Tema 2
Clases, Objetos y Métodos

Programación II

Alicia Garrido Alenda

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

lsi

¿Qué es un objeto?

- Los objetos pueden representar:
  - elementos reales ⇒ una persona, un coche
  - elementos conceptuales ⇒ una lista, un mensaje
- Un objeto se caracteriza por:
  - Sus datos.
  - Su comportamiento.
  - Su identidad.

Principio de Ocultación de la Información: La idea subyacente es que todas estas propiedades están encapsuladas en una sola entidad cohesiva → el objeto.

lsi

¿Cómo se definen los objetos? (I)

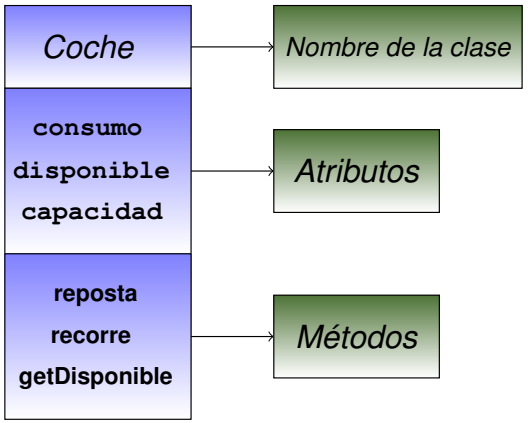
Una clase es una abstracción que define las propiedades comunes a una colección de objetos.

La clase define:

- Los atributos de los objetos: En cada objeto de una clase los atributos pueden tener diferentes valores, pero todos los objetos de una clase tienen los mismos atributos (**datos miembro o variables de instancia**).
- La interfaz de los objetos: Todos los objetos de una clase muestran la misma interfaz mediante la que se crean, consultan o modifican.
- El comportamiento de los objetos: La implementación de las operaciones de un objeto se realiza en su clase (**funciones miembro o métodos**).

lsi

¿Cómo se definen los objetos? (II)



Una clase es un tipo de datos definido por el programador → Tipo Abstracto de Datos

Definición de objeto: Instancia de una clase

Diseño y especificación de una clase

El diseño y la especificación de una clase que modele algunas propiedades de la entidad Coche

1. Poder crear objetos de la entidad Coche.
2. Añadir carburante al que hay disponible.
3. Recorrer una distancia determinada.
4. Consultar el carburante que hay disponible.

Los objetos de la clase Coche deberán proporcionar el comportamiento anterior.

Modificadores de acceso

Modificadores de acceso:

Palabras reservadas que se anteponen a la declaración de los miembros de la clase para indicar cómo acceder a dichos miembros desde el exterior de dicha clase.

- Podemos distinguir:
  - ***public***: accesibles desde el exterior de la clase utilizando la referencia al objeto.
  - **protected**: permiten que sean visibles en las subclases los miembros de las superclases y accesibles desde las clases del mismo paquete.
  - ***sin modificador***: únicamente accesible desde las clases del mismo paquete.
  - ***private***: no son accesibles desde el exterior de la clase.

Modificadores de acceso: tabla resumen

	Misma clase	Otra clase mismo paquete	Subclase otro paquete	Otra clase otro paquete
public	X	X	X	X
protected	X	X	X	
sin modif.	X	X		
private	X			

Aplicación del principio de ocultación de información

- Sólo se harán ***públicos*** los métodos que necesariamente deben ser accesibles desde el exterior de la clase.
- Las variables de instancia serán ***privadas*** o ***protegidas*** si existe una jerarquía de herencia.
- Para acceder a las variables de instancia se construyen métodos que acceden al estado para cambiarlo o mostrarlo.

Métodos constructores

Un método constructor es el invocado en la creación de un objeto.

- Es el primero que ocurre en la vida del objeto y tiene el mismo nombre que la clase en la que se define.
- No especifican tipos o valores devueltos.
- Toda clase tiene un constructor, si no se indica ninguno se asigna un constructor por defecto y sin argumentos.

*El operador **new** tiene asociado crear un objeto de la clase llamando al constructor del objeto y después devuelve una referencia al objeto creado.*

Interfaz de una clase (I)

La *interfaz* de una clase la configuran los métodos públicos de dicha clase y se define a partir de la especificación de la clase.

```
public class Coche{
// un constructor
 public Coche(double cm,double cap){
  //declaraciones y sentencias
 }
//metodos
 public double reposta(double cant){
  //declaraciones y sentencias
 }
 public boolean recorre(double distancia){
  //declaraciones y sentencias
 }
 public double getDisponible(){
  //declaraciones y sentencias
 }
}
```


Interfaz de una clase (II)

Las variables de instancia las declaramos privadas para que no sean accesibles desde el exterior de forma directa (*principio de ocultación de información*).

```
private double consumo;
private double disponible;
private double capacidad;
```

Se llaman variables de instancia ya que deben perdurar mientras viva el objeto y deben ser visibles para todos los métodos definidos en la clase.

Especificación de la clase Coche en Java

```
public class Coche
{
 private double consumo;
 private double disponible;
 private double capacidad;
 // un constructor
 public Coche(double cm,double cap)
 {//declaraciones y sentencias
 }
 //metodos
 public double reposta(double cant) {...}
 public boolean recorre(double dist) {...}
 public double getDisponible() {...}
}
```

Un objeto de la clase Coche

- Para declarar objetos de la clase Coche, una vez definida la clase haremos
  `Coche mini;`
- Crearemos entonces el objeto usando *new* y un método constructor de la clase Coche:
  `mini=new Coche(3.5,38.5);`

Métodos y comunicación entre objetos

Función: sección de código autocontenida que puede recibir datos de entrada y producir datos de salida

Abstracción funcional: las funciones se invocan para realizar una tarea, pero sin necesidad de saber cómo lo hacen.

Pero en una programación orientada a objetos los métodos son algo más

```
public class Nota{
 public static int minimo(int[] n){
 int min=Integer.MAX_VALUE;
 ...
 return min;
 }
 public static int maximo(int[] n){
 int max=Integer.MIN_VALUE;
 ...
 return max;
 }
 public static void main(String[] args){
  int [] numeros={3,7,8,9};
  int min=minimo(numeros);
  int max=maximo(numeros);
  System.out.println("Mayor="+max+"; Menor
 }
}
```


Métodos y comunicación entre objetos

Un objeto de caracteriza por:

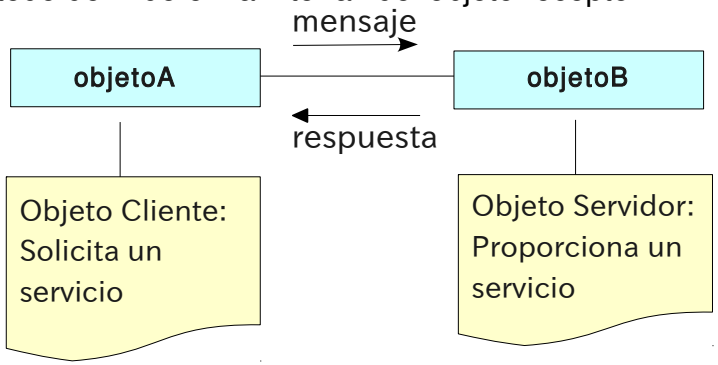
- Su identidad: Propiedad que distingue unos objetos de otros (nombre de variable).
- Su estado: Los valores que tomen en un momento dado los distintos atributos del objeto.
- Su comportamiento: Secuencia de acciones y reacciones que tienen lugar a lo largo del ciclo de vida del objeto.

Métodos y comunicación entre objetos

- Métodos: Son una sección de código autocontenida que puede recibir datos de entrada y producir datos de salida ⇒ *Abstracción funcional*.
- El comportamiento de un objeto se expresa como un *conjunto de métodos* definidos en la clase del objeto.
- ***¿Cómo accede un objeto a las operaciones de otro objeto?*** → ***Conceptualmente los objetos se comunican por medio de mensajes***.

Mensajes y comunicación entre objetos

- Si un objeto desea invocar los métodos de otro objeto debe enviarle un mensaje.
- Cada mensaje enviado a un objeto debe corresponderse con un método definido en la interfaz del objeto receptor.



Semántica de los mensajes

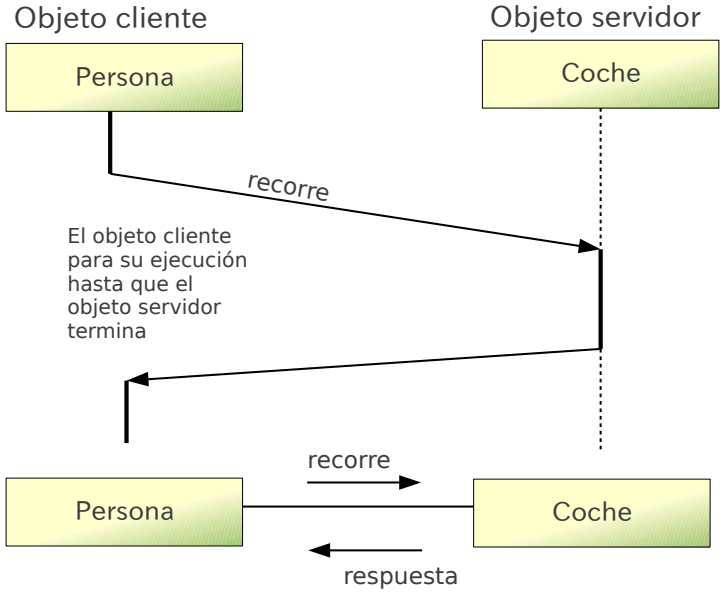
objetoServidor.metodo(argumentos)

¿Qué ocurre en la llamada?

- El objeto cliente queda a la espera de la finalización del método.
- Cuando el objeto servidor termina, el objeto cliente continúa su ejecución.
- El objeto cliente sólo conoce la interfaz del método y la funcionalidad asociada (qué hace) pero no debe saber nada acerca de cómo se hace.

Synchronous request-reply semantics

¿Cómo se comunican los objetos



¿Cómo se comunican los objetos?

```
public class Persona{
 private String nombre;
 private double edad;
 private Coche mini;
 ...

 public boolean desplaza(Lugar destino)
 {
   boolean desplazado=false;
   if((mini!=null)&&(destino!=null)){
      double distancia=destino.calculaDistancia(actual);
      desplazado=mini.recorre(distancia);
      if(desplazado)
        actual=destino;
   }
   return desplazado;
 }
}
```

Mensajes

- La invocación de un método es un mensaje → es necesario que exista un objeto receptor para el mismo
- ya que el paso de mensajes conlleva más información que la abstracción funcional:
  - el objeto siempre está presente en el mensaje;
  - las variables de instancia que se consultan/modifican en el método invocado son las del objeto

```
public double reposta(double cant){
 ...
 disponible+=cant; //disponible variable de  instancia del objeto
 ...
 return cant;
}
```

  - ¿Cómo acceder al propio objeto dentro de un método?
  - Con la variable **this**

```
public double reposta(double cant){
 ...
 this.disponible+=cant;//this es el objeto sobre el que
 // se invoca el metodo; es equivalente a disponible+=cant;
 ...
 return cant;
}
```


Mensajes

- El hecho de necesitar que exista un objeto receptor para un mensaje a a veces puede ser:
  - **Excesivo**: Deseamos ejecutar un método para obtener un resultado sin crear un objeto, por ejemplo para calcular la raíz cuadrada de un número.
    No podemos hacer

```
y = sqrt(x)
```

    deberíamos hacer

```
y = objeto.sqrt(x)
```


Mensajes

- Otras puede resultar:
  - Inconveniente: Tenemos un juego que modela la realidad de manera que los `guerreros` de una tribu (*objetos* de la clase `Guerrero`) se comportan de una manera o de otra en función del número de guerreros de su tribu que quedan vivos.
    - ★ Es necesario llevar la cuenta de los `guerreros`:

```
int fuerzas;
```

    - ★ Es necesario mantener de forma consistente el valor de `fuerzas` en cada uno de los `guerreros`.
  - Posibles soluciones:
    - ★ Cada vez que un `guerrero` muere o se incorpora a la tribu se lo notifica a todos los demás (envia un mensaje a todos los objetos) ⇒ complejo
    - ★ Existe un objeto supervisor o controlador que realiza tales notificaciones ⇒ ineficiente

Mensajes a clase

- La clase modela como deben ser los objetos, pero puede ser algo más:
  - Puede mantener información global sobre sí misma o sus instancias.

Atributos static:

- ★ Atributos que se mantienen a nivel de clase, no a nivel de objeto.
- ★ Mantienen información sin necesidad de crear un objeto.

  - Puede ser un receptor de mensajes.

Métodos static:

- ★ Modelan mensajes cuyo receptor es la clase, no sus objetos.
- ★ Necesarios cuando se quiere invocar un método sin crear un objeto.

Atributos y métodos **static**

- Los métodos estáticos tienen limitaciones, sólo pueden:
  - Invocar métodos estáticos.
  - Acceder a atributos estáticos.

¿Se pueden invocar métodos estáticos desde métodos de instancia?

¿Se puede acceder a atributos estáticos desde métodos de instancia?

Atributos y métodos **static**

Los métodos se invocan usando el nombre de la clase para referenciar al receptor del mensaje.

```
int vivos=Guerrero.consulta();
```

```
public class Guerrero{
 private static int fuerzas=0;
 public static int consulta(){
   return fuerzas;
 }
 public void cae() { fuerzas--; }
 public void incorpora() { fuerzas++; }
 public void lucha(){
  if (fuerzas>13)
   // Ataca
  else
   // Defiende o Huye
 }
}
```

Sólo hay una variable `fuerzas` a la que pueden acceder todos los objetos `guerrero`

Atributos y métodos **static**

Los métodos se invocan usando el nombre de la clase para referenciar al receptor del mensaje.

```
Guerrero.consulta();
```

```
public class Guerrero{
 private static int fuer
 public static int consu
   return fuerzas;
 }
 public void cae() { fue
 public void incorpora()
 public void lucha(){
  if (fuerzas>13)
   // Ataca
  else
   // Defiende o Huye
 }
}
```

```
public class Nota{
public int minimo(int[] n){...}
public void calcula(int[] n){
int min = minimo(n);
   ...
}
public static void main(String[] args){
int[] numeros={ ... };
int min=minimo(numeros);//Error
       // ...
 }
}
```


Ejemplo: La clase Math

Todos los métodos de la clase son `static`

Métodos	Descripción
`abs(x)`	Valor absoluto. Sobrecargado para `int`, `float` y `double`.
`floor(x)`	Mayor valor entero menor o igual que x.
`round(x)`	Valor entero más cercano a `x`[a].
`exp(x)`	`e` elevado a `x`.
`log(x)`	Logaritmo natural de `x` (base `e`).
`max(x,y)`	El mayor de `x` e `y`. Sobrecargado.
`min(x,y)`	El menor de `x` e `y`. Sobrecargado.
`pow(x,y)`	`x` elevado a `y`.
`sqrt(x)`	Raíz cuadrada.
`random()`	Devuelve `double` aleatorio entre `0.0` y `1.0`.

[a] El resultado de `(int)Math.floor(x + 0.5)`

- Constantes de uso común
  `Math.PI` `3.141592...` el double más cercano a π
  `Math.E` `2.718281...` el double más cercano a e

Relaciones entre objetos

- Los objetos de un programa interactúan entre sí para conseguir su objetivo.
- Estas interacciones vienen dadas por el tipo de relación que se establece entre los distintos objetos.
- Los tipos de relación más generales entre objetos son:
  - Asociación
  - Composición (agregación)
  - Uso

Relación de Asociación

- Expresa una conexión (unidireccional o bidireccional) entre los objetos de las clases implicadas en la asociación. Ejemplos de asociación pueden ser:
  - Persona $\rightarrow$ Dinero
  - Frigorífico $\rightarrow$ Alimento
  - Estudiante $\leftrightarrow$ Proyecto
- Los objetos asociados entre sí existen de forma independiente.
- La creación o desaparición de un objeto asociado a otro implica únicamente la creación o destrucción de la relación entre ellos (nunca afectará a la creación o destrucción del otro objeto).

Ejemplo de asociación en Java

```
class Alumno{
 private String dni;
 ...
 private Trabajo project;

 public Alumno(){
  // Implementacion del
  // constructor
 }
 public void setTrabajo(Trabajo p){
  project=p;
 }
}
```

```
import java.util.ArrayList;
class Trabajo{
 private String codigo;
 private ArrayList<Alumno> workers;
 ...
 public Trabajo(){
  // Implementacion del contructor
  workers=new ArrayList<Alumno>();
  ...
 }
 public boolean setAlumno(Alumno w){
  if(w!=null){
    ...
    workers.add(w);
    return true;
  }
  return false;
 }
}
```


Composición (agregación) de objetos

La ***composición*** es uno de los mecanismos de abstracción que usa el paradigma orientado a objetos para facilitar el modelado de sistemas: ***Una clase tiene objetos de otras clases como miembros.***

- Es una manera de reutilizar el software y representa la relación *Todo-Parte* en la que un objeto forma parte de la naturaleza del otro.
- *Asociación* vs *Todo-Parte*:
  - Asimetría: Si un objeto **A** está compuesto de un objeto **B**, **B** no puede estar compuesto por **A**.
  - Transitividad: Si un objeto **A** está compuesto de un objeto **B**, y **B** está compuesto por **C**, entonces **A** también está compuesto por **C**.

Composición (agregación) de objetos

Hay dos tipos de composición:

Agregación *Biblioteca→Libro*	**Composición** *Libro→Capítulo*
Las partes pueden existir aunque no exista el todo (y viceversa).	Las partes y el todo no existen de forma independiente.
Las partes pueden compartirse entre diferentes propietarios.	Las partes NO pueden compartirse entre diferentes propietarios.
Las partes pueden ser añadidas y eliminadas del todo.	Si el todo es eliminado, también son eliminados sus objetos parte.

Pero los lenguajes de programación no soportan directamente esta diferencia.

Ejemplo de composición en Java

```
import java.util.ArrayList;
public class Casa{
 private class Habitacion{
   private String color;
   private boolean luz;
   public Habitacion(){
     color=new String("blanco");
     luz=false;
   }
   public void enciende(){luz=true;}
   public void apaga(){luz=false};
   public boolean getLuz(){return luz;}
   ....
 }
 private ArrayList<Habitacion> estancias;
 public Casa(int n){
   if(n<3) n=3;
   estancias=new ArrayList<Habitacion>();
   for(int i=0;i<n;i++)
     estancias.add(new Habitacion());
 }
 public boolean ilumina(int i){
  boolean encendido=false;
  if(i>=0 && i<estancias.size() && !estancias.get(i).getLuz()){
    estancias.get(i).enciende();
    encendido=true;
  }
  return encendido;
 }}
```


Relación de Uso (Dependencia)

- Un objeto de la clase **A** usa una clase **B** cuando no contiene variables de instancia del tipo **B** pero:
  - Utiliza alguna instancia de la clase **B** como parámetro o variable local en alguno de sus métodos.
  - Utiliza algún método *static* de la clase **B**.
- Ejemplos de relación de uso pueden ser:
  - Persona → Peluquería
  - Coche → Gasolinera
  - Estudiante → Autobús

Recogida de basura

Cuando se termina de utilizar un objeto entonces ya no es referido por ningún otro. Esto ocurre cuando:

- Se cambia la referencia a la de otro objeto o a null (vacío).
- Se retorna de un método, con lo que se pierden las variables locales.

- Los objetos no referenciados se llaman **basura** y el proceso de buscarlos y recogerlos se llama **recogida de basura** (*garbage collection*).
- La máquina virtual de Java recoge automáticamente la basura liberando la memoria y garantiza la permanencia de los objetos referenciados en algún sitio.
- La recogida de basura resuelve automáticamente el problema de las pérdidas de memoria (memory leak).

Interfaces

Con las ***interfaces*** el programador puede definir un tipo de un modo abstracto en forma de colección de métodos y atributos (el contrato del tipo definido).

Las interfaces no contienen implementación: todos los métodos son ***abstractos*** (vacíos) y todos los atributos son constantes.
No se pueden crear instancias de una interfaz: son las clases las que expanden el tipo implementando los métodos.

```
class unaClase implements unaInterfaz {
...
}
```

Una interfaz es un ***elemento de diseño*** mientras que una clase es una mezcla de diseño e implementación.

Interfaces

- Una interfaz puede tener tres tipos de miembros:
  - Constantes.
  - Métodos.
  - Clases e interfaces anidadas.
- Todos los miembros de una interfaz son públicos aunque se omita el modificador:
  - Las *constantes* son implícitamente `public`, `static` y `final`.
  - Los *métodos* de una interfaz son implícitamente `abstract` (no tienen cuerpo).
- Las interfaces se pueden extender con la palabra `extends` desde una o más interfaces (lo veremos en detalle en el tema de *herencia*).

Ejemplo de interfaz en Java

```
public interface Arma{
 public boolean carga();
 public boolean dispara();
 public boolean cargada();
}
```

```
public class Gun implements Arma{
 private int balas,cargador;
 public Gun(int carga){
  balas=0;cargador=carga;
 }
 public boolean carga(){
  if(balas<cargador){
   balas+=1;
   return true;
  }
  return false;
 }
 public boolean dispara(){
  if(balas>0){   balas-=1;
   System.out.print("Pum");
   return true;
  }
  return false;
 }
 public boolean cargada(){
  if (balas>0) return true;
  return false;
 }
}
```

```
public class Bazooka implements Arma{
 private boolean cohete;
 public Bazooka(){
  cohete=false;
 }
 public boolean carga(){
  cohete=true;
  return true;
 }
 public boolean dispara(){
  if(cohete){
   System.out.print("fffsssshhh");
   cohete=false;
   return true;
  }
  return false;
 }
 public boolean cargada(){
  return cohete;
 }
}
```


Ejemplo de interfaz en Java

```
public class Cyborg{
 private Arma acoplada;
 public Cyborg(Arma tipo){
  acoplada=tipo;
 }
 public boolean ataca(){
 if (acoplada.cargada()){
  return acoplada.dispara();
 }
 else
  System.out.println("Es necesario
        recargar el arma");
 return false;
 }
 public void recarga(){
  acoplada.carga();
 }
 public void acopla(Arma nea){
  acoplada=nea;
 }
}
```

```
public class Principal{
public static void main(String args[]){
 Cyborg replicant;
 Arma gral=new Bazooka();
 replicant=new Cyborg(gral);

 if (!replicant.ataca()){
  replicant.recarga();
  replicant.ataca();
 }
}
```