Programación Orientada a Objetos en Java

Fundamentos de la Programación II

Indice

- Previo:
  - Tipos de datos
  - Estructuras de control y
  - Operadores.
- Clases
- Objetos
- Polimorfismo
- Herencia
- Paquetes
- Interfaces

Tipos de datos simples

Tipo	Formato	Descripción/Rango
int	32 bits complemento a 2	[-2.147.483.648, 2.147.483.647]
byte	8 bits complemento a 2	[-128, 127]
short	16 bits	[-32768, 32767] (Big-endian)
long	64 bits	[-9,223,372,036,854,775,808 y 9,223,372,036,854,775,807]
double	64 bits	[-1.797 a -4.9E-324] negativos [4.9E-324 a 1.797] positivos
float	32 bits	[-3.402E38 a -1.4E-45] negativos [1.401E-45 a 3.4E38] positivos
char	16 bits	
boolean	true/false	

Los tipos de datos simples no son objetos

Los operadores y su prioridad

Precedencia más alta	
Operadores postfijo	() [] . expr++ expr--
Operadores unarios	++expr --expr +expr -expr ~ !
Creación o cast	new (typo)expr
multiplicativas	* / %
aditivas	+ -
shift	<< >> >>>
relacional	< > <= >= instanceof
igualdad	== !=
Bit AND	&
Bit OR exclus.	^
Bit OR inclus.	|
AND lógico	&&
OR lógico	||
condicional	?:
asignación	= += -= *= /= %= &= ^= |= <<= >>= >>>=
Precedencia más baja	

Estructuras de control

selección	iteración	salto

`if(condición){`
`  sentencia1;`
`}else{`
`  sentencia2;`
`}`
	`while(condición) {`
	`  //cuerpo del bucle`
	`}`
		`break`

`switch (expresión){`
`  case valor1:`
`  //sentencias`
`  break;`
`  case valor2:`
`  //sentencias`
`  break;`
`  case valorN:`
`  //sentencias`
`  break;`
`  default:`
`  // sentencias`
`}`
	`for(inicialización;`
	`  condición;`
	`  iteración) {`
	`  //cuerpo del bucle`
	`}`
		`return`

	`do{`
	`  //cuerpo del bucle`
	`}while(condición);`
		`continue`
		(implica código no estructurado)

Forma general de la definición de una clase en Java

```
class NombreDeLaClase{
    tipo variableDeInstancia1;
    //...
    tipoN variableDeInstanciaN;
    tipo nombreDeMetodo1(parámetros){
       /* cuerpo del método*/
    }
    tipo nombreDeMetodo1(parámetros){
       /* cuerpo del método*/
    }
}
```

Objetos

- Los objetos se obtienen a partir de una clase. Dos pasos:
  1. Declarar una variable del tipo de la clase.
     - La declaración no define un objeto.
     - Mediante la variable podemos referirnos a un objeto.
  2. Obtener una copia física y real del objeto y asignarla a esa variable.
     - Operador new.

new

- El operador **new** permite una asignación dinámica en tiempo de ejecución, reserva memoria para un objeto y
  - devuelve una referencia al mismo.
- Esta referencia se puede almacenar en la variable.
- Todos los objetos en Java deben ser asignados dinámicamente.
- Si no es capaz de reservar memoria se produce una excepción.
- Sintaxis para new:

```
Variable = new NombreDeLaClase();
```

- **`Variable`** es una variable de la clase que se quiere crear.
- **`NombreDeLaClase`** es el nombre de la clase que está siendo instanciada.
- El nombre de la clase seguido por los paréntesis especifica el **constructor de la clase**.

Constructor

- Un constructor define qué ocurre cuando se crea un objeto de una clase.
- Si no se ha especificado explícitamente el constructor, entonces Java automáticamente utiliza un constructor por defecto.
- Se pueden definir explícitamente los constructores dentro de la definición de la clase.
- Los tipos de datos simples no necesitan constructor ya que no son objetos.

```
public class Mesa{
    private double altura;
    private double anchura;
    private double profundidad;
}

class Principal{
   public static void main (String args[]){
       Mesa miMesa = new Mesa();
    }
}
```

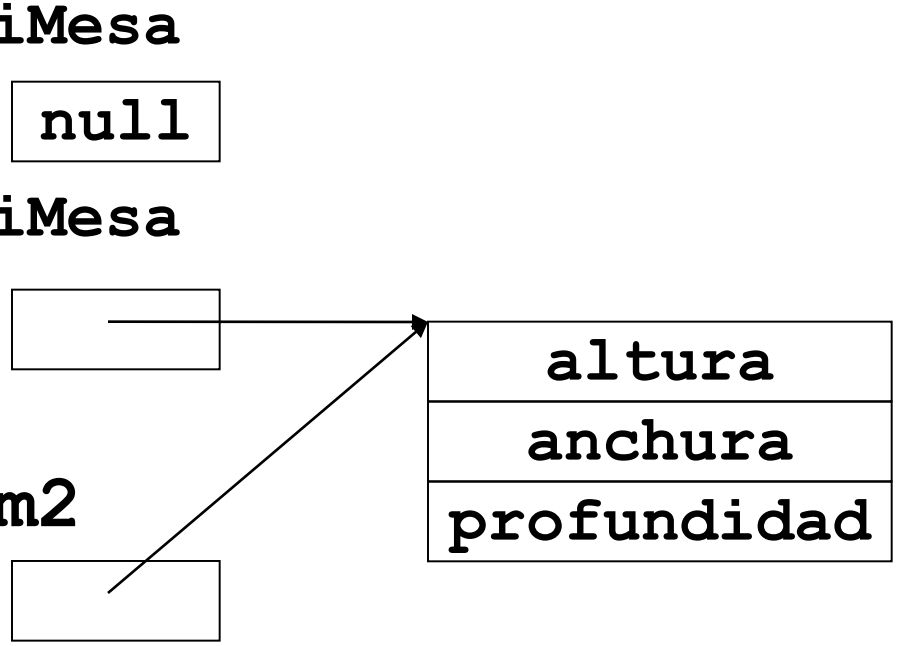
Ejemplo de declaración e instanciación de un objeto

Setencia

```
Mesa miMesa;

miMesa = new Mesa();

Mesa m2 = miMesa ;
```



Ciclo de vida de un objeto

Creación de un objeto

La sentencia

```
Mesa m1 = new Mesa();
```

tiene tres acciones:

Declaración: el compilador puede usar la variable `m1` para referirse a un objeto `Mesa .`

La declaración no crea un objeto.

La declaración se realiza con tipo nombre;

Instanciación: El operador **`new`** crea un nuevo objeto y lo pone en memoria.

Inicialización: `Mesa()` es una llamada al constructor, que inicializa al objeto.

Uso del objeto. Para hacer referencia a variables o métodos del objeto mediante el punto (.).

Eliminación. Los objetos no usados.

Métodos

- Un **parámetro** es una variable definida por un método y que recibe un valor cuando se llama a ese método.
- Un **argumento** es un valor que se pasa a un método cuando éste es invocado.
- Sintaxis general de un método:

```
tipo nombreDeMétodo(lista_de_parametros){
    //cuerpo del método
}
```

tipo especifica el tipo devuelto por el método.
Si no devuelve ningún tipo, **void.**
El valor devuelto: **return expresion;**

Constructores

- Un constructor inicializa un objeto inmediatamente después de su creación.

- Tiene exactamente el mismo nombre que la clase en la que reside y sintácticamente es similar a un método.
- Se llama automáticamente al constructor después de crear el objeto.
- No devuelven ningún tipo.

```
public class Mesa{
 private double altura;
 private double anchura;
 private double profundidad;

 public Mesa(double h, double a, double p){
   altura = h;
   anchura = a;
   profundidad = p;
         }
   public void area(){
     return (anchura* profundidad);
   }
}
class Principal{
  public static void main (String args[]){
    Mesa miMesa = new Mesa(100.0,120.,90.);
    System.out.println("Area "+
                      miMesa.area());
  }
}
```

this

- Permite hacer referencia al objeto que lo invocó.
- Puede ser utilizada dentro de cualquier método para referirse al objeto actual.
- **`this`** siempre es una referencia al objeto sobre el que ha sido llamado el método.

```
public class Mesa{
 private double altura;
 private double anchura;
 private double profundidad;

 public Mesa(double h, double a, double p){
   this.altura = h;
   this.anchura = a;
   this.profundidad = p;
             }
    public void area(){
      return (anchura* profundidad);
    }
}
class Principal{
  public static void main (String args[]){
    Mesa miMesa = new Mesa(100.0,120.,90.);
    System.out.println("Area "+
                       miMesa.area());
  }
}
```

Recogida de basura

- Cuando no hay ninguna referencia a un objeto determinado, se asume que ese objeto no se va a utilizar más, y la memoria ocupada por el objeto se libera.
- No hay necesidad explícita de destruir los objetos, como sucede en C/C++.
- La recogida de basura ocurre de forma esporádica durante la ejecución de un programa y no se produce simplemente porque uno o más objetos hayan dejado de utilizarse.
- Las diferentes implementaciones de los intérpretes de Java siguen un procedimiento diferente cuando realizan la recogida de basura.

Polimorfismo

Introducción

- Java utiliza la sobrecarga de método para implementar el Polimorfismo.
- Es la posibilidad de tener dos o más **métodos** con el mismo nombre pero funcionalidad diferente.
    - Es decir, dos o más **métodos** con el mismo nombre realizan acciones diferentes.
- El compilador usará una u otra dependiendo de los parámetros usados.
- Se generará un error si los métodos solo varían en el tipo de retorno.
- Java no implementa la sobrecarga de **operadores**.

Ejemplo de sobrecarga de método

```
public class Articulo {
 private float precio;
 public void setPrecio(){
     precio = 3.50f;
 }
 public void setPrecio(  float nuevoPrecio){
     precio = nuevoPrecio;
 }
 float getPrecio(){
     return precio;
 }
}
class Principal{
 public static void main( String args[]){
    Articulo obj =  new Articulo ();
    obj.setPrecio();
    obj.setPrecio(4.f);
    System.out.println( "precio = " +  obj.getPrecio());
  }
}
```

Sobrecarga de constructores

```
public class Caja {
       private double altura;
       private double anchura;
       private double profundidad;

       public Caja(double h, double a, double p) {
         //constructor
          this.altura = h;
          this.anchura = a;
          this.profundidad = p;
       }
       public Caja() {          //constructor
         this.altura = this.anchura =  this.profundidad = -1;
       }
       public Caja(double dimension) {          //constructor
         this.altura = this.anchura =  this.profundidad = dimension;
       }
       public double volumen () {
          return this.altura * this.anchura * this.profundidad;
       }
}
```

static

- Definición de un miembro de una clase que será utilizado independientemente de cualquier objeto de esa clase.
- `main ()` es `static`.
- Las variables de instancia declaradas como `static` se utilizan normalmente para acceder a ellas desde cualquier parte del programa.

Argumentos en la línea de comandos

- Los argumentos de la línea de órdenes es la información que sigue al nombre del programa en la línea de órdenes al ejecutar el programa.
- El acceso a los argumentos de la línea de órdenes se realiza mediante cadenas almacenadas en la matriz de `main(String` que se pasa a `main()`.

```
public class LineaComandos {
      public static void main(String args[ ] ) {
      for (int i = 0; i<args.length; i++)
         System.out.println("args [ "+i +"] : "+ args[i] );
      }
}
```

Herencia

Introducción

- Permite realizar clasificaciones jerárquicas.
- Reutilización de código.
- Componentes:
    - Clase más general: superclase (padre).
    - Clase más específica: subclase (hija).
- La subclase hereda las variables de instancia y métodos.
- Una subclase puede referirse a su superclase inmediata, mediante la palabra clave **`super`**.
- Se puede utilizar de dos formas:
    1. para llamar al constructor de la superclase.
    2. para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

Ejemplo

```
class SuperClaseA{
 int i;
 void muestraI(){
  System.out.println("Valor de i " +i);
    }
}
class SubClaseB extends SuperClaseA{
  int j;
  SubClaseB (int par){
     this.j = par;
     i = par + 20;
  }
  void muestraIJ(){
    System.out.println("Valor de i "
        + i + "\nValor de j "+ j);
    }
}
```

```
class prueba{
  public static void main(String args[]){
   SubClaseB obj = new SubClaseB(5);
   obj.muestraIJ();
   System.out.println();
   obj.muestraI();
 }
}
```

Variable de la superclase referenciando a un objeto de la subclase

```
class prueba{
  public static void main( String args[]){
   SubClaseB objSub = new SubClaseB(5);
   SuperClaseA objSuper = null;
   objSuper = objSub ;
  }
}
```

Acceso a miembros y herencia: `private`

No se accede a los miembros **`private`** de la de la superclase

```
class SuperClaseA{
    int i;
    private int k; // atributo privado
    void muestraI(){
        System.out.println("Valor de i "+i);
    }
}
class SubClaseB extends SuperClaseA{
    int j;
    SubClaseB (int par){
        this.j = par;
        i = par + 20;
        k = i; //error de compilación...
    }
    void muestraIJ(){
        System.out.println("Valor de i " + i + "\nValor de j "+ j);
    }
}
```

`super`: Llamada a un constructor de la superclase

Se usa en los constructores
Debe ser la primera línea
Sintaxis: **`super(ListaDeParametros);`**
`ListaDeParametros` `especifica los parámetros del constructor de la superclase.`

`super`: Acceso a un miembro de la superclase

- **`super`** es similar a **`this`**, excepto que **`super`** siempre se refiere a la superclase de la subclase en la que se utiliza.
- Su formato es

  `super.`*`miembro`*

  donde *miembro* puede ser un método o una variable de instancia.
- **`super`** se utiliza cuando los nombres de miembros de una subclase ocultan los miembros que tienen el mismo nombre en la superclase.

Ejemplo

```
//uso de super para evitar ocultar nombres.
class A {
 int i;
}

//Crea una subclase extendiendo la clase A.
class B extends A{
 int i; //esta i oculta a la i de A

 B(int a, int b) {
  super.i = a; //i de A
  i = b ;        //i de B
 }
 void show() {
  System.out.println("i de la superclase: " +
super.i);
  System.out.println("i de la subclase: " +
i);
        }
}
```

```
class Ejem2 {
 public static void
   main(String args[]){
  B subOb = new B(1,2);
  subOb.show();
 }
}
/*
i de la superclase: 1
i de la subclase: 2
*/
```

Orden de ejecución de los constructores

- **Los constructores se ejecutan en orden de derivación** desde la superclase a la subclase
- **`super()`** tiene que ser la primera sentencia que se ejecute dentro de constructor de la subclase,
- este orden es el mismo tanto si se utiliza **`super()`** como si no se utiliza.
- Si no se utiliza **`super()`**, entonces se ejecuta el constructor por defecto o sin parámetros de cada superclase.

Ejemplo

```
class A {
      A(){System.out.println("En el constructor de A.");}
}
class B extends A{
      B(){  System.out.println("En el constructor de B.");}
}
class C extends B {
      C(){  System.out.println("En el constructor de C.");}
}
class Ejem3bis {
      public static void main(String args[]){
       C c = new C();
      }
}
/*
En el constructor de A.
En el constructor de B.
En el constructor de C.
*/
```

Sobreescritura de un método

- Se dice que un método de la subclase **sobreescribe** al método de la superclase en una jerarquía de clases, cuando un método de una subclase tiene el mismo nombre y tipo que un método de su superclase.
- La invocación de un método sobreescrito de una subclase, se refiere a la versión del método definida por la subclase.
- La versión del método definida por la superclase queda oculta.

```
class A {
          int i, j;
          A(int a, int b){
                    i = a;
                    j = b;
          }

          //imprime i y j
          void imprime(){
           System.out.println("i y j: " + i + "" + j);
          }
}
class B extends A{
          int k;
          B(int a, int b, int c){
                    super(a, b);
                    k=c;
          }

          //imprime k sobrescribe el método de A
          void imprime()
          {
                    System.out.println("k: " + k);
          }
}
class Ejem4 {
          public static void main(String args[]){
           B subOb = new B(1, 2, 3);
           subOb.imprime();
           //llama al método imprime() de B
          }
}
 /*
 k: 3
 */
```

métodos sobrecargados

```
class A {
 int i, j;
 A(int a, int b){
  i = a;
  j = b;
 }

 //imprime i y j
 void imprime(){
  System.out.println("i y j: " + i + " " + j);
 }
}
class B extends A{
 int k;
 B(int a, int b, int c){
  super(a, b);
  k=c;
 }

 //sobrecarga el método imprime()
 void imprime(String msg) {
  System.out.println(msg + k);
 }
}
```

```
class Ejem5 {
 public static void main(String args[]){
  B subOb = new B(1, 2, 3);

   //llama al método imprime() de B
   subOb.imprime("Esto es k: ");
   //llama al métodoimprime() de A
   subOb.imprime();
 }
}

/*
Esto es k: 3
i y j: 1 2
*/
```

Selección de método dinámica

- La selección de método dinámica es el mecanismo mediante el cual una llamada a una función sobrescrita se **resuelve en tiempo de ejecución**, en lugar de durante la compilación.
- La selección de método dinámica es importante ya que es la forma que tiene Java de implementar el **polimorfismo durante la ejecución**.
- Utiliza dos cosas:
    - **Una variable de referencia de la superclase puede referirse a un objeto de la subclase**
    - **Sobreescritura de método**
- Es el **tipo del objeto que está siendo referenciado**, y no el tipo de la variable referencia, es el que determina qué versión de un método sobrescrito será ejecutada.

Ejemplo

```
class A {
 void imprime(){
  System.out.println(
        "Se ejecuta el método imprime en A");
 }
}
class B extends A{
 void imprime(){ //sobrescribe imprime
  System.out.println(
        "LLama al método imprime en B");
 }
}
class C extends A{
 void imprime(){//sobrescribe imprime
  System.out.println(
        "LLama al método imprime en C");
 }
}
```

```
class Ejem7 {
         public static void main(String args[]){
          A a = new A();//objeto del tipo A
          B b = new B();//objeto del tipo B
          C c = new C();//objeto del tipo C
          A r; //obtiene una referencia de tipo A

          r = a;//r hace referencia a un objeto A
          r.imprime();//llama al metodo de A

          r = b; //r hace referencia a un objeto B
          r.imprime();//llama al metodo de B

          r = c;//r hace referencia a un objeto C
          r.imprime();//llama al metodo de C
         }
}
/*
Se ejecuta el método imprime en A
LLama al método imprime en B
LLama al método imprime en C
*/
```

final

- Su contenido no puede ser modificado.
- Debe inicializarse cuando se declara.

```
class Circulo {
    . . .
    public final static float PI = 3.141592;
    . . .
}
```

Elegir identificadores en mayúsculas para las variables **`final`**.

- Para clases: no se permite que sea superclase.

```
final class Ejecutivo {
//. . .
}
```

- Para métodos: no se permite sobreescritura.

```
class Empleado {
  . . .
    public final void aumentarSueldo(int porcentaje) {
      . . .
    }
  . . .
}
```

Object

- Es la clase raíz de todo el árbol de la jerarquía de clases Java.
- Proporciona métodos de utilidad general que pueden utilizar todos los objetos.
- Por ejemplo:

```
public    boolean equals( Object obj );
```

Establece el criterio de igualdad para los objetos de esta clase.

```
public    String  toString();
```

Obtiene la representación en forma de cadena (String) de un objeto.

Clases Abstractas

- Define una superclase que declara la estructura de una abstracción sin proporcionar implementación completa de todos los métodos.
- Deja a cada subclase la tarea de completar los detalles.
- La superclase determina la naturaleza de los métodos que las subclases deben implementar.
- No se pueden instanciar clases abstractas.
- Sintaxis de método:

```
abstract tipo nombre(ListaDeParametros);
```

- Sintaxis de clase :

```
abstrac class nombre{
  //
}
```

Ejemplo

```
//Un ejemplo sencillo de abstract
abstract class A {
	abstract void Llamada();

	void OtraLlamada(){
		System.out.println("Este es un método concreto.");
	}
}

class B extends A{
	void Llamada(){
		System.out.println("Llamada en B.");
	}
}

class Abstract0 {
	public static void main(String args[]){
		B b = new B();
		b.Llamada();
		b.OtraLlamada();
	}
}
```

Paquetes

Introducción

- Los paquetes proporcionan un mecanismo para organizar de forma estructurada las clases.
- Las clases y los paquetes son dos medios de **encapsular** y contener el nombre y el ámbito de las variables y métodos.
- Los paquetes actúan como contenedores de datos y código.
- Todas las clases incorporadas en Java se almacenan en paquetes.
- Los paquetes son un mecanismo que permiten dar nombres y restringir la visibilidad.
- Es posible declarar clases dentro de un paquete sin hacerlas accesibles fuera del paquete.
- Es posible declarar miembros que sólo están accesibles a otros miembros del mismo paquete.

Partes de un archivo fuente

- Un archivo fuente de Java puede contener las cuatro partes internas siguientes:
  - Una única sentencia de paquete (opcional).

```
package paq1[.paq2[.paq3]];
```

Asociado a una estructura de directorios.
En este ejemplo debe existir un directorio **paq3** dentro del directorio **paq2**, dentro del directorio **paq1**.

  - Las sentencias de importación deseadas (opcional).

```
import java.util.Date;
```

  - Una única declaración de clase pública (obligatorio).
  - Las clases privadas de paquetes deseadas (opcional).
- Ejemplo

```
package java.awt.Image;
```

Niveles de acceso

- Debido a la relación entre clases y paquetes, Java distingue cuatro categorías de visibilidad entre elementos de clase:
  - `public` (accesibles en cualquier parte del programa).
  - Paquete [nivel por defecto] (accesible dentro del paquete).
  - `protected` (accesible dentro del paquete y fuera del paquete en sus subclases).
  - `private` (accesibles sólo dentro de la clase).

Importar paquetes

- Todas las clases estándares están almacenadas en algún paquete.
  - Si en una clase no se especifica el paquete, esta clase pertenece a un paquete por defecto (que no tiene nombre)
- La sentencia **`import` se usa** para que se puedan ver ciertas clases

```
import java.util.Date;
```

- o paquetes enteros.

```
import java.lang.*;
/*Este paquete se importa siempre de forma
  automática aunque no se especifique*/
```

- Sintaxis:

```
import paquete1[.paquete2].(nombre_clase|*);
```

- Una vez importada, una clase puede ser referenciada directamente, utilizando sólo su nombre.

Interfaces

Introducción

- Abstracción completa de la implementación de una clase.
- La interfaz no define la implementación.
- Métodos sin cuerpo.
- Una clase puede implementar cualquier número de interfaces.
- Una clase sólo puede heredar de una superclase (abstracta o no)
- Admiten resolución de método dinámica.

Definición de Interfaz (I)

```
acceso interface nombre{
      tipo var_final1 = valor;
      tipo var_final2 = valor;
            // …
      tipo var_finalN = valor;

      tipo_devuelto método1 (lista_de_parámetros);
      tipo_devuelto método2 (lista_de_parámetros);
            // …
tipo_devuelto métodoN(lista_de_parámetros);
}
```

acceso: o public o no se utiliza (por defecto).