Programación
Tema 6:Encapsulación en clases

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Clases (1/2)

- Las clases soportan varios conceptos:
  - Definición de tipos de datos
    - Definen los datos (campos) y comportamiento (métodos) para representar conceptos tratados del programa
  - Espacios de nombres
    - La clase define un ámbito de nombres reconocibles en todo su espacio
  - Encapsulamientos
    - Las clases permiten agrupar datos y comportamientos relacionados

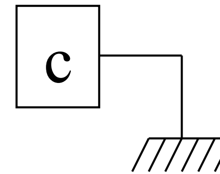
Clases (2/2)

- Las clases son las entidades fundamentales del desarrollo de los programas
- Los objetos son las entidades fundamentales de la ejecución
  - En Java solo accedemos a los objetos mediante las referencias
  - Un programa no incluye especificación de objetos, solo se crean durante su ejecución
  - Todos los objetos tienen asociada una clase que es la que se emplea en su creación

Ciclo de vida del objeto

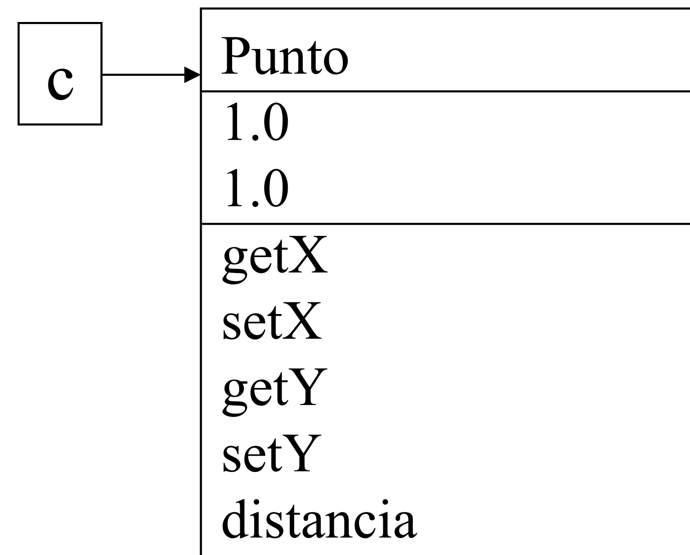
- Declaración: Crear una referencia para el objeto

```
Punto c; // inicialmente null
```



- Creación: Crear el objeto mediante new y llamada a un método constructor

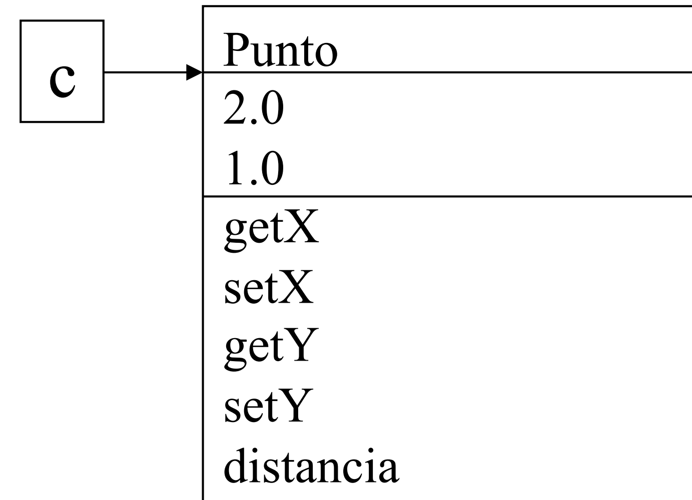
```
c = new Punto(1.0, 1.0);
```



Ciclo de vida del objeto

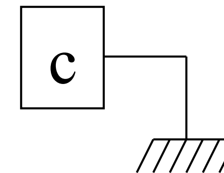
- Manipulación: Usar el objeto mediante su referencia: llamar a sus métodos, usar sus datos, pasarlo como parámetro, etc.

  c.setX (2.0);



- Destrucción: Olvidar el objeto cuando no haga falta; cuando no tiene referencias, el recolector de basura lo destruye

  c = null;



Referencia `this`

- Es una variable predefinida en todos los ámbitos de una clase (salvo los métodos `static`)
- El tipo de `this` es la clase en la que aparece
- Es una referencia al objeto sobre el que se ejecuta el método. Esa referencia no es modificable (el objeto si)
- Podemos usarla para:
  - Pasar referencias de este objeto mediante parámetros de métodos (de esta u otras clases)
  - Asignaciones a variables (locales o atributos)
- Cuando referenciamos campos y métodos sin usar referencia, usamos implícitamente `this`
  - `this.nombreCampo` equivale a `nombreCampo`
  - `this.nomMetodo(arg)` equivale a `nomMetodo(arg)`

Constructor `this`

- Todo objeto tiene un método que se llama this
  - Representa cualquiera de sus constructores (según su signatura)

```
public class Punto {
    private double x, y;

    public Punto (double x, double y) { ... }
    public Punto (double x) {this(x,0.0); }
    public Punto (Punto q) {this(q.x,q.y); }
}
```

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Ámbitos

- El cuerpo de una clase determina un ámbito
  - Métodos y campos que declara son visibles en toda la clase
- Un método (incluido constructores) determina otro ámbito
  - Los parámetros son visibles en el ámbito del método
- Un bloque de código es otro ámbito
  - Bloques de código
    - Cuerpo de métodos y constructores
    - Cuerpos de sentencias
  - La declaración de variables es visible en las sentencias que siguen a la declaración

Ámbitos

- Un elemento es visible en su ámbito y en los ámbitos que incluye
- Cuando dos elementos tienen el mismo identificador la declaración mas interna esconde la externa dentro del ámbito interno
  - En el mismo ámbito no podemos sobrecargar el mismo identificador/signatura

```
int x = 0;
public int m(int x) {
    x=4;
    this.x=6;
    return x;
}
```

Paquetes

- Los paquetes son una forma de agrupar clases relacionadas y con fuertes dependencias
  - Una encapsulación de alto nivel
  - Una forma de organizar las clases
    - java.lang
    - java.awt, java.awt.color, java.awt.event
- Los paquetes son espacios de nombres
  - Para acceder a una clase hay que hacer visible el espacio de nombres del paquete en el que se encuentra o definir todo el camino hasta llegar a la clase

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

Paquete de una clase

- Al principio del fichero que contiene una clase (antes de declarar la clase) definimos el paquete que contiene la clase

  ```
  package nombre1.nombre2. ... nombren;
  ```

- Un único paquete para todo el fichero
- Cuando compilamos (salvo opciones de compilación específicos) se crea un directorio para cada nombre del camino y un directorio contiene a los nombres que le siguen
- Dentro del mismo paquete no pueden existir dos clases o paquetes con el mismo nombre

Visibilidad de paquetes y clases

- Una clase debe ser visible para poder usarla
- Las importaciones las incluimos al principio del fichero
  - Importación de una clase

    ```
    import nomPaquete1.nomPaquete2....NomClase;
    ```

    - Podemos referenciar la clase (NomClase) libremente
  - Importación de todo el paquete

    ```
    import nomPaquete;
    ```

    - Para poder usar una clase hay que cualificarla `nomPaquete.nomClase`
  - Importación sin uso qualificado

    ```
    import nomPaquete1.nomPaquete2....*;
    ```

    - Son visibles todas las clases incluidas en ese paquete
- Si no importamos, hay que definir todo el camino:

  ```
  java.util.Date d = new java.util.Date();
  ```

  Necesario cuando utilizamos varias clases con el mismo nombre en paquetes diferentes

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Características de clase

- En general los campos y métodos son de objeto
  - Métodos se ejecutan en el contexto de un objeto
  - Cada objeto tiene valores específicos para sus campos
- Hay métodos y campos que no están asociados a objetos y que están asociados a la clase
  - Los campos de clase son únicos
    - La comparten todos los objetos de clase
    - Las modificaciones son visibles y afectan todos los objetos de la clase
  - El modificador `static` hace un campo o método de clase
  - No es necesario un objeto para referenciarlos

```
NombreClase.campoStatic
NombreClase.metodoStatic(arg)
```

  - Dentro de la clase, no es necesario el nombre de la clase

Elementos de clase

- ¿Cuando definir elementos de clase?
  - cuando son campos constantes (`static final`)
    ```
    public static final double PI = 3.14;
    ```
  - variables únicas y comunes para todos los objetos de esa clase
    ```
    class Punto {
       public static Punto origen = new Punto(0.0,0.0);
    ...}
    ```
  - métodos que necesitemos utilizar aunque no haya objetos
    - Métodos en la clase `Math`
  - métodos que sólo utilizan elementos de clase
  - `main`, para poder entrar en el programa sin más

Características de clase

- En métodos static no podemos usar this, y solo podemos usar campos static. ¿Por qué?
  - La llamada puede llegar de un contexto que no es de ningún objeto
  - La llamada puede llegar del contexto de un objeto de otra clase
- Campos static se deben inicializar en la declaración o
- Mediante bloques static:

```
class C {
   static int i=0;
   static {   // solo podemos acceder a statics
              // podemos incluir bucles y alternancias
   }
...}
```

Método main

- main es la forma *estándar* para entrar en un programa que no incluye ningún objeto todavía
  - Si no fuese static, primero habría que crear un objeto, y se ejecutaría el constructor
- La máquina virtual inicializa la ejecución y busca main para cederle control; ejecuta NombreClase.main(args)
- Por eso main debe ser un método de clase, hay que respetar sus argumentos y retorno, y debe ser público

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Relaciones entre Clases

- Las clases son un mecanismo complejo para soportar la reutilización del software (no reinventar la rueda)
- Las clases integran funcionalidades para tratar al tipo de concepto que soportan
- Una clase compleja se puede construir a partir de clases simples
- Tipos de relaciones entre clases
  - Uso
  - Composición
  - Agregación
  - Dependencia
- En Java las implementamos todas mediante `import`, referencias a clases y referencias a objetos

Relaciones Uso

- Utilizamos los métodos de otra clase para poder implementar algunos métodos
- Hay que importar la clase, y debemos tener algún tipo de referencia (campos o variables locales), o que los métodos sean `static`
- Una clase puede utilizar los métodos definidos en otra, aunque no incluya campos que la referencien

```
import ClaseUsada;
class ClaseQueUsa {
        public void metodo(ClaseUsada usada) {
            usada.metodoUsado();
            (new ClaseUsada()).metodoUsado();
        }
}
```

Relaciones Composición y Agregación (1/2)

- En una relación de composición construimos un tipo de objeto mas complejo mediante un conjunto de objetos simples
  - Por ejemplo, clase Ascensor a partir de las clases: Motor, Botones, Display, Puerta
  - Por ejemplo, clase Escuela a partir de las clases: Secretaria, Departamento, Direccion, Curso, Estudiante
  - Los métodos de la clase compleja se implementan con llamadas a los objetos simples:
    - Por ejemplo el método irPiso(int i) de Ascensor se implementa llamando a cerrar(Puerta), arrancar(Motor), parar(Motor), abrir(Puerta), setValor(Display)

Relaciones Composición y Agregación (1/2)

- La clase *todo* incluye `import` y campos referencia a las clases *parte*
- En una composición pura la clase *todo* no comparte los objetos *parte* (dos ascensores no referencian el mismo motor)
  - La clase *todo* construye los elementos *parte*
- En una agregación la clase *todo* puede compartir las *partes* (una escuela puede compartir departamentos con otras escuelas)
  - Si comparte, el *todo* no siempre construye las *partes*
- Un campo referencia no tiene por que ser ni composición ni agregación. Por ejemplo, `Cursos` que cursa un `Estudiante`, en la clase `Estudiante`

Otras Relaciones

- Una clase puede depender de otra aunque no llame a sus métodos o no incluya referencias a ella
- Algunas situaciones:
  - Capturamos excepciones
  - Declaramos parámetros para poder pasar los objetos en llamadas a otra clase
- En algunos casos hay que incluir `import` (por ejemplo para utilizarla como tipo)

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Estilo: nombrado

- Clases, Interfaces: Mayúscula-Mayúscula
  - Ej. Punto, PolinomioGrado2
- Métodos: minúscula-Mayúscula
  - Ej. getX(), distancia(), buscaMayor()
- Atributos: minúscula-Mayúscula
  - Ej. nombrePropio, x, y, apellidos
- Constantes: MAYÚSCULA + “_”
  - Ej . ORIGEN_X

Comentarios

- Comentarios de código
  - /* ... */
  - // comentario táctico: hasta fin de línea
- Comentarios de documentación
  - /**

    **/
- Herramienta javadoc para generar páginas HTML de documentación
  - al principio de cada clase
  - al principio de cada método
  - ante cada campo

Comentarios de clase

- Tras los “imports” y antes del “class”

```
import java.util.List;

/**
 * Esta clase representa...
 * @author Pepe Pérez
 * @version 2 (22-2-2004)
**/

public class Autor {
}
```

Comentarios de documentación de clase

- *@author* nombre del autor
- *@version* identificación de la versión y fecha
- *@see* referencia a otras clases y métodos

- *@since* indica desde qué versión o fecha existe esta clase o interfaz en el paquete
- *@deprecated* esta clase no debería usarse pues puede desaparecer en próximas versiones

Comentarios de atributos

- Justo antes de su definición

```
/**
 * Nombre del alumno
**/
private int nombre;
```

- @*since* indica desde qué versión o fecha existe este atributo en la clase
- @*deprecated* este atributo no debería usarse pues puede desaparecer en próximas versiones

Comentarios de métodos

- Justo antes de su definición

```
/**
 * Inserta elemento en la lista el número de veces indicado. Lanza una
   excepción si la lista está llena y devuelve la posición del primer
   elemento insertado.
 * @param elemento Elemento que se desea insertar
 * @param repetidos Int con el número de veces que se inserta
 * @throws Exception si lista está llena
 * @return Posición del primer elemento
**/
int inserta(Elemento elemento, int repetidos) throws Exception {}
```

Comentarios de documentación de métodos

- *@param* <nombre del parámetro> descripción de su significado y uso
- *@return* descripción de lo que se devuelve
- *@exception* <nombre de la excepción> excepciones que pueden lanzarse
- *@since* indica desde qué versión o fecha existe este constructor o método en la clase
- *@deprecated* este método no debería usarse pues puede desaparecer en próximas versiones

Contenidos

- 1. Clases
- 2. Visibilidad-derechos de acceso
- 3. Elementos estáticos
- 4. Relaciones de uso y composición de clases
- 5. Estilo y documentación
- 6. Diseño de clases

Diseño de clases

- Identificar clases
- Definir las cabeceras de métodos
- Especificar los atributos
- Rellenar cuerpos de los métodos con sentencias

Identificar clases

- Un solo concepto: número de lotería, persona, Punto
- Técnicas para identificar clases
  - Almacena información sobre el estado de un objeto: bombilla, semáforo, conexión web, serie aritmética
    - Atributo para el estado
    - Métodos para consultar y cambiar de estado: encender(), apagar(), siguienteColor(), siguienteValor(), conectar()
  - Agregación/composición: Localidad
    - Junta objetos de diferentes tipos
  - Almacén: Trayectoria
    - Junta objetos del mismo tipo
  - Herencia-concepto derivado: Coordenada geográfica
  - Factoría (fábrica): GeneradorPuntos
    - Crea objetos de otra clase

Métodos

- Siempre: constructor, get/set, equals, toString
  - Normalmente públicos
- Un único objetivo por método:
  - devuelve un resultado o cambia una cosa
- Métodos pequeños: 10 líneas
  - partir en otros más pequeños hasta llegar a básicos
- Métodos de consulta (get)
  - Devuelven un resultado, pueden lanzar excepciones
- Métodos de cambio (set)
  - Información de cambio en los parámetros
  - Depende del caso, devuelven un código de resultado, pueden lanzar excepciones

Atributos

- Un atributo contiene la información del objeto que es necesario guardar
- Los atributos coinciden con la información esencial necesaria del objeto
- El menor número de atributos
  - no poner como atributo una variable automática
  - no poner como atributo el resultado de un cálculo-método
  - No poner como atributo un parámetro de un método

Cuerpos de métodos

- Patrón para definir el cuerpo de un método
  - Procesado/comprobación de parámetros, lanzamiento de excepciones
  - Copiar los parámetros si es necesario; si no, manejar referencias
  - Transformaciones o cálculos
  - Devolución de resultado
- Variables automáticas
  - En el ámbito más interno posible
- Sentencias
  - Bucles mínimos, sacar fuera todo lo posible, atención a los límites
  - Salir cuanto antes con return