Prácticas de Programación

Tema 6. Modularidad y abstracción: aspectos avanzados

6.1 Paquetes

Un paquete en Java es un conjunto de clases que están en el mismo directorio

- **evita colisiones de nombre**
- **permite agrupar clases relacionadas entre sí**
- **permite organizar el código mejor**

Ya hemos usado algunos paquetes:

- **fundamentos**
- **`java.util`: contiene las clases `Arrays`, `ArrayList`, ...**
- **...**

El directorio de trabajo se considera como un paquete sin nombre

Desde una clase pueden usarse directamente otras del mismo paquete

Paquetes con nombre

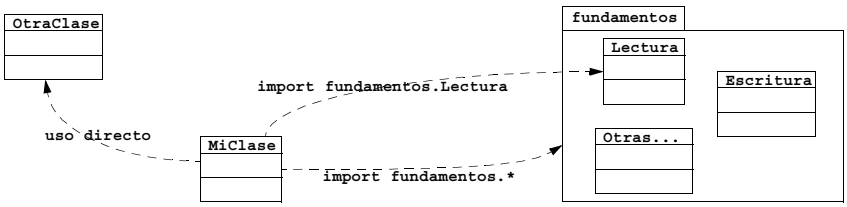
- se guarda en un directorio de igual nombre que el paquete
- el directorio que le contiene debe estar en la ruta de búsqueda
  - en `libraries` en `Bluej`
- se pueden especificar subpaquetes (subdirectorios):
  - notación: `paquete.subpaquete`
- las clases de un paquete con nombre llevan como primera línea:

```
package nombrepaquete;
```

- el nombre de un paquete se suele poner en minúsculas
  - evita problemas de portabilidad entre sistemas operativos que distinguen o no minúsculas y mayúsculas

Para usar clases de un paquete con nombre se usa `import`:

```
import fundamentos.Lectura;   // permite utilizar
                              // la clase Lectura
import fundamentos.*; // permite utilizar todas
                      // las clases del paquete
```



Existe un paquete especial, el paquete `java.lang`

- contiene las clases básicas del lenguaje Java: `Object`, clases envoltorio (`Integer`, `Double`, ...), `String`, `StringBuilder`...
- siempre está disponible: no hace falta utilizar `import`

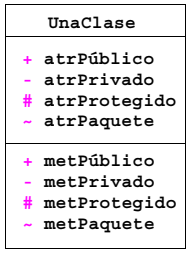
Modificadores de acceso y paquetes

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



6.2 Tipos de módulos de programa

Es habitual encontrarse estos tres tipos de clases:

- ***clases sin métodos*: son simples contenedores de datos**
  - **Ejemplo: clase con las coordenadas de un punto en el espacio**

```
public class Punto {
  public double x, y, z;
}
```

- ***clases sin estado*: son contenedores de métodos relacionados entre sí, y quizás también constantes estáticas**
  - **Se llaman también librerías**
  - **Ejemplo: la clase `Math`**
- ***clases con datos privados y métodos públicos*: se llaman también tipos abstractos de datos**

Tipo abstracto de datos (ADT)

La mayoría de las clases que hemos hecho durante el curso son ADTs

- **clases `Alumno`, `Curso`, `Paciente`, `Complejo`, ...**

Un ADT sirve para encapsular

- **una parte privada que almacena el estado del objeto**
  - **implementada mediante los atributos privados**
- **una parte pública que proporciona las operaciones que es posible realizar con el tipo de datos**
  - **formada por los métodos públicos**

Una de las principales ventajas de utilizar ADTs es que facilitan la *reutilización del código*

- **como veremos, la posibilidad de definir ADTs genéricos facilitará aún más la reutilización del código**

6.3 Módulos genéricos

Algunas estructuras de datos tienen un comportamiento independiente de los objetos sobre los que operan

- **p.e. el comportamiento de una lista (insertar, extraer, ver su tamaño, ...) es independiente del tipo de objetos que contiene**

Lo ideal sería poder escribir el código de la clase independientemente del tipo de objetos que vaya a contener

- **permite reutilizar el código en distintas aplicaciones**

Java nos proporciona un mecanismo para escribir clases genéricas

- **al escribir su código, la clase sobre la que operan queda indeterminada**

Al crear un objeto a partir de una clase genérica

- **es preciso indicar la clase que quedó indeterminada**

Clases genéricas

Declaración de una clase genérica:

```
public class ClaseGenérica <T> {
  // se pueden definir atributos y métodos como
  // en cualquier clase no genérica
  ...
}
```

- `T` es el parámetro genérico: clase indeterminada de objetos sobre la que opera `ClaseGenérica`

Por convenio, el nombre del parámetro genérico suele ser una letra mayúscula:

- `E`: tipo del elemento en una lista, cola, ... (muy usado en las *Java Collections*)
- `T`: tipo
- ...

Ejemplo de clase genérica

```
/**
 * Almacén genérico
 */
public class Almacén <T> {

    // objeto almacenado (de tipo indeterminado)
    private T contenido;

    /** Mete un dato en el almacén */
    public void almacena(T dato) {
       contenido=dato;
    }

    /** Obtiene el dato almacenado */
    public T obtiene() {
        return contenido;
    }
}
```


Instanciación de clases genéricas

Cuando se crea (se *instancia*) un objeto de una clase genérica hay que especificar el parámetro genérico

```
// crea un almacén de coches
Almacén<Coche> ac=new Almacén<Coche>();
// crea un almacén de doubles
Almacén<Double> ad=new Almacén<Double>();
```

Un objeto de una clase genérica

- **puede usarse con objetos de la clase del parámetro genérico o de sus subclases**
- **p.e. en una lista genérica instanciada para la clase `Vehículo` podríamos meter objetos de sus subclases `Coche`, `Barco`, `Avión`, ...**

Ejemplo de uso de clases genéricas

```
public static void main (String[] args) {
  // crea dos objetos
  Double d=new Double(5.0);
  Coche c=new Coche("ibiza",Color.ROJO);
  // crea un almacén de coches
  Almacen<Coche> ac=new Almacen<Coche>();
  // crea un almacén de doubles
  Almacen<Double> ad=new Almacen<Double>();
  // almacena objetos
  ac.almacena(c);
  ad.almacena(d);
  ac.almacena(d); // ¡¡error de compilación!!
  // obtiene los contenidos de los almacenes
  Coche c1=ac.obtiene();
  Double d1=ad.obtiene();
}
```


Parámetros genéricos restringidos

En ocasiones puede ser interesante restringir el tipo del parámetro genérico

- **de forma que sólo pueda ser una clase o cualquiera de sus subclases**

Ejemplo:

```
public class Garaje <T extends Vehículo> {
  ... igual que la clase almacén ...
}
```

- **`T` sólo puede ser la clase `Vehículo` o cualquiera de sus subclases `Coche`, `Barco`, `Avión`, ...**

Esto es un error de compilación:

```
Garaje<Double> g=new Garaje<Double>(); //¡¡ERROR!!
```


6.4 Introducción a las Interfaces

En ocasiones es interesante poder asegurar que una clase verifica un "*contrato*"

- **implementa un conjunto de métodos**

Con este propósito Java define el concepto de interfaz

- **una interfaz es similar a una clase abstracta que sólo puede contener constantes y cabeceras de métodos**
- **no es posible crear objetos de una interfaz (igual que ocurría con las clases abstractas)**

Ejemplo de interfaz

```
public interface PuertoCom {
  void envía(String str);
  String recibe();
}
```

Clase que *implementa* la interfaz:

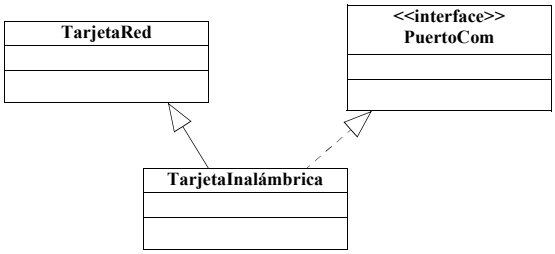
```
public class PuertoSerie implements PuertoCom {
  @Override
  public void envía(String str) {
    código de envía
  }
  @Override
  public String recibe(){
    código de recibe
  }
  ... otros métodos ...
}
```


Interfaces y herencia múltiple

La herencia múltiple no está permitida en el lenguaje Java

- **las interfaces proporcionan una alternativa**

```
public class TarjetaInalámbrica
                  extends TarjetaRed
                  implements PuertoCom {
  ...
}
```



Las interfaces se verán en mayor detalle en EDA (2º curso)

6.5 Programación con módulos predefinidos

Hay diversas *clases genéricas* predefinidas para almacenar datos con diversas estructuras: *Java Collections*

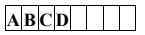
Nombre	Descripción
`ArrayList`	Tabla de tamaño que puede crecer
`LinkedList`	Lista de tamaño indefinido También sirve para definir colas
`HashSet`	Conjunto
`HashMap`	Mapa

Estudiaremos las dos más utilizadas: `ArrayList` y `LinkedList`

- **ambas implementan la interfaz `List`:**
  - **operaciones para insertar, extraer y buscar elementos**
  - **las inserciones y extracciones pueden realizarse en cualquier posición de la lista**

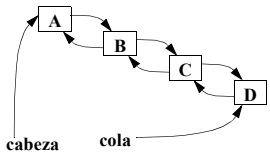
Comparación entre `ArrayList` y `LinkedList`

- `ArrayList`: lista implementada con un array
  - Acceso posicional eficiente (`O(1)`)
  - Inserción y extracción costosas (`O(n)`)
    - menos en la última posición que es `O(1)`
  - Cuando se supera el tamaño del array, se crea uno nuevo más grande y se copian en él los elementos del antiguo



numElementos=4

- `LinkedList`: lista doblemente enlazada
  - Acceso posicional costoso (`O(n)`)
  - Inserción y extracción costosas (`O(n)`)
    - menos en la primera y última posición que es `O(1)`
  - Tamaño ilimitado



Uso de `LinkedList` y `ArrayList`

Debe importarse el paquete `java.util`:

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

Son clases genéricas

El constructor crea una lista vacía

```
LinkedList<MiClase> lista=
                    new LinkedList<MiClase> ();
```

Con las colecciones se puede usar el lazo `for-each`

```
for(MiClase elem: lista) {
  ... utiliza elem ...
}
```

En las operaciones que se muestran en las trasparencias siguientes la clase `E` es el parámetro genérico

Interfaz `List`: Operaciones de modificación

Descripción	Interfaz
Añade `o` al final de la lista. Retorna `true`	`boolean add (E o)`
Busca el primer elemento de la lista igual a `o` y lo elimina. Retorna `true` si existe (para encontrar el objeto utiliza su método `equals`)	`boolean remove (E o)`
Reemplaza el elemento de la lista que ocupa la posición `index`, por `element`, y retorna el elemento que estaba ahí	`E set(int index, E element)`
Inserta `element` en la posición `index`, "desplazando" los elementos posteriores	`void add (int index, E element)`
Elimina (y retorna) de la lista el elemento que ocupa la posición `index`, "desplazando" los posteriores	`E remove (int index)`
Hace la lista vacía	`void clear()`

(Las operaciones que usan `index` pueden lanzar `IndexOutOfBoundsException` si el valor de `index` no es válido)

Interfaz `List`: Obtener información

Descripción	Interfaz
Número de elementos de la lista	`int size()`
¿Está vacía?	`boolean isEmpty()`
Retorna true si la lista contiene a `o` al menos una vez	`boolean contains(Object o)`
Retorna el elemento de la lista que ocupa la posición `index`	`E get(int index)`
Búsqueda: retorna el índice de la primera aparición de `o`, o -1 si no existe	`int indexOf(Object o)`
Búsqueda: retorna el índice de la última aparición de `o`, o -1 si no existe	`int lastIndexOf(Object o)`

(Los métodos `contains`, `indexOf` y `lastIndexOf` utilizan el método `equals` para localizar el objeto)

(La operación `get()` puede lanzar `IndexOutOfBoundsException` si el valor de `index` no es válido)

Operaciones de específicas de `LinkedList`: Acceso al principio o el final

Descripción	Interfaz
Retorna el primer elemento	`E getFirst()`
Retorna el último elemento	`E getLast()`
Elimina y retorna el primer elemento	`E removeFirst()`
Elimina y retorna el último elemento	`E removeLast()`
Añade o al principio de la lista	`void addFirst(E o)`
Añade o al final de la lista	`void addLast(E o)`

(`getFirst()`, `getLast()`, `removeFirst()` y `removeLast()` lanzan `NoSuchElementException` si la lista está vacía)

Método `toString` de las *Java Collections*

Las *Java Collections* (`ArrayList`, `LinkedList`, `HashMap`, ...) redefinen el método `toString` de `Object`

- **el método `toString` muestra la colección con el formato**

  `[elemento 0, elemento 1, ..., elemento n-1]`

- **a su vez cada elemento se muestra invocando su propio método `toString`**

Ejemplo `ArrayList`: Clase `ParqueMóvil`

```
/**
 * Gestiona la lista de coches pertenecientes a un parque móvil
 * @author Mario
 * @version 1.0
 */
public class ParqueMóvil {

  // lista de coches en el parque
  private ArrayList<Coche> parque =
                          new ArrayList<Coche>();

  /**
   * Retorna el coche que ocupa la posición pos en la lista
   * @param pos posición
   * @return coche que ocupa la posición pos en la lista
   */
  public Coche coche(int pos) {
    return parque.get(pos);
  }
```



```
  /**
   * añade un coche al final de la lista
   * @param c coche a añadir
   */
  public void añade(Coche c) {
    parque.add(c);
  }

  /**
   * Elimina de la lista el coche indicado
   * @param c coche a eliminar
   * @return false si el coche no está en la lista y true si está
   */
  public boolean elimina(Coche c) {
    return parque.remove(c);
  }

  /**
   * Cambia el color del coche que ocupa la posición indicada
   * @param pos posición
   * @param color nuevo color para el coche
   */
  public void pintaCoche(int pos,
                         Coche.Color color) {
    parque.get(pos).pinta(color);
  }
```



```
  /**
   * Pinta todos los coches en el parque
   * @param color color con el que pintar los coches
   */
  public void pintaTodosLosCoches(
                                Coche.Color color) {
    for(Coche c: parque)
      c.pinta(color);
  }

  /**
   * Retorna un string con todos los elementos de la colección
   * @return representación en string de la colección
   */
  public String toString() {
    return parque.toString();
  }

}
```

Ejemplo `ArrayList`: Clase `Coche`

```
/**
 * Clase Coche
 * @author Mario
 * @version 1.0
 */
public class Coche {
  /**
   * Colores que puede tener un coche
   */
  public enum Color {ROJO,AZUL,VERDE}
  // atributos
  private String matrícula;
  private Color color;
  /**
   * Construye un coche con los parámetros indicados
   * @param matrícula matrícula del nuevo coche
   * @param color color del nuevo coche
   */
  public Coche(String matrícula, Color color) {
    this.matrícula=matrícula;
    this.color=color;
  }
```



```
  /**
   * Retorna la matrícula del coche
   * @return matrícula del coche
   */
  public String matrícula() {
    return matrícula;
  }

  /**
   * Retorna el color del coche
   * @return color del coche
   */
  public Color color() {
    return color;
  }

  /**
   * Pinta el coche del color indicado
   * @param color nuevo color del coche
   */
  public void pinta(Color c) {
    color=c;
  }
```



```
  /**
   * Redefine el método equals. Dos coches son
   * iguales si sus matrículas son iguales
   * @return true si los coches son iguales y false en caso contrario
   */
  public boolean equals(Object c) {
    return matrícula.equals(((Coche)c).matrícula);
  }

  /**
   * Redefine el método toString de Object
   * @return representación en string del colche
   */
  public String toString() {
    return "{\"" + matrícula + "\"," + color + "}";
  }

}
```

6.6 Documentación de módulos de programa

Para usar una clase, lo único que se necesita conocer de ella es la interfaz *pública*:

- ***atributos*: sus nombres y tipos, y descripción**
- ***métodos*: sus cabeceras y descripción de lo que hacen**

Se puede extraer esta información de manera automática, por medio de herramientas de documentación

En el *Bluej* elegir: `Tools->Project Documentation`

Si no hay conexión a Internet, deshabilitar la opción de "*buscar documentación del SDK*" (en *Preferencias*)

Documentación de módulos de programa (cont.)

La herramienta incorpora los comentarios de documentación

- **son los que empiezan por `/**`**
- **y que estén antes de una clase, atributo o método**
- **no poner una instrucción `import` entre el comentario de documentación y la clase**

Los comentarios de documentación pueden tener etiquetas que la herramienta interpreta de manera especial. Por ejemplo:

- **para clases:**

```
@author autor
@version versión
```

- **para métodos:**

```
@param nombreParam descripción del parámetro
@return valor retornado por el método
@throws nombreExcepción razón por la que se lanza
```


6.7 Bibliografía

- **Ken Arnold, James Gosling, David Holmes, "El lenguaje de programación Java", 3ª edición. Addison-Wesley, 2000.**
- **"The Java Tutorials" http://java.sun.com/docs/books/tutorial/java/index.html**
- **Francisco Gutiérrez, Francisco Durán, Ernesto Pimentel. "Programación Orientada a Objetos con Java". Paraninfo, 2007.**
- **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**