```
private Image imagen=ImageIO.read(new File(StringDeLaRutaUbicacionArchivo));
```