Prácticas de Programación

Tema 2. Clases y objetos en Java

2.1 Introducción

La clase es la unidad básica de estructuración en un lenguaje orientado a objetos

Está formada por miembros:

- **atributos (también llamados campos): almacenan el estado**
- **métodos: definen el comportamiento**

Una *clase* es una *descripción* de un conjunto de objetos con los mismos atributos y comportamiento

```
public class UnaClase {
  atributos
  constructores
  métodos
}
```

Un *objeto* es una instancia o *ejemplar concreto* de una clase

2.2 Creación e inicialización de objetos

En el lenguaje Java los objetos se crean con el operador `new`:

```
UnaClase obj = new UnaClase();
```

Se invoca el CONSTRUCTOR

Los constructores son métodos especiales que

- **se llaman igual que la clase**
- **no tienen tipo de retorno**

Una clase puede tener varios constructores con diferentes parámetros

```
public UnaClase() { ... }
public UnaClase(int i) { ... }
public UnaClase(boolean b, int i) { ... }
```

Si *y sólo si* no definimos constructor, la clase tendrá el constructor por defecto (constructor sin parámetros que no hace nada)

```
public UnaClase() {}
```


Inicialización de los atributos

Cuando se crea un objeto sus atributos se inicializan en 3 etapas:

1. **Se asigna a cada atributo su valor por defecto**
   - **tipos numéricos (`int`, `double`, ...): 0 (o 0.0)**
   - **Caracteres (`char`): carácter nulo**
   - **Booleanos (`boolean`): `false`**
   - **Referencias a objetos (`String`, otras clases): `null`**
2. **Si la declaración del atributo contiene un inicializador, éste es evaluado y asignado**

   ```
   private int numRuedas = numCoches * 4;
   ```
3. **Se ejecuta el constructor de la clase, que puede cambiar el valor de los atributos que desee**

Recordar: sólo se inicializan los atributos, las variables locales de los métodos no son inicializadas por defecto

Atributos “finales”

Su valor no se puede cambiar después de haber sido inicializados

- **pueden inicializarse en la definición del atributo o en el constructor de la clase**

Ejemplo:

```
public class Círculo {

  private final double PI = 3.14159;
  public final double área;
  private double radio;

  public Círculo(double radio) {
    this.radio = radio;
    this.área = PI * radio * radio;
  }
  ... // no se puede modificar el valor de área
```

Inicialización de “PI”

hay un valor más preciso de PI en la clase Math

La inicialización de “área” se deja para el constructor

2.3 Tipos primitivos, referencias y objetos

- **Cuando se declarara una variable de un *tipo primitivo* el compilador reserva un área de memoria para ella**

```
int i, j;
```

- **Cuando se asigna un valor, éste es escrito en el área reservada**

```
i=16;
j=3;
```

- **La asignación entre variables significa copiar el contenido de una variable en la otra**

```
i=j;
```

Referencias y objetos

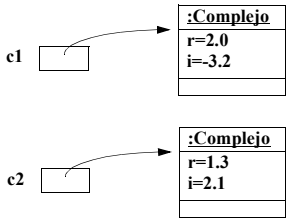
Cuando se declara una variable "objeto" la situación es diferente

- **Si no se usa el operador `new` sólo se crea una referencia a objeto**

```
Complejo c1,c2;
```

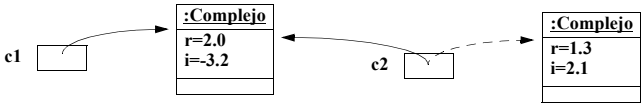
- **El espacio en memoria del objeto se reserva con el operador `new`**

```
c1=new Complejo(2.0,-3.2);
c2=new Complejo(1.3,2.1);
```



- **La asignación entre objetos sólo copia referencias**
  - **después de la copia ambas referencias apuntan al mismo objeto**

```
c2=c1;
```



- **La comparación entre objetos compara las referencias no los contenidos**
  - **en la figura anterior `c1==c2` es `true`**
  - **en la situación de debajo `c1==c2` es `false`**

  - **el método `equals` permite comparar contenidos (Ver 2.5)**

Arrays de tipos primitivos

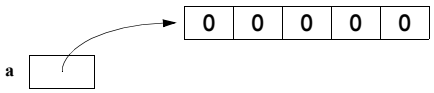
Java trata los arrays como objetos

- **Crear una variable de tipo array sólo crea una referencia**

```
int a[];
```

- **El espacio en memoria para el array se crea con `new`**
  - **cuando se trata de un array de un tipo primitivo se crean los elementos del array**
  - **y se inicializan a sus valores por defecto**

```
a = new int[5];
```



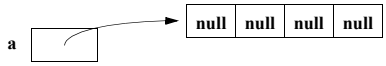
Arrays de referencias a objetos

- **Crear una variable de tipo array de objetos sólo crea la referencia**

```
Complejo a[];
```

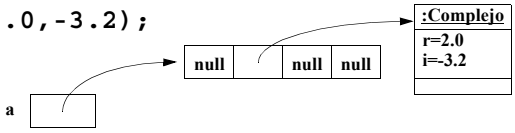
- **Con `new` se reserva el espacio para el array de referencias**
  - **pero no los objetos a los que apuntarán las referencias**
  - **las referencias que forman el array se inicializan a `null`**

```
a=new Complejo[4];
```



- **Los objetos hay que crearlos posteriormente**

```
a[1]=new Complejo(2.0,-3.2);
```

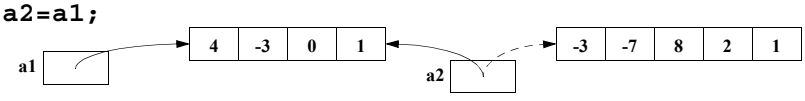


Copia y comparación de arrays (y strings)

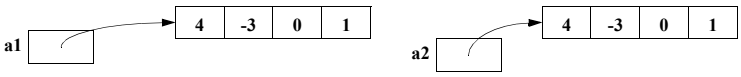
Los arrays y los strings son objetos, por consiguiente:

- **La asignación entre arrays (o strings) sólo copia referencias**

```
a2=a1;
```



- **La comparación entre arrays (o strings) compara las referencias no los contenidos**
  - **en la figura anterior `a1==a2` es `true`**
  - **en la situación de debajo `a1==a2` es `false`**



- **Para comparar contenidos debe utilizarse el método `equals`**

2.4 Recolector de basura

La memoria ocupada por un objeto se reserva con el operador `new`

- **en Java no existe ningún operador para liberar la memoria de un objeto que no se va a usar más**

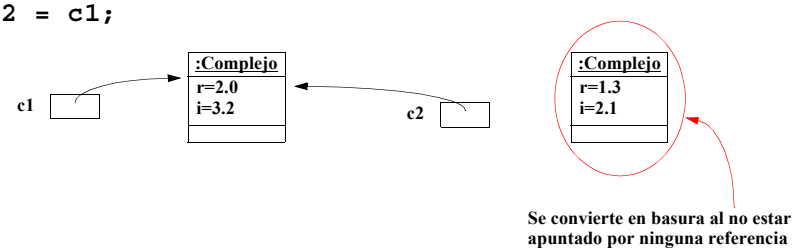
La liberación de memoria la realiza el Recolector de Basura de forma automática

- **el recolector va contando el número de referencias que "apuntan" a cada objeto**
- **cuando un objeto no está apuntado por ninguna referencia, se convierte en "basura" (no va a poder ser utilizado nunca más)**
- **en los periodos de inactividad del sistema, el recolector reincorpora la "basura" a la memoria disponible**

Ejemplo de generación de basura

```
Complejo c1 = new Complejo(2.0, 3.2);
Complejo c2 = new Complejo(1.3, 2.1);
```


```
c2 = c1;
```



2.5 Comparación de objetos

La comparación entre objetos (`obj1==obj2`) compara referencias

- **si queremos *comparar los contenidos* de los objetos debemos utilizar el método `equals`**

Ejemplo: método `equals` para la clase `Coordenada`:

```
public class Coordenada {
  private int x; // coordenada en el eje x
  private int y; // coordenada en el eje y

  ...

  public boolean equals(Object obj) {
    Coordenada c = (Coordenada) obj;
    return c.x == x && c.y == y;
  }
}
```

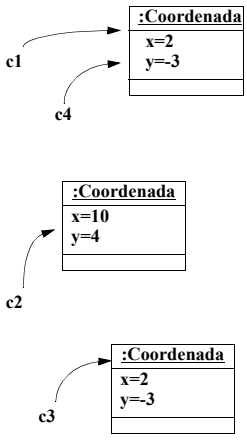
Ejemplo de uso del método `equals`

```
if (c1.equals(c1)) {
  ...
}
```
true

```
if (c1.equals(c4)) {
  ...
}
```
false

```
if (c1.equals(c2)) {
  ...
}
```
true

```
if (c1.equals(c3)) {
  ...
}
```



Comparación de contenidos de arrays y strings

La clase `String` dispone del método `equals`:

```
String str1 = "hola", str2 = "hola";
if (str1.equals(str2)) {
  System.out.println("Los strings son iguales");
}
```

Para comparar arrays existe el método `equals` de la clase `Arrays`:

```
import java.util.Arrays;
int[] a1= {4, -3, 0, 1}, a2= {4, -3, 0, 1};
if (Arrays.equals(a1, a2)) {
  System.out.println("'a1' es igual a 'a2'");
}
```

Volveremos a hablar del método equals en el tema 3, "Herencia y Polimorfismo"

2.6 Control de acceso

Los sistemas software pueden llegar a ser muy complejos

Para abordar con éxito su desarrollo es necesario aplicar los principios de:

- ***modularidad*: descomposición del sistema en un conjunto de "piezas" poco acopladas**
- ***ocultamiento de información*: se ocultan los detalles de implementación de las "piezas"**

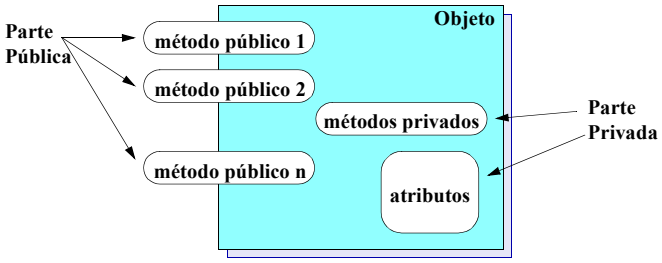
Ventajas de utilizar modularidad y ocultamiento de información

- **el ocultamiento de los detalles facilita la comprensión global del sistema**
- **el cambio en la implementación de un módulo no afecta al resto de la aplicación**
- **el acoplo débil entre módulos facilita la reutilización de código**

Modularidad y abstracción en POO

En POO ambos principios se basan en el *concepto de objeto*

- **cada objeto es una "pieza" del sistema final**
- **los objetos se relacionan mediante sus miembros públicos**
- **los miembros privados están "ocultos" para los demás objetos**



Modificadores de acceso

Se pueden poner a una clase o miembro (campo o método)

- **indican desde dónde es accesible**

Modificadores de acceso para clases:

- **<*ninguno*>: accesible desde el paquete**
- **`public`: accesible desde todo el programa**

Modificadores de acceso para miembros de clases:

- **<*ninguno*>: accesible desde el paquete**
- **`public`: accesible desde todo el programa**
- **`private`: accesible sólo desde esa clase**
- **`protected`: accesible desde el paquete y desde sus subclases en cualquier paquete**

Uso habitual

Lo normal es que una clase:

- **tenga todos sus *atributos privados***
- **sólo proporcione los *métodos públicos estrictamente necesarios***

De esta forma el estado del objeto sólo puede cambiar a través de unos puntos de entrada bien definidos

Ejemplo de uso de los modificadores de acceso

```
public class Coche {
  private String matrícula;
  private String nombrePropietario;
  private String dniPropietario;

  // constructor
  public Coche(String m, String n, String dni) {...}

  // métodos para leer los atributos
  public String matrícula() {...}
  public String nombrePropietario() {...}
  public String dniPropietario() {...}

  // métodos para cambiar los atributos
  public void cambiaPropietario(String n,
                                String dni) {...}

  ...
}
```

NO existe un método para cambiar la matrícula, ya que ese dato no cambia en toda la existencia del coche

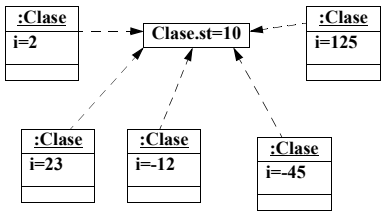
NO existe un método para cambiar el nombre y el DNI por separado, ya que cuando cambia uno, también hay que cambiar el otro

2.7 Métodos y campos de clase (o estáticos)

Una clase puede tener atributos y métodos estáticos:

- **no pertenecen a los objetos de la clase, sino a la clase misma**
- **si son atributos, sólo existe uno por clase, no uno por objeto**
  - **la vida es la de la clase, y no la del objeto**
- **si son métodos, sólo pueden acceder a los objetos pasados como parámetros y a campos y métodos estáticos**
- **nos referimos a ellos como `NombreClase.miembroEstático`**

```
public class Clase {
  private static int st;
  private int i;
  ...
}
```



Ejemplo de atributo y método de clase

```
public class Coche {

  // atributo estático
  private static int últimoNumSerie=0;

  // atributos normales
  private int numSerie;
  private String matrícula;

  // constructor
  public Coche(String matrícula){
     ultimoNumSerie++;
     numSerie=últimoNumSerie;
     this.matrícula=matrícula;
  }
```

Coche
-últimoNumSerie: int -numSerie: int -matrícula: string
+Coche(matrícula: string) +matrícula(): string +numSerie(): int +últimoNumSerie(): int

```
   // retorna la matrícula
   public String matrícula() {
      return matrícula;
   }

   // retorna el número de serie
   public int numSerie() {
      return numSerie;
   }

   // retorna el último número de serie usado
   public static int últimoNumSerie() {
      return últimoNumSerie;
   }
}
```



```
public class PruebaCoche {
   public static void main(String[] args) {
      // consulta el último número de serie usado
      System.out.println("Último número usado:" +
          Coche.últimoNumSerie());
      // crea dos coches
      Coche c1=new Coche("1111 XXX");
      Coche c2=new Coche("2222 YYY");
      // muestra los datos de un coche
      System.out.println("Coche: " + c1.matrícula());
      System.out.println("Num serie: "+c1.numSerie());
      // muestra los datos del otro coche
      System.out.println("Coche: " + c2.matrícula());
      System.out.println("Num serie: "+c2.numSerie());
   }
}
```

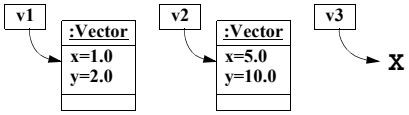

Ejemplo de método de clase

```
public class Vector2D {


   // atributos
   private double x;
   private double y;


   /** Constructor */
   public Vector2D(double x, double y) {
      this.x=x;
      this.y=y;
   }
```

```
  /**
   *  Método NO estático: Suma el vector v con
   *  el vector actual
   */
  public void suma(Vector2D v){
    x = x + v.x;
    y = y + v.y;
  }

  /**
   *  Método ESTÁTICO: Retorna el vector
   *  resultado de v1+v2
   */
  public static Vector2D suma(Vector2D v1,
                              Vector2D v2) {
    return new Vector2D(v1.x+v2.x, v1.y+v2.y);
  }
}
```

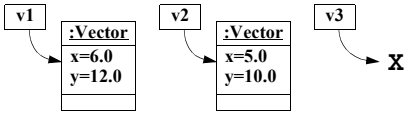


```
Vector2D v1 = new Vector2D(1,2);
Vector2D v2 = new Vector2D(5,10);
Vector2D v3;
```



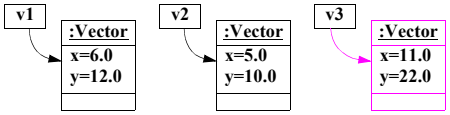

```
// suma no estática
v1.suma(v2);
```

refObjeto.nombreMétodo




```
// suma estática
v3=Vector2D.suma(v1,v2);
```

NombreClase.nombreMétodo



2.8 Anidamiento de clases

Una clase puede definirse dentro de otra

- **las clases anidadas no estáticas se denominan clases internas y están asociadas con una instancia (objeto) de la clase**
- **las clases anidadas estáticas se comportan como cualquier otra clase**
  - **salvo que su nombre es `ClaseContenedora.ClaseAnidada`**
  - **las usaremos cuando veamos las excepciones y los enumerados**

Ejemplo de clase anidada estática

```
public class Línea {

  public static class Punto {
    double x, y;
  }

  // atributos
  private Punto pIni, pFin;

  /** constructor */
  public Línea (Punto pIni, Punto pFin) {
    this.pIni=pIni;
    this.pFin=pFin;
  }

  ...
}
```

clase anidada estática


```
public class PruebaLínea {

  public static void main() {

    Línea.Punto pIni = new Línea.Punto();
    Línea.Punto pFin = new Línea.Punto();

    pIni.x=10.2; pIni.y=4.3;
    pFin.x=1.3;  pFin.y=5.6;

    Línea l = new Línea(pIni, pFin);
  }

}
```


2.9 Bibliografía

- **Ken Arnold, James Gosling, David Holmes, "El lenguaje de programación Java", 3ª edición. Addison-Wesley, 2000.**
- **Eitel, Harvey M. y Deitel, Paul J., "Cómo programar en Java", quinta edición. Pearson Educación, México, 2004.**
- **King, Kim N. "Java programming : from the beginning". W. W. Norton & Company, cop. 2000**
- **Francisco Gutiérrez, Francisco Durán, Ernesto Pimentel. "Programación Orientada a Objetos con Java". Paraninfo, 2007.**