Material de Teoría: Programación (GIT)

Juan Gutiérrez (Departamento de Informática, Universitat de València)

Curso 23/24

Índice general

Parte I

Tema 1: Orientación a Objetos

Capítulo 1

Parte 1: Repaso de conceptos e introducción a Java

Tema 1 - Parte 1
Programación orientada a objetos

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos I

- Distinguir entre clase y objeto
- Distinguir entre método y mensaje
- Explicar qué es la encapsulación y cómo se consigue en la Programación Orientada a Objetos
- Distinguir entre estado y comportamiento de un objeto
- Explicar la función del constructor y usarlo correctamente
- Describir qué es la herencia y distinguir entre las relaciones «tiene un» (composición) y «es un» (herencia)
- Declarar clases usando la sintaxis del lenguaje Java
- Distinguir entre tipos primitivos y referencias en Java
- Explicar las operaciones que se pueden realizar sobre las referencias.
- Usar `this` para llamar a otros construtores de la misma clase.
- Declarar y usar `arrays` de tipos primitivos y de referencias en Java.
- Compilar y ejecutar aplicaciones en Java

Índice

¿Qué conceptos vamos a revisar?

- Objetos (también llamados instancias o agentes)
- Mensajes y métodos.
- Clases y objetos.
- Jerarquía de clases: herencia.

La analogía

Vamos a ilustrar esos conceptos de la orientación a objetos usando una analogía.

Supongamos que el coche de una persona llamada Juan tiene una avería mientras está circulando por la CV-380.

¿Cómo resuelve Juan este problema si no tiene ni idea de mecánica del automóvil?.

Objetos (instancias o agentes) (I)

Esta claro que Juan necesitará a alguien que pueda resolver el problema.

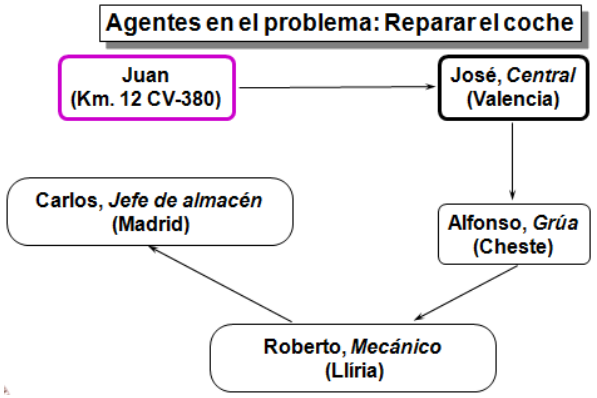
Juan llama a su seguro y habla con José. Le proporciona sus datos personales, los datos sobre el coche, su localización aproximada y una estimación de la gravedad de la avería.

Es decir, para resolver el problema hemos buscado un objeto (instancia o agente) que es el punto de partida para la resolución del mismo.

José por su parte tendrá que buscar a otros agentes (grúa, mecánico, etc).

Objetos (instancias o agentes) (II)

Un programa orientado a objetos está estructurado como una comunidad de objetos que interactúan. Cada objeto juega un papel: proporciona un servicio, o realiza una acción que es utilizada por otros miembros de la comunidad.



Mensajes y métodos (I)

La reacción en cadena (que sirvió para que el coche de Juan fuera reparado) comenzó con un mensaje (petición) a José.

José aplicará una serie de instrucciones (especificadas en un método) para atender la petición.

Juan no necesita saber la secuencia concreta de operaciones que va a seguir José para atender la petición.

Esta información es habitualmente ocultada ya que no es de interés para quien realiza la petición.

Mensajes y métodos (III)

En la programación orientada a objetos, la acción es iniciada enviando un **mensaje** a un objeto (instancia o agente).

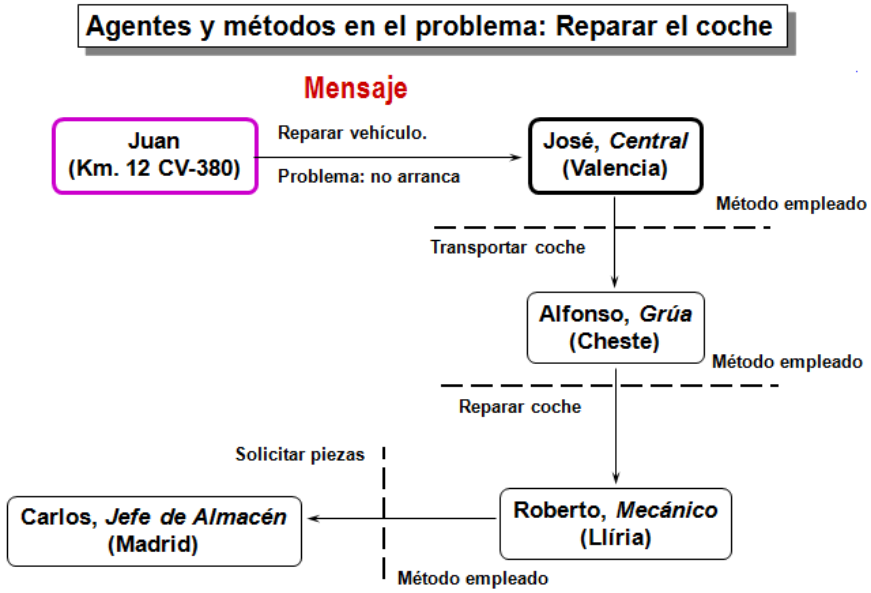
El mensaje codifica la petición de que se realice una acción y debe ir acompañado de la información adicional (**argumentos**) necesaria para llevar a cabo la petición.

El **destinatario** es el objeto al que es enviado el mensaje.

Si el destinatario acepta el mensaje, acepta la **responsabilidad** de llevar a cabo la acción.

Cuando el destinatario reciba el mensaje deberá realizar una serie de operaciones especificadas en un **método** y quien realiza la petición no necesita saber el conjunto de operaciones (la **implementación** del método).

Mensajes y métodos (IV)



Clases y objetos (I)

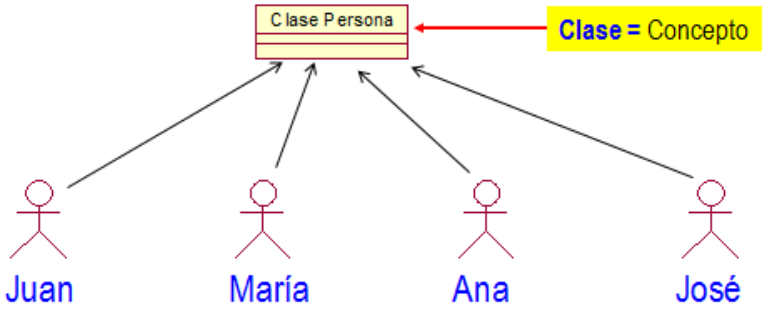
Aunque Juan haya tratado pocas veces con José, Juan tiene una ligera idea sobre lo que ocurrirá en la aseguradora.

Juan espera que José siendo una instancia de una categoría, se ajuste a un patrón general.

Podemos utilizar el término `Asegurador` para representar esta categoría (o clase) de todos los aseguradores.

Todos los objetos son **instancias** de una clase (o tipo).

Clases y objetos (II)

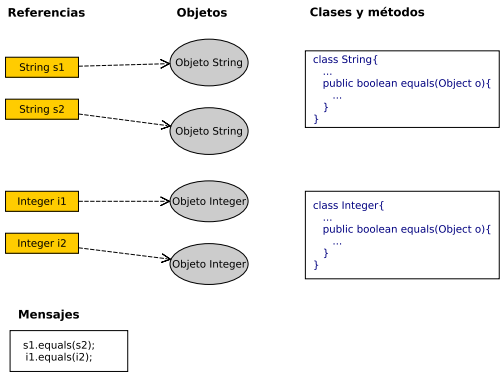


Objetos = Ejemplos concretos de la clase que responden al concepto.

Clases y objetos (III)

El método que se va a ejecutar como respuesta a un mensaje viene determinado por la clase del destinatario.

Todos los objetos de una clase utilizan el mismo método en respuesta a mensajes similares, es decir, lo que determina el comportamiento del objeto es la clase de la que es una instancia.



Jerarquía de clases. Herencia (I)

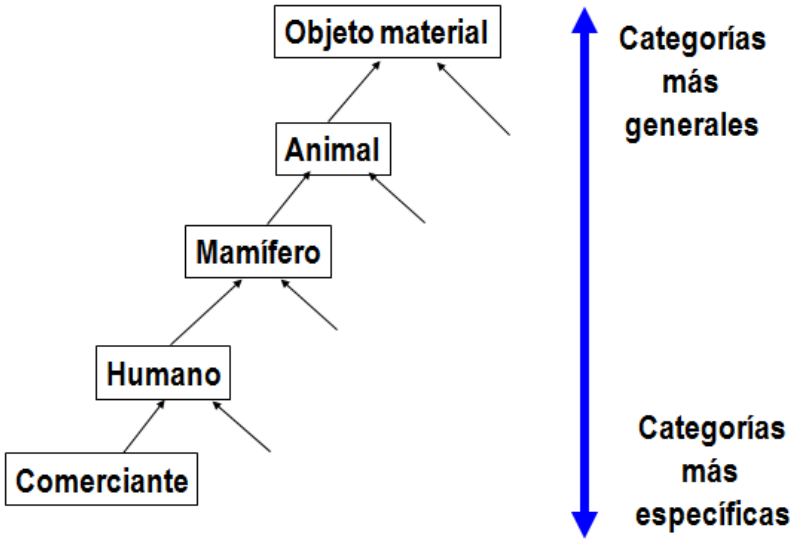
Juan tiene más información sobre José, porque además de que es `Asegurador` pertenece a la categoría de `Comerciante`.

Puesto que `Asegurador` es una forma especializada de `Comerciante`, cualquier conocimiento que Juan tenga sobre los comerciantes será cierto para los aseguradores y por tanto cierto para José.

Por ejemplo, podemos suponer que durante el proceso se emitirán facturas.

Esta característica es cierta también para otras sub-categorías de `Comerciante` como: `Fontanero`, `Cerrajero`, `Electricista`, etc..

Jerarquía de clases. Herencia (II)



Jerarquía de clases. Herencia (III)

Un `Comerciante` es también un `Humano` (por lo tanto José es bípedo, tiene una determinada forma y se le supone la capacidad de raciocinio).

Un `Humano` es un `Mamifero` (es vivíparo y es de sangre caliente).

Un `Mamifero` es un `Animal` (por lo tanto interactúa con el entorno para satisfacer sus necesidades básicas).

Un `Animal` es un `Objeto Material` (por lo que tiene una determinada masa y ocupa un volumen en el espacio).

El principio de que el conocimiento de una categoría general es también aplicable a una categoría más específica se conoce como **herencia**.

Jerarquía de clases. Herencia (IV)

La herencia establece una relación «**es-un**» entre la subclase y la superclase. Ejemplos:

```
Mamífero "es un" Animal

Humano "es un" Mamífero

Humano "es un" Objeto Material
```

Esta relación es diferente de la relación «**tiene-un**» (composición) que se da por ejemplo en:

```
Coche "tiene un" Motor

Teléfono "tiene un" Procesador
```


Jerarquía de clases. Herencia (V)

Las clases (categorías) pueden ser organizadas en una estructura jerárquica conocida como herencia.

Una clase hija (o subclase) hereda propiedades de la clase padre (o superclase).

Una clase que no puede tener instancias (como por ejemplo la clase `Mamífero`) se conoce como clase abstracta y se utiliza para agrupar propiedades y comportamiento común que será heredado por las subclases.

Resumiendo

La descomposición orientada a objetos supone un modo de dividir (simplificar) un problema, centrando la atención en:

- Los tipos de datos que modelan el problema
- Sus propiedades y comportamiento
- Interacción con otros objetos.
- La solución al problema se ve como un conjunto de agentes autónomos (los objetos) que colaboran entre sí para ejecutar una tarea de nivel superior.

La programación orientada a objetos aporta una forma particular de abordar el problema que condiciona la forma que adopta la solución.

Resumiendo

La programación orientada a objetos es (según una definición proporcionada por Booch):

Un método de implementación en el que los programas son organizados como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son miembros de jerarquías de clases unidas a través de una relación de herencia

https://en.wikiquote.org/wiki/Grady_Booch

Ejemplos

Problema: calcular el producto escalar de dos vectores $u, v \in \mathcal{R}^3$:

$$\vec{u} = (u_1, u_2, u_3), \quad \vec{v} = (v_1, v_2, v_3), \quad \vec{u}\vec{v} = \sum_{i=1}^{3} u_i v_i$$

Posible solución usando un lenguaje funcional:

```
(defun producto(a b)
  (+ (* (nth 0 a) (nth 0 b)) (* (nth 1 a) (nth 1 b)) (* (nth 2 a) (nth 2 b)))
  )
(message (number-to-string (producto (list 1 8 3) (list 3 -1 2))))
```


Ejemplos

Posible solución usando un lenguaje procedural (C):

```
#include <stdio.h>

struct Vector3{
  double v1;
  double v2;
  double v3;
};

double producto(struct Vector3 a, struct Vector3 b){
  double p = 0;
  p = p + a.v1*b.v1;
  p = p + a.v2*b.v2;
  p = p + a.v3*b.v3;
  return p;
}

int main(){
  struct Vector3 a = {1,2,3};
  struct Vector3 b = {3,-1,2};
  printf ("%f", producto(a,b));
}
```

Ejemplos I

Posible solución usando un lenguaje orientado a objetos (Java):

```
class Vector3{
   private double c1;
   private double c2;
   private double c3;

   public Vector3(double c1, double c2, double c3){
      this.c1=c1;
      this.c2=c2;
      this.c3=c3;
   }

   public double getC1(){
      return c1;
   }
   public double getC2(){
      return c2;
   }
   public double getC3(){
      return c3;
   }
```


Ejemplos II

```
   public double producto(Vector3 v){
      double p = 0;
      p = p + c1*v.getC1();
      p = p + c2*v.getC2();
      p = p + c3*v.getC3();
      return p;
   }
}

class Prueba{
   public static void main(String[] args){
      Vector3 a = new Vector3(1,2,3);
      Vector3 b = new Vector3(3,-1,2);
      System.out.println(a.producto(b));
   }
}
```


Índice

Declaración de clases

El lenguaje Java ofrece más de 3000 tipos de datos de propósito general que podemos usar para desarrollar aplicaciones.

Cuando abordamos un problema en un determinado dominio, tendremos que modelarlo determinando qué tipos de datos necesitamos.

Por ejemplo, si estamos haciendo una aplicación que permita crear y reproducir notas musicales, tendremos que crear un nuevo tipo de dato `NotaMusical` (que no lo ofrece el lenguaje).

Si estamos haciendo una aplicación que permita trabajar con estudiantes tendremos que crear un nuevo tipo de dato `Estudiante` (que tampoco lo ofrece el lenguaje).

Declaración de clases

En la programación orientada a objetos se crea un nuevo tipo de dato mediante una clase.

¿De qué consta un tipo de dato?

- Un tipo de dato tiene una representación interna
- Un tipo de dato ofrece una serie de operaciones desde las que se accede a la representación interna.

Declaración de clases

Hay dos puntos de vista:

Quien desarrolla el tipo de dato

Obviamente tiene que pensar en qué tipos de datos usará para definir la representación interna, definir cómo se van a crear instancias de este tipo y también definir qué operaciones se podrán realizar sobre el tipo.
Además deberá documentar esta información para que quien esté interesado en usar el tipo de dato sepa cómo usarlo.

Quien usa el tipo de dato (código cliente)

Como usuario no le interesa la representación interna. Le interesa cómo se construyen instancias de ese tipo, y qué operaciones puede realizar sobre el tipo (qué argumentos necesitan y qué devuelven). Por tanto, necesita el fichero o ficheros donde esté el tipo y la documentación (o API) del mismo.

Declaración de clases

Los tipos de datos ocultan la representación (los atributos), esta propiedad se denomina **encapsulación**.

¿Qué ventajas ofrece la encapsulación?

1. Solo se puede acceder a la representación desde los métodos que ofrece la clase. Si se permite el acceso a los atributos de un objeto desde código que está fuera de la clase se podría dejar al un objeto en un estado inconsistente.
2. Si se mantiene intacta la signatura de los métodos (la signatura de un método es el nombre del método, el número de sus parámetros y los tipos de las entradas y la salida) que ofrece el tipo entonces la representación se puede modificar sin que esto afecte al código cliente.

Ejemplo

Supongamos que declaramos la clase `Estudiante`:

```
class Estudiante{
   // Atributos (cada objeto tendrá valores diferentes)
   private String id;
   private String nombre;
   private char genero;
   // Constructor y métodos
   public boolean setGenero(char g){
      boolean cambiado=false;
      if ((g == 'm') || (g == 'f') || (g == 'b') || (g == 'x')){
         genero=g;
         cambiado=true;
      }
      return cambiado;
   }
}
```

Puesto que los atributos se han declarado como `private` solo se puede acceder a ellos a través de los métodos que ofrece la clase:

```
public class App{
   public static void main(String[] args){
      Estudiante e = new Estudiante("R18989","Cilantro",'m');
      e.setGenero('f');
      // Si pasamos un género no contemplado en la aplicación, no se cambiará:
      e.setGenero('z');
   }
}
```

Ejemplo

```
class Estudiante{
   // Atributos (cada objeto tendrá valores diferentes)
   private String id;
   private String nombre;
   public char genero; (1)
   // Constructor y métodos
   public boolean setGenero(char g){
      boolean cambiado=false;
      if ((g == 'm') || (g == 'f') || (g == 'b') || (g == 'x')){
         genero=g;
         cambiado=true;
      }
      return cambiado;
   }
}
```

Al establecer genero como public, en (1), hacemos que sea accesible desde código que esté en otra clase:

```
public class App{
   public static void main(String[] args){
      Estudiante e = new Estudiante("R18989","Cilantro",1992,7,20,'m');
      // Ahora podemos establecer cualquier char como género sin ningún
↪  control
      e.genero='i';
   }
}
```


Ejemplo

Consideremos el tipo de dato `Lista` (agrupación de elementos) que almacena `String` con las siguientes operaciones (métodos o acciones):

```
public boolean esVacia();
public int numElementos();
public void insertar(String dato);
public void borrar(int indice);
public void borrarTodo();
public String recuperar(int indice);
public boolean contiene(String dato);
```

¿Qué se puede usar para la representación (almacenar los datos)?

Ejemplo

Una posible representación (atributos) usando un array para almacenar los datos:

```
String[] datos;
int numDatos;
int capacidad;
```

```
public boolean esVacia();
public int numElementos();
public void insertar(String dato);
public void borrar(int indice);
public void borrarTodo();
public String recuperar(int indice);
public boolean contiene(String dato);
```


Ejemplo

Otra posible representación (atributos) usando una lista enlazada:

```
NodoLE cabeza;
NodoLE fin;
int numDatos;
```

Donde `NodoLE` es otro tipo de datos con los atributos:

```
String informacion;
NodoLE siguiente;
```

```
public boolean esVacia();
public int numElementos();
public void insertar(String dato);
public void borrar(int indice);
public void borrarTodo();
public String recuperar(int indice);
public boolean contiene(String dato);
```

Ejemplo

Nótese que en las dos transparencias anteriores no se ha cambiado la signatura de los métodos. Lo único que ha cambiado es la representación.

Resalto de nuevo la ventaja de la encapsulación (hacer inaccesible la representación al código cliente): Quien desarrolla la clase `Lista` puede cambiar la representación, y por consiguiente la implementación de los métodos, en futuras versiones y el código cliente, que fue desarrollado para la versión anterior, compilará con la nueva versión.

Declaración de clases en Java

Usamos la palabra reservada `class`:

```
class Lista{
   // Atributos
   private String[] datos;
   private int numDatos;
   private int capacidad;

   // Métodos
   public boolean esVacia(){...}
   public int numElementos(){...}
   public void insertar(String dato){...}
   public void borrar(int indice){...}
   public void borrarTodo(){...}
   public String recuperar(int indice){...}
   public boolean contiene(String dato){...}
}
```


Declaración de clases en Java

Algunas consideraciones sobre el código anterior:

- Los atributos se han declarado como `private` esto implica que no se puede acceder a ellos fuera del código de esta clase (encapsulación).
- Pero sí que se puede consultar o modificar los atributos desde el código de los métodos de la clase.
- Los métodos se han declarado como `public` esto implica que quien cree un objeto de este tipo podrá enviarle esos mensajes a través de una referencia.
- También se puede declarar métodos como `private`. Estos métodos sólo pueden ser llamados otros métodos de la clase.

Ejemplos de implementación de métodos para la representación con array

```
class Lista{
   //...
   public boolean esVacia(){
      return numDatos==0;  //consulta de un atributo
   }
   //...
   public String recuperar(int indice){
      String dato = null;   // Esta es una variable local al método solo
↪  existe en este ámbito
      if ((indice>=0) & (indice<numDatos)) // consulta del atributo indice
         dato = datos[indice];  // obtención de un dato del array
      return dato;
   }
   //...
}
```

Ejemplos de implementación de métodos para la representación con una lista enlazada

```
class Lista{
   //...
   public boolean esVacia(){
      return cabeza==null;  //consulta de un atributo
   }
   //...
   public String recuperar(int indice){
      String dato = null;   // Esta es una variable local al método solo
↪  existe en este ámbito
      if ((indice>=0) & (indice<numDatos)){ // consulta del atributo indice
         NodoLE aux = cabeza; // Comenzamos en el nodo cabeza
         for (int i=0;i<indice;i++)
            aux = aux.getSiguiente(); // Avanzamos posiciones
         dato = aux.getInfo();  // obtención del dato del nodo en el que nos
↪  hemos parado
      }
      return dato;
   }
   //...
}
```


Sobre la construcción de objetos

En las clases suele existir uno o varios métodos especiales que especifican cómo se construyen (inician) objetos de ese tipo.

Estos métodos reciben el nombre de **constructores**.

Cuando una clase no ofrece ningún constructor el compilador proporciona uno por defecto (sin argumentos).

El constructor cumple los siguientes requisitos: **se llama igual que la clase**, **no devuelve ningún valor** y **puede recibir argumentos**.

Cuando una clase declara una serie de constructores, es necesario llamar a alguno de ellos para crear instancias de ese tipo.

Sobre la construcción de objetos

Ejemplo de constructor para la clase `Estudiante`:

```
public class Estudiante{
   private String id;
   private String nombre;
   private char genero;

   public Estudiante(String identificador,String n, char g){
      id = identificador;
      nombre = n;
      genero = g;
   }
}
```

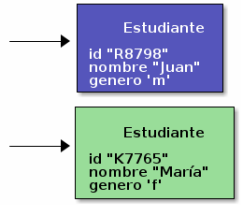
El propósito de los constructores que ofrece una clase es permitir iniciar objetos asignando valores a los atributos.

Sobre la construcción de objetos

Para crear un objeto se usa el operador `new` y a continuación se usa alguno de los constructores que ofrece la clase:

```
new Estudiante("R8798","Juan",'m');   1
new Estudiante("K7765","María",'f');  2
```

Objetos creados en el *heap* (memoria dinámica):



Sobre la construcción de objetos

Ejemplo de constructores para la clase `Lista`:

```
class Lista{
   //...
   public Lista(){
      capacidad = 100;
      datos = // creación del array para almacenar cap elementos
      numDatos = 0;
   }

   public Lista(int cap){
      capacidad = cap;
      datos =  // creación del array para almacenar cap elementos
      numDatos = 0;
   }
   //...
}
```

El propósito del constructor es iniciar los atributos a valores adecuados para que cuando se le envíen mensajes al objeto no se obtengan errores en tiempo de ejecución.

Tipos primitivos y tipo referencias

En el lenguaje de programación Java las variables deben ser declaradas, especificando su tipo, antes de se puedan usar.

En Java existen dos grupos de tipos: tipos primitivos y referencias.

Tipos primitivos:

Nombre	bits	Rango
byte	8 bits	-128 a 127
short	16 bits	-32768 a 32767
int	32 bits	-2^31 a 2^31-1
long	64 bits	-2^61 a 2^61-1
float	32 bits IEEE 754	
double	64 bits IEEE 754	
boolean	?	No especificado
char	16 bits	0 a 65635

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html

Tipos primitivos y tipo referencias I

```
// Ejemplo de uso de variables de tipo primitivo:
class Utilidades{
   public static void main(String[] args){
      int a=8;
      int b=19;
      System.out.println(a*b);
      System.out.println(a+b);

      char c ='a';
      System.out.println(c);

      double v = 1.8888;
      System.out.println(v/a);

      // La siguiente sentencia provoca un error de compilación
      // Nos informa de que v*a es un double y estamos asignando
      // a un entero por lo que tenemos una pérdida de precisión
      int d = v*a;
      System.out.println(d);

      // Para arreglarlo, hay que hacer explícita
      // la conversión de tipos
```


Tipos primitivos y tipo referencias II

```
      int d = (int)(v*a);
      System.out.println(d);
   }
}
```

Tipos primitivos y referencias

Una **referencia** es un valor que se refiere (permite acceder) a un objeto.

Al crear un objeto, usando alguno de sus constructores, se devuelve una referencia que podemos asignar a una variable. La variable almacena una referencia al objeto.

Las únicas operaciones permitidas sobre el tipo referencia son:

- Examinar o modificar el valor referencia (operadores $=$, $==$, $!=$)
- Consultar o modificar el estado (los valores de sus atributos) del objeto referenciado y enviar mensajes (operador .).

Ejemplo del tipo Lista

Por ejemplo, en el caso del tipo `Lista`:

```
// Declaración de una referencia del tipo Lista
// (puede referenciar objetos del tipo Lista)

Lista lista;

// Lo primero es el tipo de la referencia y
//lo segundo es el identificador de la referencia (variable)
```

En esa línea no se ha creado ningún objeto del tipo `Lista`, simplemente se ha declarado una referencia del tipo `Lista`.

Para asignar el objeto hay que usar el operador **new** con alguno de los constructores declarados en la clase.

```
// Creación de un objeto del tipo Lista y asignación a la referencia

lista = new Lista();
```


Ejemplo del tipo Lista

```
// Declaración de otra referencia del tipo Lista
Lista lista2;

// Asignación: ahora las dos referencias referencian el mismo objeto;
lista2 = lista;

// Comparación de referencias
boolean b = (lista2 == lista);

// Uso del operador . para enviar un mensaje al objeto a través de la
↪  referencia
boolean v = lista.esVacia();
```


Índice

La referencia this

La referencia `this` se puede usar en los métodos de una clase, y es una referencia, del tipo de la clase, al objeto que recibe el mensaje.

`this` es una referencia constante, no puede asignarse a otro objeto.

Se puede usar opcionalmente para acceder a los atributos y para enviar otros mensajes al objeto en la ejecución de un método.

También se usa para eliminar la ambigüedad cuando un argumento (de un método o de un constructor) se llama igual que un atributo.

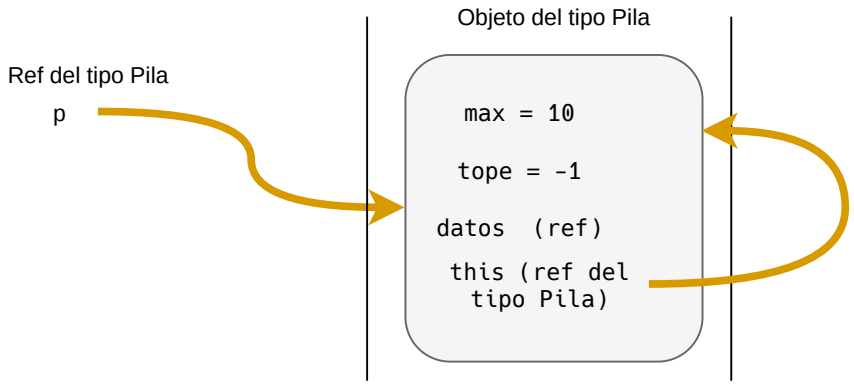
La referencia this

Supongamos que tenemos una clase `Pila` para almacenar `String`:

```
class Pila{
   private String[] datos;
   private int max;
   private int tope;

   public apilar(String dato){}
   public void desapilar(){}
   public String cima(){}
   public boolean esVacia(){}
}
```


La referencia this

```
Pila p = new Pila();
```



El valor a `this` es asignado al crear el objeto.

La referencia this (uso para eliminar ambigüedad)

```
class Pila{
   private String[] datos;
   private int max;
   private int tope;

   Pila(int max){
      // No podemos poner:
      // max = max;
      // Problema: a quien nombra cada max
      //    al argumento del método
      //    o al atributo de la clase
      this.max = max;  // usando this queda claro
   }
   //...
}
```

Uso de this para llamar a otro constructor

`this` también se puede utilizar para llamar desde un constructor a otro constructor de la misma clase.

```
public class Lista{
   private String[] datos;
   private int numDatos;
   private int capacidad;

   public Lista(int cap){
      capacidad = cap;
      datos = new String[capacidad];
      numDatos = 0;
   }

   public Lista(){
      // En este caso fijamos que la lista tiene como máximo 100 elementos
      // En lugar de repetir el código del otro constructor
      // lo llamamos pasando el argumento 100
      this(100);
   }

}
```

El objetivo es reutilizar código.

Índice

Arrays

Un array permite declarar `N` variables de un determinado tipo en una sola sentencia, y permite acceder a las variables mediante un índice entero.

Por ejemplo, si queremos almacenar las muestras de un segundo de canción muestreadas a 44.1 kHz necesitamos 44100 variables del tipo `double`. Obviamente, sería inmanejable declarar 44100 variables del siguiente modo:

```
double dato1;
double dato2;
//...
double dato44100;
```

Para ello podemos usar un array:

```
double[] muestras = new double[44100];
muestras[0] = 0.8765;
System.out.println(muestras[0]);
```


Todos los arrays son objetos

Los arrays son objetos y por tanto también se usan referencias con ellos y se crean con el operador `new`.

Vamos a distinguir dos tipos de arrays:

- Arrays de tipos primitivos.
- Arrays de referencias.

Arrays de tipos primitivos

Declaración de una referencia a un **array de tipos primitivos**:

```
tipo[] ref;
```

donde `tipo` es alguno de los tipos primivitos (`byte`, `char`, `boolean`, `short`, `int`, `long`, `float` o `double`).

Para asignarle un objeto usamos el operador `new`:

```
int tam = ...;
ref = new tipo[tam];
```

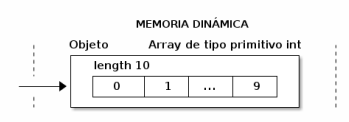
a partir de ese momento podemos acceder al elemento *i* del siguiente modo:

```
// Ejemplo de asignación
ref[i] = valor;

// Ejemplo de consulta del valor
if (ref[i] == valor){
   //...
}
```

Es decir, `ref[i]` es, a todos los efectos, una variable del tipo primitivo correspondiente.

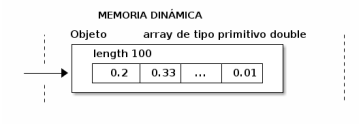
Arrays de tipos primitivos: ejemplos

```
int[] valores; // valores es null (no se le ha asignado ningún objeto)

valores = new int[10]; // asignación de un objeto creado con el operador new

for (int i=0; i<valores.length; i++)
   // valores[i] es de tipo int
   valores[i] = i;
```



Arrays de tipos primitivos: ejemplos

```
double[] notas = new double[100]; // Declaración y asignación de un objeto

for (int i=0; i<notas.length; i++)
   // notas[i] es del tipo double
   notas[i] = Math.random();
```



Arrays de referencias

Declaración de una referencia a un **array de referencias**:

```
Tipo[] ref;
```

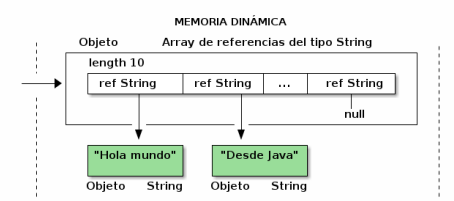
Para asignarle un objeto usamos el operador `new`:

```
int tam = ...;
ref = new Tipo[tam];
```

La diferencia respecto a los tipos primitivos es que ahora `ref[i]` es una referencia y por tanto hay que asignarle un objeto.

Arrays de referencias: ejemplo

```
...
// Declaración y asignación
String[] cadenas = new String[10];

System.out.println(cadenas.length);

//cadenas[0] es una referencia del tipo String
cadenas[0] = new String("Hola mundo");
// cadenas[1] es una referencia del tipo String
cadenas[1] = " desde Java";

// Podemos enviar un mensaje al objeto que referencia cadenas[0]
System.out.println(cadenas[0].compareTo(cadenas[1]);
```



Índice

Compilación

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Al instalar el SDK instalamos:

- Los tipos que define el lenguaje (en ficheros jar que contienen los ficheros .class)
- Una serie de herramientas que permiten (entre otras cosas) compilar y ejecutar aplicaciones (el Java Runtime Environment o JRE).

La herramienta que permite compilar se llama `javac`.

Si tenemos varios ficheros fuente en un directorio podemos compilarlos del siguiente modo:

```
javac *.java
```


Compilación

Al ejecutar `javac` se compila el código java y se genera un fichero `.class` por cada clase.

El código generado por javac se conoce como *bytecode*.

Es un código independiente del sistema operativo (a diferencia de lo que sucede con C/C++).

La función main

Para ejecutar una aplicación compuesta por una serie de clases, una de ellas debe implementar la función main declarada **exactamente** como figura a continuación:

```
class Aplicacion{
   //...
   public static void main(String[] args){
      //...
   }
}
```

Esta función se declara:

- `public`: es visible fuera de la clase
- `static`: los métodos y atributos declarados como `static` no necesitan de una instancia, están asociados a la clase.
- `void`: no devuelve nada
- Recibe una referencia a un array de referencias del tipo `String`.

Ejecución de la aplicación

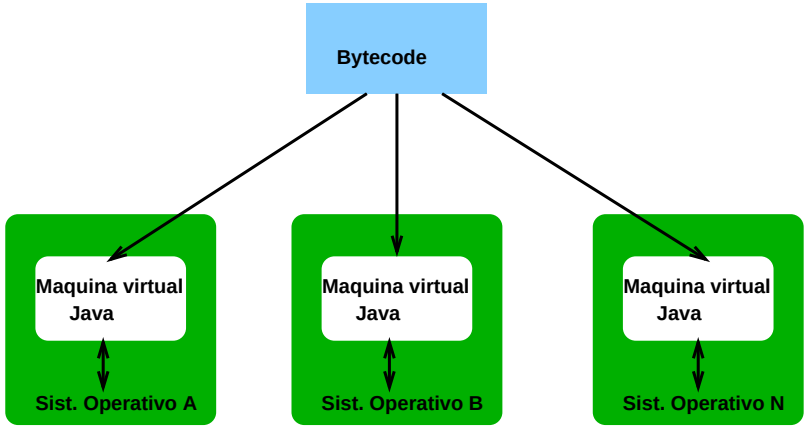
La ejecución se realiza usando la orden java a la que se pasa como primer argumento el nombre de la clase que contiene el método `main`:

```
java Aplicacion
```

Esta orden lanza una máquina virtual que cargará las clases necesarias y ejecutará el bytecode del método `main`.

Esta máquina virtual sí que es dependiente del sistema operativo.

Ejecución de la aplicación



Índice

Organización de las clases en paquetes

Las clases se pueden organizar en paquetes.

Para indicar que una clase pertecene a un paquete se usa la palabra reservada `package` seguida del nombre del paquete (al comienzo del fichero `.java`):

```
package es.uv.inf.coding;
// Declaración de la clase o clases
```

Desde el punto de vista físico, la clase debe estar en el siguiente directorio: `es/uv/inf/coding` (que es el nombre del paquete).

Y habitualmente se empaquetan en ficheros jar (fichero zip que contiene varias clases compiladas y ficheros con información adicional).

Organización de las clases en paquetes

Las clases **declaradas como públicas** serán visibles fuera del paquete (podemos declarar referencias o crear objetos desde código que no esté en el paquete).

```
package es.uv.inf.coding;
public class Video{
    //...
}
```

Las clases que **no estén declaradas como públicas** solo son visibles dentro del código de otras clases del paquete (esto se conoce como *package visibility*).

```
package es.uv.inf.coding;
class MediaSource{
    //...
}
// Desde la clase Video podemos usar MediaSource ya que
// está en el mismo paquete
```

Como vemos es otra forma de encapsulación (podemos ocultar clases).

Organización de las clases en paquetes

Supongamos que en nuestra aplicación queremos usar `Video`.

Para ello tenemos que usar la palabra reservada `import`

```
import es.uv.inf.coding.Video;

class VideoPlayer{
   public static void main(String[] args){
      Video v = new Video(...);
      // Pero no podríamos usar MediaSource ya que no es pública
   }
}
```


Organización de las clases en paquetes y módulos

En Java 9 han introducido la idea de módulo: un módulo es un conjunto de paquetes donde podemos controlar qué paquetes son visibles fuera del módulo (otra forma de encapsulación pero a nivel de paquete).

Supongamos que hacemos un módulo con 3 paquetes:

- `es.uv.inf.coding`
- `es.uv.inf.mediainfo`
- `es.uv.inf.headers`

y queremos que solo sea visible fuera del módulo el paquete `es.uv.inf.coding`. En este caso, incluiríamos un fichero `module-info.java` con la siguiente información:

```
module es.uv.inf.videocoding {
   exports es.uv.inf.coding;
}
```

Organización de las clases en paquetes y módulos

Si hacemos un módulo que quiere usar las clases del paquete `es.uv.inf.coding` exportado por el módulo `es.uv.inf.videocodig` entonces tendremos que incluir en el fichero `modules-info.java`:

```
module es.uv.app{
   // Nombre del módulo
   requires es.uv.inf.videocoding;
   // Si exporta las clases de algún paquete declarado en este módulo
   exports ...;
}
```

Al hacer esto, en el código fuente de las clases de este módulo podemos añadir:

```
// Importamos una clase del paquete exportado
// por el módulo que hemos indicado
import es.uv.inf.conding.Video;
//...
```

Capítulo 2

Parte 2: Clases abstractas e interfaces

Tema 1 - Parte 2
Programación orientada a objetos

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Crear y usar jerarquías de clases en Java
2. Determinar qué código se va a ejecutar cuando se envía un mensaje a un objeto cuando hay una jerarquía de clases
3. Redefinir (ocultar o sobreescribir) en una subclase el comportamiento heredado de la superclase
4. Describir y usar el polimorfismo en aplicaciones.
5. Usar el *upcasting* y *downcasting* y determinar la validez del código que lo use.
6. Enumerar, identificar y usar métodos de la clase `Object`.
7. Usar `super` para llamar a métodos y constructores de la superclase.
8. Declarar y usar atributos y métodos `protected`.
9. Describir, declarar y usar clases y métodos abstractos en Java.
10. Describir, declarar y usar clases y métodos finales en Java.
11. Declarar interfaces y clases que las implementen.
12. Usar interfaces para desarrollar algoritmos genéricos.

Índice

Índice

Jerarquías de clases: herencia

- **Herencia**: organización jerárquica del conocimiento de modo que lo definido en niveles superiores es aplicable a lo definido en niveles inferiores.
- Desde el punto de vista de los lenguajes de programación esto significa que lo definido en una superclase será heredado por las subclases y por lo tanto no es necesario volver a definirlo.
- Puede haber excepciones a esta norma general: una subclase puede modificar (redefinir) el comportamiento especificado en la superclase.
- Además, las subclases pueden añadir comportamiento específico (añadir atributos y nuevos métodos).
- Java no permite la herencia múltiple (una clase sólo puede extender a otra clase).

Sintaxis Java

Usamos la palabra reservada `extends` para indicar que una clase extiende a otra:

```
class Vehiculo { // Clase base o superclase
   //Definicion de atributos y métodos
   //que posean todos los vehículos
}

class VehículoConMotor extends Vehiculo {  // Subclase
   //Los elementos declarados en Vehiculo se heredan

   //Atributos adicionales declarados en esta clase

   //Constructor(es)

   //Metodos de la clase Vehiculo redefinidos

   //Metodos adicionales que son exclusivos de VehículoConMotor

   //Metodos privados adicionales
}
```

Sintaxis Java

- Los elementos privados de una clase solo son accesibles desde los métodos de esa clase (desde el código de la subclase no se puede acceder a atributos privados de la superclase).
  - Por ejemplo, los elementos definidos como `private` de `Vehiculo` no se pueden usar en los métodos de `VehiculoConMotor`.
  - El acceso a los atributos `private` de `Vehiculo` se debe realizar mediante los métodos públicos que ofrece la clase.
- Los atributos declarados como `protected` en una clase son visibles en las subclases, pero no son visibles para otras clases que no estén en la jerarquía.

Sintaxis Java

```
class Circulo {
   protected int x_centro;
   protected int y_centro;
   protected int radio;

   public Circulo () {...}
   public void ponerCentro (int x, int y){...}
   public void ponerRadio (int r) {...}
   public double longitud () {...}
   public double area () {...}
   public void dibujar () {...}
}

class CirculoColor extends Circulo {
   private int color;

   public CirculoColor () {...}

   // Nuevos métodos definidos en esta clase
   public void ponerColor (int c) {...}
   public int obtenerColor () {...}

   // Redefine el método dibujar ya que se dibuja de forma diferente
   public void dibujar () {...}
}
```


Sintaxis Java

```
class Circulo {
   // protected ...

   public Circulo () {
      x_centro = 0;
      y_centro = 0;
      radio = 10;
      System.out.println("Constructor de Circulo");
   }
   public void ponerCentro (int x, int y){
      x_centro = x;
      y_centro = y;
   }
   public void ponerRadio (int r) {
      radio = r;
   }
   public double longitud () {
      return ( 2*Math.PI*radio );
   }
   public double area () {
      return ( Math.PI*radio*radio );
   }
   public void dibujar () {....}
}
```

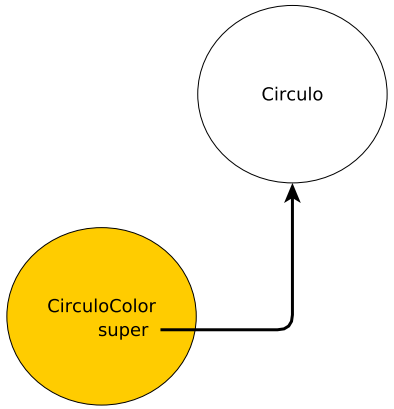

Sintaxis Java

```
class CirculoColor extends Circulo {
   private int color;

   public CirculoColor () {
      color = 1;
   }

   public void ponerColor (int c) {
      color = c;
   }

   public int obtenerColor () {
      return (color);
   }

   public void dibujar () {
      // Se redefine el método dibujar de la superclase
      // ya que se debe rellenar con el color
      // correspondiente
   }
}
```

Creación de objetos en una jerarquía

Dada la jerarquía anterior, el compilador añade el código para que al crear un objeto del tipo `CirculoColor` la máquina virtual cree automáticamente un objeto del tipo `Circulo` que se vincula con `CirculoColor` mediante un atributo del tipo `Circulo` y con nombre `super`.

```
CirculoColor c = new CirculoColor();
```



Esta es una desventaja de la herencia: creación de más objetos en memoria.

Índice

Introducción

Hasta ahora las sentencias de creación de objetos no representaban ningún problema porque el tipo del objeto y el tipo de la referencia coincidían.

```
Circulo c = new Circulo();
```

Tampoco lo presentaban las sentencias de asignación de referencias porque también coincidían los tipos de las referencias.

```
Circulo c = new Circulo();
// Ejemplo de asignación de referencias
Circulo c1 = c;
```

Sin embargo, cuanto tenemos una jerarquía de clases puede ser que no coincidan los tipos que tenemos a la izquierda de la igualdad con los tipos que tenemos a la derecha.

Vamos a ver qué tipo de sentencias son válidas cuando tenemos estas situaciones.

Notación y definiciones

Supongamos que tenemos un conjunto de clases $\{C_i\}_{i=1,\ldots,N}$ que definen una jerarquía.

Denotemos como $E(C_i, C_j)$ la relación: C_i extiende a C_j.

Denotemos como $S(C_k, C_m)$ la relación: C_k es una superclase de C_m.

Está claro que si $S(C_k, C_m)$ entonces existe una serie de clases $C_t, C_r, \ldots, C_w$ tales que se cumple:

$E(C_t, C_k)$, $E(C_r, C_t)$, $\ldots$, $E(C_w, C_m)$

Notación y definiciones

Sea $O(C_k)$ un objeto del tipo C_k creado mediante `new`.

Sea $R(C_k)$ una referencia del tipo C_k.

Recordemos que la sentencia `Tipo ref;` declara una referencia y no crea ningún objeto.

Usando esta notación podemos afirmar que:

- $R(C_k) = O(C_k)$ es correcto
- $R(C_j) = O(C_k)$ es correcta siembre que $S(C_j, C_k)$.
- $R(C_j) = R(C_k)$ es correcta siempre que $S(C_j, C_k)$ (asignación de referencias).
- Si $S(C_k, C_m)$ entonces $R(C_m) = R(C_k)$ es incorrecta.
- Si $S(C_k, C_m)$ entonces $R(C_m) = (C_m)R(C_k)$ puede ser correcta en tiempo de compilación (aunque puede que sea incorrecta en tiempo de ejecución).

Notación y definiciones

En la expresión: $R(C_j) = R(C_k)$ (con $S(C_j, C_k)$) se realiza un *upcasting* de la referencia (conversión de tipo hacia arriba) automático.

En la expresión: $R(C_m) = (C_m)R(C_k)$ (con $S(C_k, C_m)$) se realiza un *downcasting* de la referencia (conversión de tipo hacia abajo) y ha de ser explícito.

Nótese que en ambos casos se realiza una conversión de tipo de la referencia y no del objeto (solo cambia a través de qué referencia vemos el objeto).

Ejemplos

```
Circulo c;

CirculoColor cc;

cc = new CirculoColor();

// Correcto: Circulo es una superclase de CirculoColor
// o visto de otro modo: un CirculoColor es un Circulo
c = cc;

Circulo c = new CirculoColor();
```


Ejemplos

```
Circulo c;

CirculoColor cc;

c = new Circulo();

// Incorrecto (error de compilación)
cc = c;
// Si nos dejara hacer esta asignación podríamos enviar a través de
// cc el mensaje ponerColor(8) a un objeto que no tiene color.
```

Ligadura dinámica

Hay situaciones en las que hasta el momento de la ejecución no se sabrá qué método se va a ejecutar. Consideremos el siguiente código:

```
class Ejemplo{

   public static void pintaCirculos(Circulo[] c){
      for (int i=0; i<c.length; i++)
         c[i].dibujar(); // Qué método se ejecuta?
   }

   public static void main(String[] args){
      Circulo[] circulos = new Circulo[100];
      for (int i=0;i<100;i++){
         if (Math.random()<0.5)
            circulos[i] = new Circulo();
         else
            circulos[i] = new CirculoColor();
      }
      pintaCirculos(circulos);
   }
}
```

Cada ejecución de la sentencia `c[i].dibujar()` en el método `pintaCirculos(...)` puede provocar la ejecución de diferentes métodos.

Ligadura dinámica

El método `dibujar()` (de `Circulo` o de `CirculoColor`) no se conocerá hasta el momento de ejecutar esa sentencia.

Esta capacidad se denomina **polimorfismo** (una misma sentencia puede dar lugar a la ejecución de diferente código) y el mecanismo de determinación del método se denomina **ligadura dinámica** o *dynamic binding*

Las referencias determinan qué mensajes se pueden enviar al objeto que referencian, pero el método a ejecutar no depende del tipo de la referencia sino del tipo del objeto al que referencia.

Ligadura dinámica

Para determinar el método que se va a ejecutar cuando una clase pertenece a una jerarquía se procede del siguiente modo:

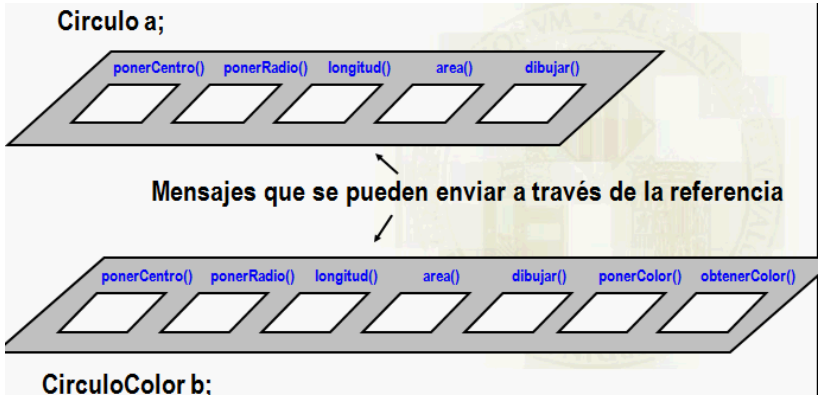
Se busca de forma ascendente (comenzando por la clase del objeto que recibe el método): si no se encuentra se busca en la superclase, y así ascendiendo hasta que se encuentre la primera clase que lo implementa.

Ejemplo

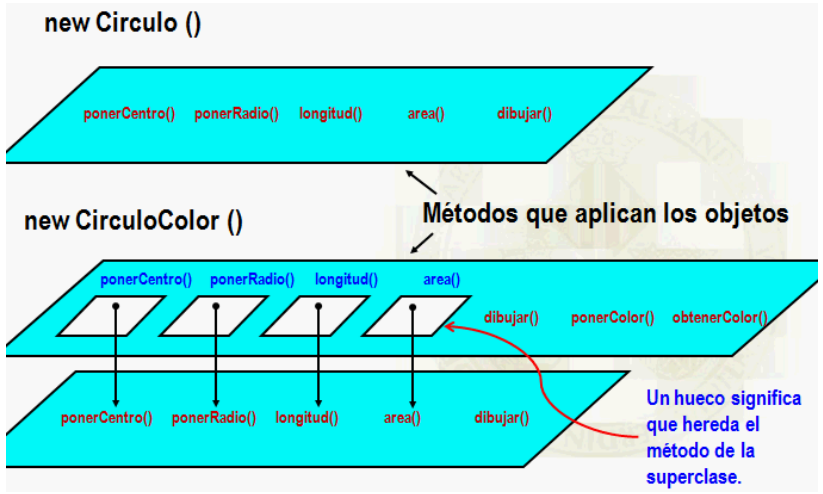
```
class Mi_Programa {

   public static void main (String[] args) {

      Circulo a, b, c;

      a = new Circulo();
      a.dibujar(); //1

      b = new CirculoColor();
      b.dibujar(); //2

      c = new Circulo();
      c.dibujar(); //3

      c = b;
      c.dibujar(); //4
   }
}
```

¿Qué método se ejecutará al ejecutar cada una de las sentencias comentadas con 1, 2, 3 y 4?

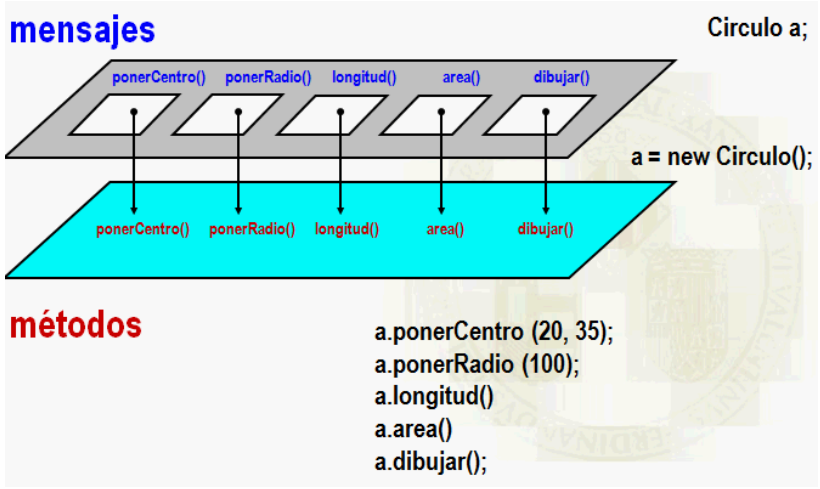
Una forma gráfica de ver las referencias y los objetos



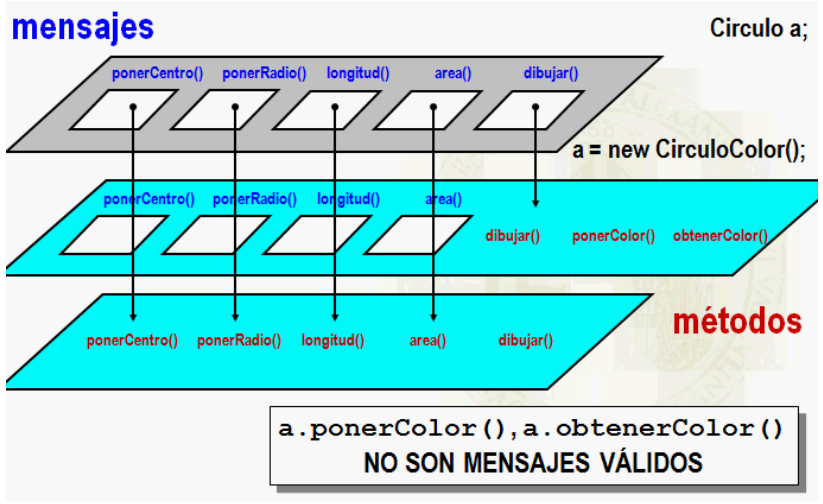
Una forma gráfica de ver las referencias y los objetos



Una forma gráfica de ver las referencias y los objetos



Una forma gráfica de ver las referencias y los objetos



Índice

Métodos declarados en la clase Object

En Java todas las clases derivan de la clase `Object`, aunque no se indique explícitamente.

`Object` es la clase raíz del árbol de jerarquías de Java. Esto implica que:

- Cualquier objeto se puede referenciar mediante una referencia del tipo `Object`.
- A cualquier objeto le podemos enviar un mensaje de un método declarado en la clase `Object`.

Vamos a ver dos métodos interesantes: `equals` y `toString` declarados en la clase `Object`.

Índice

toString

Finalidad típica: obtener un `String` con información de un objeto.

Para ello, la clase `Object` declara el método `toString()`:

```
public String toString ()
```

La implementación de este método en la clase `Object` devuelve una cadena con el tipo del objeto y un valor entero del código *hash* (función sobre los atributos del objeto que devuelve un entero) en hexadecimal.

Para poder obtener una representación en forma de cadena de nuestros objetos hace falta incluir en nuestra clase la redefinición correcta de `toString` (sobreescribir el método `toString()` en nuestra clase).

Ejemplo

```
class ProgramaLavado {

   private String nombre;
   private int num_fases;
   private int duracion;
   private int consumo_agua;
   private boolean centrifugado;
   private boolean prelavado;

   public ProgramaLavado (...)
   {
      //...
   }
   public void activar () {...}
   public int consumoAgua () {...}
   public String nombre () {...}
   public int duracion () {...}

   public String toString () {
      String cadena;
      cadena = "Programa: " + nombre + " , ";
      cadena += "Duración: " + duracion + " , ";
      cadena += "Consumo agua: " + consumoAgua;
      // Hay que devolver el String, no mostrarlo por salida estándar
      return ( cadena );
   }
}
```


Índice

equals

Está pensado para comprobar si dos objetos son iguales.

Prototipo de `equals` en la clase `Object`:

```
public boolean equals (Object o)
```

La implementación de `equals` en la clase `Object` solo comprueba si el objeto que recibe el mensaje y el argumento hacen referencia al mismo objeto.

Es decir, la implementación de `Object` comprueba igualdad de referencias, no de contenidos.

Para poder comparar bien nuestros objetos hace falta incluir en nuestra clase la redefinición del método `equals`.

Ejemplo

```
class Circulo {

   private int x_centro;
   private int y_centro;
   private int radio;

   public Circulo () {...}
   public void ponerCentro (int x, int y){...}
   public void ponerRadio (int r) {...}
   public double longitud () {...}
   public double area () {...}
   public void dibujar () {...}

   public boolean equals (Object o) {
      // Hay que hacer una conversión explícita de la referencia
      // del tipo Object (parámetro formal del método)
      // para obtener una referencia del tipo Circulo
      Circulo c = (Circulo) o;
      if ( x_centro = c.x_centro)
         return ( false );
      else
         if ( y_centro=c.y_centro)
            return ( false );
         else
            if ( radio != c.radio )
               return ( false );
            else return ( true );
   }
}
```

Índice

Clases abstractas

Clases de las que no tiene sentido crear objetos, pero sirven para agrupar características comunes de las clases que derivan de ellas.

Ejemplo: por la carretera no circulan Vehículos en abstracto, circulan Coches, Camiones, Bicicletas (tipos concretos de Vehículos).

Ejemplo La clase `Component` representa un componente en una interfaz gráfica de usuario. En una aplicación creamos componentes concretos: botones, campos de texto, áreas de texto, etc. (subclases de `Component`).

La finalidad de las clases abstractas es definir los aspectos comunes a todas las subclases.

Clases abstractas

Para definir una clase abstracta en Java usamos la palabra reservada `abstract` al declarar la clase:

```
abstract class Component{
   //...
}
```

```
abstract class Vehiculo {
   //...
}
```

Las clases abstractas pueden (y suelen) incluir métodos sin implementación, son los llamados métodos abstractos.

Métodos abstractos

Son métodos que a esta altura de la jerarquía no pueden tener implementación (no sabemos implementarlos) y que obligan a ser implementados necesariamente en alguna clase derivada.

Si una clase posee un método abstracto la clase se debe declarar como abstracta (si no se hace se obtiene un error de compilación).

Si una clase deriva de una clase abstracta y no implementa alguno de los métodos abstractos, entonces hay que declararla como abstracta.

Para poder crear objetos de una clase se deben haber implementado en la jerarquía todos los métodos abstractos heredados (da igual en qué nivel de la jerarquía se hayan implementado).

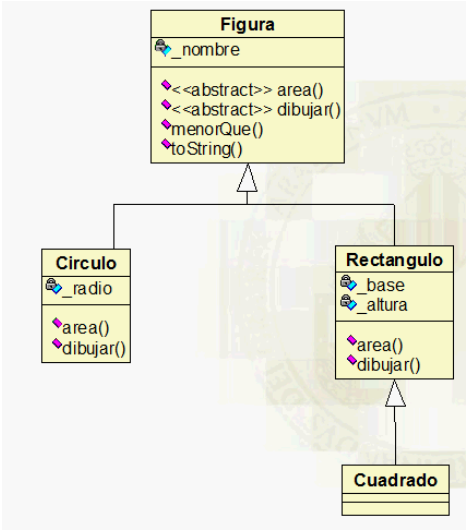
Métodos abstractos

¿Qué sentido tiene la definición de un método abstracto en una clase si no podemos implementarlo?

- Puede ser que sea un método que tiene sentido que esté a este nivel, pero no podemos implementarlo todavía. Al declararlo abstracto obligamos a que las subclases deban implementarlo. A través de una referencia del tipo de la clase donde está declarado podemos enviar **ese mensaje** a los objetos que estemos referenciando y que están por debajo en la jerarquía.
- Con la declaración de un método abstracto, $m_a(...)$, podemos implementar otros métodos de la clase abstracta que llamen en su implementación a $m_a(...)$.

Ejemplo: jerarquía

Especificamos que cualquier `Figura` debe tener una área y se puede dibujar. Sin embargo, a este nivel no podemos implementar estos métodos.



Ejemplo: clase Figura

```
abstract class Figura {
   private String nombre;

   abstract public double area ();

   abstract public void dibujar ();

   public Figura (String nombreFigura) {
      nombre = nombreFigura;
   }

   public boolean menorQue (Figura f) {
      return ( this.area() < f.area() );
   }

   public String toString () {
      return ( nombre + `` , '' + area() );
   }
}
```

- ¿Es correcto el código del método `menorQue(.)`?
- ¿Qué código se ejecuta al llamar a `area()`?

Ejemplo: clase Circulo

```
class Circulo extends Figura{
   private double radio;

   public Circulo (double r) {
      super ("Circulo");
      radio = r;
   }

   public Circulo () {
      this (10.0);
   }

   public double area () {
      return ( Math.PI * radio * radio );
   }

   public void dibujar () {
      System.out.println ("Soy un Circulo de radio: " + radio);
   }
}
```

Cuando una superclase declara un constructor con argumentos, hay que llamarlo explícitamente desde el constructor de la subclase mediante `super(argumentos)`.

Ejemplo: clase Rectangulo

```
class Rectangulo extends Figura{
   private double base;
   private double altura;

   public Rectangulo (double ancho, double alto) {
      super ("Rectangulo");
      base = ancho;
      altura = alto;
   }
   public Rectangulo () {
      this (10.0, 10.0);
   }

   public double area () {
      return ( base * altura );
   }

   public void dibujar () {
      System.out.println ("Soy un Rectangulo de: " + base + "*"
                          + altura);
   }
}
```


Ejemplo: clase Cuadrado

```
class Cuadrado extends Rectangulo{
   public Cuadrado (double lado) {
      super (lado, lado);
   }

   public Cuadrado () {
      super (10.0, 10.0);
   }
}
```


instanceof

En tiempo de ejecución se puede comprobar el tipo del objeto que estamos referenciando usando el operador `instanceof` (puede ser útil para hacer *downcast* de forma segura).

```
class Ejemplo{

   public static Figura creaFigura(){
      Figura f = null;
      if (Math.random()>0.5)
         f = new Circulo(30);
      else
         f = new Rectangulo(10,15);
      return f;
   }

   public static void main(String[] args){
      Figura f = creaFigura();
      if (f instanceof Circulo){
         Circulo c = (Circulo)f;
         // Hacer algo con el circulo
      }else
         if (f instanceof Rectangulo){
            Rectangulo r = (Rectangulo)f;
            // Hacer algo con el rectangulo
         }
   }
}
```


Índice

Métodos finales

En ocasiones interesa indicar que un determinado método no se puede redefinir, para ello el lenguaje ofrece la palabra reservada: `final`.

```
//...
final public boolean menorQue (Figura f) {
   return ( this.area() < f.area() );
}
//...
```

Las clases derivadas **NO** pueden redefinir `menorQue`. Si lo hacen obtendrán un error de compilación.

Ojo: redefinir implica mantener la signatura del método.

Los métodos finales tienen **ligadura estática** (compilación).

Clases finales

También se puede prohibir que una clase sea extendida declarándola como final.

```
final class Cuadrado extends Rectangulo{

   public Cuadrado (double lado) {
      super (lado, lado);
   }

   public Cuadrado () {
      super (10.0, 10.0);
   }
}
```

En este caso no sería posible definir una nueva clase que derive de `Cuadrado`.

Índice

Interfaces

Las interfaces son clases abstractas y hasta la versión 7 de Java no implementaban ninguno de los métodos.

A partir de la versión 8 de Java las interfaces pueden ofrecer una implementación por defecto de alguno de sus métodos.

Para definir una interfaz usamos la palabra reservada `interface` en lugar de `class`:

```
public interface NombreInterface{
   public Tipo nombreMetodo1(argumentos);
   //...
   public Tipo nombreMetodoN(argumentos);
}
```

Si una interfaz tiene atributos estos deben ser declarados como `public`, `static` (no asociadas a instancias) y `final` (son constantes).

Interfaces

A partir de la versión 8 de Java, una interfaz puede proporcionar una implementación por defecto de alguno de sus métodos.

Para ello se usa la palabra reservada `default`:

```
public interface Nombre{
   default Tipo nombreMetodo(argumentos){
      // Implementación
   }
   // Resto de métodos
}
```


Interfaces

Se dice que una clase **implementa** a una interfaz (en lugar de extender).

Si una clase implementa a una `interface` debe implementar todos los métodos (que no hayan sido implementados en la interfaz) o si no lo hace se debe declarar como abstracta (dejamos a las subclases que los implementen).

```
class A implements NombreInterface{
   // Atributos
   // Metodos de la clase

   // Implementación de los métodos de la interface
   public Tipo nombreMetodo1(argumentos){
      //...
   }
   //...
   public Tipo nombreMetodoN(argumentos){
      //...
   }
}
```


Interfaces

Una clase puede implementar a varias interfaces.

¿Qué ocurre si una clase implementa a dos interfaces que definen el mismo método y que ofrecen implementación por defecto?.

En ese caso es necesario implementar el método en la clase e indicar cuál de las dos implementaciones queremos usar:

```
interface I1{
   default String mensaje(){
      return "Hola desde I1";
   }
}
interface I2{
   default String mensaje(){
      return "Hola desde I2";
   }
}

class Prueba implements I1, I2{
   public String mensaje(){
      return I1.super.mensaje();
   }
}

class App{
   public static void main(String[] args){
      Prueba p = new Prueba();
      System.out.println(p.mensaje());
   }
}
```


Interfaces

Si no implementamos el método en la clase `Prueba` obtendremos un error en tiempo de compilación indicando que se hereda el mismo método implementado de las dos interfaces.

También se ha añadido, a partir de la versión 8, la posibilidad de que las interfaces puedan declarar métodos estáticos con implementación.

Herencia múltiple

La motivación inicial para la introducción de las interfaces era evitar problemas con la herencia múltiple.

Como ya hemos comentado Java no permite la herencia múltiple.

Pensemos en qué puede pasar con la herencia múltiple: una clase que extiende a dos clases en las que se define en ambas un método con la misma signatura.

¿Qué ocurre cuando a la subclase se le envíe ese mensaje? ¿Cuál de los dos códigos debe ejecutar?.

Con las interfaces no ocurre esto. Si una clase implementa a dos interfaces y en esas dos interfaces se define el mismo método no hay ningún conflicto puesto que **no está heredando código sino la responsabilidad de implementar ese método**.

Este argumento pierde fuerza cuando las interfaces pueden definir la implementación por defecto para sus métodos.

Índice

Generalización de un algoritmo

Las interfaces se pueden usar para exigir requisitos a la hora de hacer algoritmos genéricos (que puedan trabajar con cualquier tipo de objetos siempre que cumplan los requisitos exigidos en la `interface`).

Vamos a suponer que tenemos un algoritmo complejo que puede ser interesante para otros usuarios. Supongamos que ese algoritmo trabaja con un determinado tipo de dato.

Vamos a suponer (sin pérdida de generalidad) que este algoritmo super complejo que deseamos compartir con el resto de usuarios es el siguiente (en la realidad puede tratarse de algoritmos de ordenación).

```
class Utilidades {
   public static int maximo ( int a, int b ) {
      int val = a;
      if ( b > a )
         val = b;
      return val;
   }
}
```

Este algoritmo devuelve el mayor de dos valores.

Generalización de un algoritmo

¿Sería posible una generalización directa de ese código para que cumpla su propósito (obtener el mayor de dos elementos) del siguiente modo?

```
class Utilidades {
   public static Object maximo ( Object a, Object b ) {
      Object val = a;
      if ( b > a )
         val = b;
      return val;
   }
}
```

En este código hacemos uso del hecho de que cualquier objeto es de tipo `Object`.

La respuesta es **NO**.

Ese código no es correcto ya que sobre las referencias no está definida la operación $>$.

Generalización de un algoritmo

Para hacer nuestro super algoritmo general tendremos que comenzar por buscar cuál es la operación clave que se realiza sobre los datos:

```
class Utilidades {
   public static int maximo ( int a, int b ) {
      int val = a;
      if ( b > a )
         val = b;
      return val;
   }
}
```

La operación clave es el $>$ o **«mayor que»**.

Generalización de un algoritmo

El segundo paso consiste en definir esta operación en una *interface*:

```
public interface Compara{
   public boolean mayorQue(Compara c);
}
```

El tercer paso consiste en exigir que a nuestro super algoritmo se le pasen dos objetos del tipo `Compara`:

```
class Utilidades {
   public static Compara maximo ( Compara a, Compara b ) {
      Compara val = a;
      if ( b.mayorQue(a) )
         val = b;
      return val;
   }
}
```

Ya podemos ofrecer el super algoritmo junto con la *interface*.

Uso del algoritmo

Ahora adoptamos el rol de un usuario que desea **usar** el algoritmo (recordemos que se trata de un algoritmo útil y que se supone que ha sido probado exhaustivamente) con sus tipos de datos.

Supongamos que tenemos la siguiente jerarquía:

```
abstract class FiguraCerrada{
   abstract public void pintar();
   abstract public double area();
}

class Rectangulo extends FiguraCerrada{
   // Atributos
   // Métodos

   public void pintar(){...}
   public double area(){...}
}

class Circulo extends FiguraCerrada{
   // Atributos
   // Métodos

   public void pintar(){...}
   public double area(){...}
}
```


Uso del algoritmo

¿Qué debo hacer si quiero usar ese algoritmo para determinar qué figura es mayor en función de su área?.

Hacer que en mi jerarquía se implemente la interfaz `Compara`.

¿En esta jerarquía dónde se podría implementar esa interfaz?

En la clase `FiguraCerrada`:

```
abstract class FiguraCerrada implements Compara{
   abstract public void pintar();
   abstract public double area();
   public boolean mayorQue(Compara o){
      FiguraCerrada f = (FiguraCerrada)o;
      return (area()>f.area());
   }
}
```

Uso del algoritmo

Con esta modificación, el siguiente código es válido:

```
//...
Rectangulo r = new Rectangulo(10,20);
Circulo c = new Circulo(20);
FiguraCerrada f = (FiguraCerrada)(Utilidades.maximo(r,c));
// Ahora f es una referencia a la figura con mas area
//...
```


Resumiendo

Quien desarrolla el algoritmo genérico

- Determina las operaciones clave que deben poseer los datos para que el algoritmo funcione.
- Recoge estas operaciones en una interfaz.
- Escribe el algoritmo en función de la interfaz.
- Ofrece el algoritmo y la interfaz.

Quien quiere usar el algoritmo genérico

- Hace que sus datos implementen la interfaz.
- Usa el algoritmo.

Capítulo 3

Parte 3: Collections framework, expresiones lambda y streams

Tema 1 - Parte 3
Programación orientada a objetos

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Declarar y usar tipos parametrizados.
2. Imponer requisitos sobre el parámetro del tipo: `<T extends Interface>`.
3. Usar contenedores parametrizados (del *Collections Framework*) que ofrece Java
4. Usar los mecanismos que ofrece Java para facilitar el recorrido de las colecciones.
5. Expresiones lambda para el trabajo con interfaces funcionales.
6. Usar streams junto con expresiones lambda para procesar datos de colecciones.

Índice

Motivación

Crear contenedores de datos que puedan contener cualquier tipo de dato y que permitan detectar errores en su uso en tiempo de compilación.

En la parte 2 del tema 1 hemos visto que es posible referenciar cualquier objeto con una referencia del tipo `Object`.

Por tanto un modo trivial de hacer un contenedor sería haciendo uso de esta referencia. Por ejemplo:

```
public class Contenedor{
   private Object el;

   public setElement(Object o){
      el = o;
   }
   public Object getElement(){
      return el;
   }
}
```


Motivación

El código anterior es correcto y se puede usar del siguiente modo:

```
Contenedor c = new Contenedor();
String s = "Hola mundo";

c.setElement(s);

Object o = c.getElement();
```


Motivación

Pero:

- Es **incómodo de usar** ya que al recuperarlo tengo una referencia del tipo `Object`, por lo que será necesario realizar un *downcasting* explícito para acceder a la funcionalidad del objeto almacenado:

  ```
  System.out.println(((String)c.getElement()).indexOf('a'));
  ```

- Puede dar lugar a **errores en tiempo de ejecución** al insertar objetos de un tipo y al recuperarlos intentar referenciarlos con referencias no compatibles:

  ```
  Contenedor c = new Contenedor();
  String s = "Hola mundo";
  c.setElement(s);
  Integer entero = (Integer)c.getElement();
  ```

  El error se produce en ejecución ya que estamos intentando referenciar un objeto de tipo String con una referencia del tipo Integer.

¿Ofrece algún mecanismo el lenguaje para solucionar este inconveniente y el probable error?

Declaración de tipos parametrizados

Solución: Tipos parametrizados (también llamados plantillas o *templates*)

Se declara un tipo de dato que tiene un parámetro libre. Ese parámetro representa a un tipo E que por ahora es desconocido.

```
public class Contenedor<E>{
   private E el;

   public void setElement(E e){
      el = e;
   }
   public E getElement(){
      return el;
   }
}
```

El código se escribe en función de un tipo que hemos denotado como E y que no es conocido en el momento de la compilación.

Ese parámetro libre será fijado en el momento de creación del objeto.

Ejemplos de uso de un tipo parametrizado

El parámetro de tipo se especifica cuando se crea el objeto:

```
class Prueba{
   public static void main(String[] args){
      Contenedor<String> c = new Contenedor<String>();
      c.setElement("Esto es una cadena");
      int val = c.getElement().indexOf('n');
   }
}
```

En este código vemos que se ha creado una instancia del tipo `Contenedor` y el parámetro del tipo es de tipo `String`.

Por tanto para esta instancia <E> se sustituye por String.

Ventajas

1. Vemos que el inconveniente de realizar el *downcasting* ha desaparecido.
2. Además, ahora el compilador tiene información sobre lo que contiene y por tanto puede realizar la comprobación:

```
class Prueba{
   public static void main(String[] args){
      Contenedor<String> c = new Contenedor<String>();

      // Esta sentencia provoca un error de compilación
      // ya que c solo puede almacenar objetos del
      // tipo String
      c.setElement(new Integer(9));
   }
}
```


Índice

Motivación

En la parte 2 del tema vimos que se podían desarrollar algoritmos genéricos y que las operaciones sobre el tipo se pueden exigir mediante una interfaz.

En el caso de los tipos parametrizados también podemos exigir requisitos sobre el parámetro del tipo.

El motivo es que para el correcto funcionamiento del contenedor puede ser necesario que el tipo posea alguna operación.

Necesidad de que el tipo posea un método

Por ejemplo supongamos que queremos realizar un contenedor que mantenga ordenados los datos.

Está claro que para mantener los datos ordenados necesitamos compararlos:

```
class ContenedorOrdenado<T>{
   private T[] datos;
   private int numEl;

   public void inserta(T d){
      int cont = 0;
      while ((cont<numEl) & (datos[i] #\color{blue} compara con # d))
         cont++;
      ...
         }

   public E getMenor(){
      return datos[0];
   }
   ...
}
```

Se necesita que los datos se puedan comparar, pero ¿cómo se exige?.

Necesidad de que el tipo posea un método

Tanto el argumento que se pasa al método como los datos almacenados son del tipo `T`.

¿Qué mensajes se pueden enviar a través de referencias del tipo `T` y que pueda comprobar el compilador que es correcto?

Solo los métodos que están declarados en la clase `Object`.

Aproximación 1: declaración del contenedor con requisito en el parámetro del tipo

Al igual que hicimos al generalizar algoritmos, declaramos una interfaz con la operación que necesitamos:

```
public interface EsComparable{
   public int compara(EsComparable c);
}
```

y exigimos que el tipo del parámetro implemente a esa interfaz:

```
class ContenedorOrdenado<T extends EsComparable>{
   private T[] datos;
   private int numEl;

   public void inserta(T d){
      int cont = 0;
      while ((cont<numEl) & (datos[i].compara(d)<0))
         cont++;
      //...
   }
   //...
}
```

Aproximación 1: uso

Las clases cuyas instancias vamos almacenar en este contenedor deben implementar a la interfaz.

- Ejemplo 1:

```
class Empleado implements EsComparable{
   //...
   public int compara(EsComparable c){
      //Comparación de este empleado con el que pasan
      //Hay que hacer un cast de EsComparable a Empleado
      Empleado e = (Empleado)c;
      //...
   }
}
```

- Ejemplo 2:

```
class Figura implements EsComparable{
   //...
   public int compara(EsComparable c){
      //Comparación de esta figura con la que pasan
      //Hay que hacer un cast de EsComparable a Figura
      Figura f = (Figura)c;
      //...
   }
}
```


Aproximación 2: exigir el requisito pero parametrizar la interfaz

```
public interface EsComparable<T>{
   public int compara(T t);
}


class ContenedorOrdenado<T extends EsComparable<T>>{
   private T[] datos;
   private int numEl;

   public void inserta(T d){
      int cont = 0;
      while ((cont<numEl) & (datos[i].compara(d)==0))
         cont++;
      //...
   }
}
```


Aproximación 2: uso y ventajas

La clases cuyas instancias quiero almacenar en este contenedor deben implementar a la interfaz.

- Ejemplo 1:

```
class Empleado implements EsComparable<Empleado>{
   //...
   public int compara(Empleado e){
      //Comparación de este empleado con el que pasan
   }
}
```

- Ejemplo 2:

```
class Figura implements EsComparable<Figura>{
   //...
   public int compara(Figura e){
      //Comparación de esta figura con la que pasan
   }
}
```

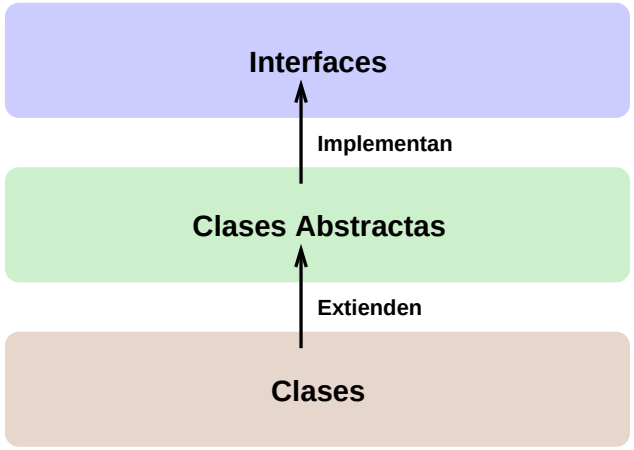
La ventaja de esta segunda aproximación es que nos evitamos el cast en la implementación de los métodos de la interfaz.

Índice

Collections Framework

El *Collections Framework* es una biblioteca de tipos que define contenedores de datos y algoritmos (ordenación y búsqueda).

Esta biblioteca de tipos está organizada del siguiente modo:



Algunas interfaces del Collections Framework

- `Collection<E>` Define los métodos que deben tener todas las clases (es la interfaz base)
- `Iterator<E>` Las colecciones ofrecen un método para obtener un iterador. El iterador permite recorrer los elementos de la colección.
- `Set<E> extends Collection<E>` Representa a un conjunto (no puede haber elementos repetidos).

Algunas interfaces del Collections Framework

- `SortedSet<E> extends Set<E>` Representa a un conjunto donde los elementos están ordenados.
- `NavigableSet<E> extends SortedSet<E>` Ofrece operaciones adicionales: obtener los mayores a uno dado, ...
- `List<E> extends Collection<E>` Grupo de elementos (puede haber elementos repetidos).
- `Map<K,V>` Representa a un objeto que almacena pares del tipo (clave,valor) y el acceso al valor se realiza a través de la clave. Es una generalización del *array*.

Algunas clases abstractas del Collections Framework

- `AbstractCollection<E> implements Collection<E>` Esqueleto de implementación de la interfaz `Collection`.
- `AbstractList<E> extends AbstractCollection<E>` Esqueleto de implementación para representar una lista de elementos con acceso aleatorio
- `AbstractSequentialList<E> extends AbstractList<E>` Esqueleto de implementación para representar una lista de elementos con acceso secuencial (para llegar al elemento `i` hay que pasar por los `i-1` anteriores).
- `AbstractSet<E> extends AbstractCollection<E> implements Set<E>` Esqueleto de implementación de la interfaz `Set`.
- `AbstractMap<K,V> implements Map<K,V>` Esqueleto de implementación de la interfaz `Map`.

Algunas clases del Collections Framework

`LinkedList<E> extends AbstractSequentialList<E> implements List<E> ...` Lista enlazada con acceso secuencial, por tanto si la aplicación requiere un almacén en el que se acceda a diferentes posiciones, este no será un buen almacén. Sin embargo, puede ser un candidato si simplemente queremos recorrer todos los elementos mediante un iterador.

`Vector<E> extends AbstractListimplements List<E> ...` En este caso el acceso es aleatorio. Utiliza un array para almacenar los datos y puede crecer de tamaño.

`ArrayList<E> extends AbstractListimplements List<E> ...` Similar a la anterior.

Algunas clases del Collections Framework

- `TreeSet<E> extends AbstractSet<E> implements NavigableSet<E> ...` Implementación del conjunto mediante un árbol binario donde las operaciones insertar, borrar y comprobar si tiene un elemento tienen un coste de O(log n).
- `HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>...` Las operaciones insertar y obtener tienen un coste constante O(1).

Es importante conocerlas ya que la elección del contenedor de los datos tiene una gran influencia en el rendimiento de la aplicación.

Índice

Recorrido de colecciones con un iterador

Tal y como hemos visto, a cualquier objeto que sea una colección le podemos enviar un mensaje para que nos devuelva un iterador.

```
Collection<T> col = ...;

Iterator<T> it = col.iterator();

T dato;
while (it.hasNext()){
   dato = it.next();
   // Procesar el dato
}
```

Ejemplo:

```
ArrayList<Figura> figs = new ArrayList<Figura>();
// Insertar figuras en el ArrayList

Iterator<Figura> it = figs.iterator();

Figura f;
while (it.hasNext()){
   f = it.next();
   // Hacer algo con la figura
}
```

Recorrido de colecciones con un bucle for

Cualquier objeto que sea una colección se puede recorrer mediante un bucle `for` del siguiente modo (también se puede usar con arrays):

```
Collection<T> col = ....;

for (T dato: col){
   // Hacer algo con el elemento dato
}
```

Ejemplo:

```
TreeSet<String> palabras = new TreeSet<String>();
// Insertar cadenas en el TreeSet

for (String p: palabras)
   System.out.println(p);
```

Este código usa internamente un iterador.

Índice

Incorporación de aspectos de la programación funcional en otros lenguajes

En la programación funcional las funciones se consideran *first class objects* ya que: se pueden pasar como argumento a otra función, una función puede devolver otra función como resultado, etc.

Algunas de estas ideas se han adoptado en otros lenguajes (que no son funcionales) como por ejemplo en JavaScript, Python o Java.

Este ejemplo es JavaScript:

```
// Función que recibe una función y un número como argumento
function apply(f,x){
    return f(x);
}
// Función que asignamos a una variable
var inc = function (x){
   return x+1;
}
// Invocamos a la función pasando la función que hemos definido
console.log(apply(inc,1))
```


Incorporación de aspectos de la programación funcional en otros lenguajes

JavaScript permite escribir el código anterior de forma más concisa pasando directamente la implementación de la función en el argumento:

```
// Función que recibe una función y un número como argumento
function apply(f,x){
    return f(x);
}
// Invocamos a la función pasando una función
// que definimos en el propio argumento:
console.log(apply(x => x+1,1))

// Invocamos a la función pasando otra función
// que definimos en el propio argumento:
console.log(apply(x => x*x,4))
```

Interfaces funcionales

Java ofrece un conjunto de interfaces que solo declaran un método.

A partir de la versión 8 de Java a estas interfaces se les llama **interfaces funcionales** (por analogía con el paradigma funcional).

Tomemos por ejemplo la interfaz:

```
public interface Predicate<T>{
   public boolean test(T t);
}
```

y la clase

```
public class Contenedor<T>{
   public List<T> cumplenCondicion(Predicate<T> p){
      // Devuelve una lista con el subconjunto de los elementos que cumplen
      // una condición.
      // La condición viene encapsulada en el objeto del tipo Predicate
      // que nos pasan como argumento
   }
}
```


Interfaces funcionales

El método `cumplenCondicion` indica que hay que pasar una instancia que podamos referenciar mediante una referencia del tipo `Predicate`.

Tenemos dos opciones:

Declarar una clase que implemente a la interfaz `Predicate` y pasar una instancia al constructor

```
class P implements Predicate<String>{
   public boolean test(String s){
      return s.contains("the");
   }
}

Contenedor<String> c = ...;
List<String> lista = c.cumplenCondicion(new P());
```


Interfaces funcionales

Usar una clase anónima (hacemos la declaración y la implementación en el argumento)

```
Contenedor<String> c = ...;
List<String> lista = c.cumplenCondicion(new Predicate<String>(){
      public boolean test(String s){
         return s.contains("the");
      }});
```


Expresiones lambda

Java 8 introduce las expresiones *lambda* que pretenden simplificar el uso de las interfaces funcionales.

La expresión lambda implica la creación de una instancia de una clase que implementa la interfaz funcional (como en el ejemplo de la página anterior) pero sin usar el operador `new` ni hacer toda la declaración de la clase.

Las expresiones lambda constan de tres partes:

- Lista de argumentos del método declarado en la interfaz
- Flecha
- Cuerpo del método

Expresiones lambda

Cuando en el cuerpo aparece una expresión, equivale a evaluar la expresión y devolver el resultado.

Cuando en el cuerpo aparece un bloque de código (`{...}`) y el método debe devolver un resultado entonces debe aparecer explícitamente la sentencia `return`.

Ejemplos:

Lista de argumentos	Flecha	Cuerpo
`(int x, int y)`	`->`	`x+y`
`(x, y)`	`->`	`{int z=x+1; return z+y;}`
`()`	`->`	`{System.out.println("Texto");}`
`(String s)`	`->`	`{System.out.println(s);}`
`s`	`->`	`{System.out.println(s);}`

En aquellos casos donde no aparece el tipo en la lista de argumentos, el compilador debe inferirlos.

Expresiones lambda

El ejemplo anterior se podría escribir usando una expresión lambda y quedaría un código muy compacto:

```
Contenedor<String> c = ...;
List<String> lista = c.cumplenCondicion( s -> s.contains("the"));
```

donde:

- Estamos pasando una referencia a un objeto del tipo `Predicate<String>`
- `s` es el argumento del **único método** que define la interfaz
- `s.contains("the")` es la **implementación** de ese método (que en este caso es una sola sentencia).

Nótese la similitud entre la expresión lambda en Java y la notación que usa en JavaScript mostrada al inicio de la sección.

Algunas interfaces funcionales I

Java 8 define cuatro grupos de de interfaces funcionales en el paquete `java.util.function`. A continuación se muestran algunas interfaces funcionales:

```
public interface Predicate<T>{
   // Para realizar una comprobación sobre t
   boolean test(T t);
}
```

```
public interface Supplier<T>{
   // Para obtener una instancia de t
   T get();
}
```

```
public interface Consumer<T>{
   // Para realizar alguna acción con t sin devolver nada
   void accept(T t);
}
```


Algunas interfaces funcionales II

```
public Function<T,R>{
   // A partir de un objeto del tipo T devuelve un objeto del tipo R
   R apply(T t);
}
```

```
public UnaryOperator<T>{
   // Recibe algo del tipo T y devuelve algo del tipo T
   T apply (T t);
}
```

```
public interface BiFunction<P,S,R>{
   // A partir de argumentos del tipo P y S devuelve uno del tipo R
   R apply(P p, S s);
}
```

```
public interface BinaryOperator<T> extends BiFunction<TTT>{
   // A partir de dos argumentos del tipo T devuelve algo del tipo T
   T apply(T t1, T t2);
}
```

Interfaces funcionales y expresiones lambda

Vamos a ver un ejemplo de cómo usar la interfaz `Predicate<T>` para implementar un método en un contenedor de forma que devuelva el número de elementos que cumplan una propiedad (desconocida en el momento de implementar el contenedor).

```
class Contenedor<T>{
   private ArrayList<T> datos;
   //...
   public int cumplenPredicado(Predicate<T> prop){
      int num=0;
      for (T t: datos)
         if (prop.test(t))
            num++;
      return num;
   }
}
```


Interfaces funcionales y expresiones lambda

Supongamos que tenemos ahora dos clases que pueden ser almacenadas en el contenedor.

Por un lado la clase `Empleado`:

```
class Empleado{
   private int edad;
   //...
   public int getEdad(){
      return edad;
   }
}
```

y por otro la clase `VariacionBolsa`:

```
class VariacionBolsa{
   private double variacion;
   //...
   public double getVariacion(){
      return variacion;
   }
}
```


Interfaces funcionales y lambdas

```
class Prueba{
   public static void main(String[] args){
      Contenedor<Empleado> ce = new ...;
      ce.cumplenPredicado( e -> e.getEdad()>60);

      Contenedor<VariacionBolsa> cv = new ...;
      cv.cumplenPredicado( v -> v.getVariacion()>0 );
      cv.cumplenPredicado( v -> v.getVariacion()<=0 );

      // La última sentencia es equivalente a esta otra (creación explítica
↪  de una
      // instancia de una clase anónima que implementa a la interface
↪  Predicate)
      System.out.println(cv.cumplenPredicado( new Predicate<VariacionBolsa>(){
            public boolean test(VariacionBolsa v){
               return v.getVariacion() <= 0;
            }
         }));
   }
}
```


Índice

Motivación

Cuando tenemos una serie de elementos (que pueden estar almacenados en una colección, en un fichero, etc) es habitual tener que realizar operaciones sobre ellos:

- Filtrarlos para obtener los que cumplen un determinado criterio
- Encontrar los *N* primeros que cumplen un determinado criterio
- Encontrar el mínimo, máximo, etc
- Realizar un mapeo para extraer otro tipo de dato (por ejemplo, a partir de datos del tipo Empleado obtener los nombres, que será del tipo String).
- Realizar una operación sobre todos los datos para obtener un resultado (por ejemplo una acumulación, la multiplicación, etc).

Para realizar estas operaciones, es necesario realizar uno o varios bucles `for` y en el cuerpo del bucle se procesa cada elemento de la colección.

Sin embargo, los *streams* permiten realizar estas operaciones ofreciendo métodos que declaran parámetros que son interfaces funcionales (por lo tanto se usan lambdas).

Obtención de un Stream

Las colecciones definen métodos que permiten obtener un *stream*:

- `Stream<E> stream()` para un procesado secuencial.
- `Stream<E> parallelStream()` para un procesado en paralelo de los elementos.

La clase `BufferedReader` (que permite leer de un fichero de texto) ofrece el método:

- `Stream<String> lines()` que permite obtener un `Stream` con las líneas del fichero

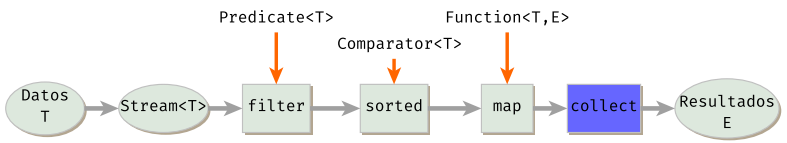
La clase *Stream* ofrecen métodos estáticos. Por ejemplo:

- `Stream<T> Stream.of(T[])`
- `IntStream IntStream.range(int ini,int fin)` devuelve un `IntStream` con datos enteros que van desde `ini` hasta `fin` en pasos de 1.

Operaciones sobre un Stream

Se puede distinguir entre diferentes tipos de operaciones:

- **Operaciones intermedias** que devuelven otro *stream* y se pueden concatenar.
- **Operaciones terminales** que devuelven un valor y solo puede haber una de ellas en una expresión.
- **Operaciones que cortocircuitan** que finalizan el procesado cuando se cumplen las condiciones adecuadas (pueden ser intermedias o terminales).



Métodos de Stream<T>: operaciones intermedias

Ejemplos de operaciones intermedias:

- `Stream<T> filter(Predicate<? super T> p)`: devuelve solo los elementos que cumplen el predicado.
- `Stream<R> map(Function<? super T, ? extends R> mapper)`: aplica una transformación a los elementos del `Stream`.
- `Stream<T> skip(long n)`: descarta los `n` primeros elementos y devuelve el resto.
- `Stream<T> sorted(Comparator<? super T> comp)`: devuelve un `Stream` con los elementos ordenados.
- `Stream<T> distinct()`: el `Stream` resultante tiene elementos distintos (comparados usando `equals`).
- `Stream<T> peek(Consumer<? super T> accion)`: a cada elemento del `Stream` le aplica la acción pasada como argumento (está pensado para depurar).

Métodos de Stream<T>: operaciones terminales

Ejemplos de operaciones terminales:

- `void forEach(Consumer<? super T> action)`: aplica una acción a cada elemento del `Stream`
- `Optional<T> max(Comparator<? super T> comp)`: obtiene el mayor elemento del `Stream`
- `Optional<T> min(Comparator<? super T> comp)`: obtiene el menor elemento del `Stream`
- `long count()`: cuenta los elementos del `Stream`
- `Object[] toArray()`: devuelve los elementos que contenga el `Stream` en un array
- `T reduce(T identity, BinaryOperator<T> accumulador)`: realiza una reducción aplicando el operador sobre cada dato del `Stream` y el acumulador. El acumulador inicialmente tiene el valor `identity`. Se puede usar para sumar, multiplicar, etc.
- `R collect(Collector<? super T,A,R> collector)`: permite obtener los elementos del `Stream` como una colección o realizar agrupaciones de los datos. La clase `Collectors` ofrece métodos estáticos para obtener `Collector`. Ejemplo: `Collectors.toList()`.

Métodos de Stream<T>

Ejemplos de operaciones intermedias que corto-circuitan:

- `Stream<T> limit(long tam)`

Ejemplos de operaciones terminales que corto-circuitan:

- `boolean anyMatch(Predicate<? super T> pred)`: comprueba si hay algún elemento que cumple el predicado
- `Optional<T> findAny()`: devuelve un elemento
- `Optional<T> findFirst()`: devuelve el primer elemento del `Stream`

Ejemplo I

```
// imports

class Empleado{
   private String nombre;
   private String departamento;
   private int edad;

   Empleado(String nombre, String departamento, int edad){
      this.nombre=nombre;
      this.departamento=departamento;
      this.edad=edad;
   }

   public void setNombre(String arg){
      this.nombre=arg;
   }
   public String getNombre(){
      return nombre;
   }
   public void setEdad(int arg){
      this.edad=arg;
   }
```


Ejemplo II

```
   public int getEdad(){
      return edad;
   }

   public String getDepartamento() {
         return departamento;
   }

   public void setDepartamento(String departamento) {
         this.departamento = departamento;
   }
}
class EjemploStream{
   public static void main(String[] args){
      ArrayList<Empleado> empleados = new ArrayList<Empleado>();
      empleados.add(new Empleado("Pedro","IT",43));
      empleados.add(new Empleado("Antonio","RRHH",32));
      empleados.add(new Empleado("Rodrigo","IT",25));
      empleados.add(new Empleado("Carlos", "RRHH",20));
      empleados.add(new Empleado("Pablo", "IT", 28));

      System.out.println("Empleados mayores de 27 años:");
      empleados.stream().filter(e -> e.getEdad()>27).map(e ->
↪  e.getNombre()).forEach(n -> System.out.println(n));
```

Ejemplo III

```
      Optional<Empleado> min = empleados.stream().min((e1,e2) ->
↪  e1.getEdad()- e2.getEdad());
      min.ifPresent(e -> System.out.println("Empleado de menor edad: " +
↪  e.getNombre()));

      // Edad media
      System.out.println("Edad media de los empleados:");
      double suma = empleados.stream().mapToDouble((e) ->
↪  e.getEdad()).reduce(0, (acc,dato) -> acc+dato);
      System.out.println(suma/empleados.size());
   }
}
```

Capítulo 4

Parte 4: Excepciones

Tema 1 - Parte 4
Programación orientada a objetos

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Realizar código donde se traten excepciones (`try`, `catch`, `finally`).
2. Crear nuevas excepciones y realizar métodos que las lancen (`throws`, `throw`).
3. Explicar las diferencias (respecto al lanzamiento y al tratamiento) entre excepciones que extienden a `Exception` y las que extienden a `RuntimeException`.
4. Realizar la traza de un programa a partir del código fuente en el que aparecen excepciones.
5. Crear aplicaciones en las que se traten excepciones (ya sean las que proporciona el lenguaje o las creadas según una especificación).

Índice

Errores durante la ejecución

- ¿Qué ocurre si un programa está leyendo un archivo que está en una unidad extraíble y se extrae antes de tiempo?
- ¿Qué ocurre si el usuario de un programa introduce un carácter por teclado, p.ej. "W", cuando la aplicación solicita un número?
- Las aplicaciones deben de ser robustas y saber manejar situaciones inesperadas (errores). En caso contrario pueden pasar cosas como estas:
  El lanzador de satélites Ariane 5 explotó los pocos segundos de su lanzamiento. El error: una excepción en la conversión de coma flotante de 64-bits a entero de 16-bits. La información de excepción fue interpretada como datos y hicieron variar el ángulo del cohete y explotó.

¿Qué hacer cuando de detecta un error?

Opción 1: No se tratan: la aplicación finaliza con un error.

Opción 2: Quien detecta el error decide qué hacer. Si se trata de una librería y no de una aplicación final, esta opción es poco flexible puesto que no sabemos en qué contexto se usará el código.

```
public void desapilar () {
   if ( this.esVacia() )
      System.err.println ("Error: Desbordamiento inferior.");
   else
      laCima = laCima.siguiente();
}
```

¿Tiene sentido mostrar por consola? No sabemos en qué condiciones se va a ejecutar la aplicación.

¿Qué hacer cuando de detecta un error?

Opción 3 Quien detecta el error informa de que el error se ha producido.

```
public boolean desapilar () {
   boolean res = false;
   // Detectamos el error
   if ( this.esVacia() )
      res = false ;
   else {
      laCima = laCima.siguiente();
      res = true ;
   }
   // En el valor devuelto se informa del éxito o error
   return res;
}
```

¿Qué ocurre si en un método se pueden dar diferentes causas de error?

¿Cómo informo? ¿Con un `int` donde cada entero representa un posible error? Esto sería poco legible.

¿Qué hacer cuando de detecta un error? Excepciones

Un método puede retornar dos tipos de información:

1. Datos (con return)
2. Excepciones (con throw)

Existen unos requisitos para que un método pueda funcionar correctamente, si estos requisitos no se cumplen y se produce un error podemos informar mediante una excepción.

Por ejemplo: un método que lee datos de un flujo:

Requisito: que el medio de donde se leen los datos esté disponible
Excepción: no es posible leer (se ha desconectado el usb, se ha perdido la conexión con la máquina remota, etc).

Fase 1: lanzar la excepción cuando se detecta un error

En el código de un método se puede lanzar una excepción mediante la palabra reservada `throw`:

```
throw new TipoExcepcion(...);
```

donde se crea un objeto de alguna subclase de la clase `Exception`.

El lanzamiento de una excepción implica que el método finaliza inmediatamente (si la excepción no es tratada dentro del método).

La excepción debe aparecer en la signatura del método y debe ser documentada:

```
/** Método que desapila el dato que está en la cima
  * @throws Exception si no hay datos en la pila
  */
public void desapilar () throws Exception{
   if (esVacia())
      throw new Exception();
   else{
      // Codigo
   }
}
```


Fase 2: recoger y tratar una excepción

Ahora nos ponemos en el rol de quien realiza la llamada a un método que puede lanzar una excepción.

Para poder recoger y tratar una excepción es preciso declarar un bloque `try` que envuelve a la llamada al método, y el tratamiento de la excepción (si se produce) se realiza dentro de un bloque `catch`:

```
try{
   // Llamadas a métodos que pueden lanzar una excepción
}
catch (TipoExcepcion e) {
   // Si se produce una excepción del tipo TipoExcepcion se ejecuta este código
   //Sentencias que tratan la excepción e
}finally{
   //Sentencias que se ejecutan se lance o no la excepción
}
```


Documentar las excepciones

Cualquier excepción lanzada por un método es una parte pública de él: los usuarios del método deben conocer qué excepciones puede lanzar para poder decidir qué hacer con ellas.

Las excepciones que puede lanzar un método tienen el mismo nivel de importancia que los argumentos o el valor de retorno del método. Por lo tanto aparecen en la documentación.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/InputStream.html#read()

```
// Método de la clase InputStream
public abstract int read() throws IOException

// Documentación del método
  Returns:
     the next byte of data, or -1 if the end of the stream is reached.
  Throws:
     IOException - if an I/O error occurs.
```

Índice

Ejemplos de clases que representan excepciones en Java (el nombre de la clase de la excepción informa del problema):

`ClassCastException`, `ClassNotFoundException`, `FileNotFoundException`, `IndexOutOfBoundsException` , `InterruptedException`, `IOException`, `MalformedURLException`, `NullPointerException`, `NumberFormatException`, `RemoteException`, `SecurityException`, `SocketTimeoutException`, `SQLException`,...

Puede ser que en el dominio del problema sobre el que estemos trabajando sea necesario definir nuevas excepciones que representen errores específicos.

Ejemplo de declaración de una excepción

Por ejemplo, supongamos que un método puede lanzar una excepción si su tiempo de ejecución es demasiado largo. Llamemos a esta excepción `LongExecutionException`.

Creamos una nueva clase que representa a esta excepción:

```
public class LongExecutionException extends Exception {
   public LongExecutionException ( long t ) {
      super ("Tiempo de ejecución mayor a: "+ t);
   }
}
```


Ejemplo de uso de la excepción en un método

Ejemplo de uso de la excepción anterior:

```
class ConsultasDB<E>{
   public void inserta(E dato) throws LongExecutionException{
      // Código
      // Si detectamos que la conexión a la BD
      // tarda mucho:
         throw new LongExecutionException(t);
      //
      // Resto del código
   }
   //...
}
```

Ejemplo de llamada al método que puede lanzar la excepción

```
public class Ejemplo{
   public static void main(String[] args){
      ConsultaBD<Estudiante> c = new ConsultaBD<>();
      List<Estudiante> lista = new ArrayList<>();
      // Añadir estudiantes a la lista

      // Insertar los estudiantes a la BD
      for (Estudiante e: lista)
         try{
            c.inserta(e);
         }catch(LongExecutionException ex){
            // Qué hacer si se produce la excepción
         }
   }
}
```

En este caso si se produce una excepción, el bucle `for` se sigue ejecutando (ya que la excepción es tratada dentro del bucle).

Ejemplo de llamada al método que puede lanzar la excepción

```
public class Ejemplo{
   public static void main(String[] args){
      ConsultaBD<Estudiante> c = new ConsultaBD();
      List<Estudiante> lista = new ArrayList<>();
      // Añadir estudiantes a la lista

      // Insertar los estudiantes a la BD
      try{
         for (Estudiante e: lista)
            c.inserta(e);
      }catch(LongExecutionException ex){
        // Qué hacer si se produce la excepción
      }
   }
}
```

En este otro ejemplo, cuando se produzca la primera excepción se salta al bloque `catch` y se corta la ejecución del bucle `for`.

Ejemplo de llamada al método que puede lanzar la excepción

```
public class Ejemplo{
   public static void main(String[] args) throws LongExecutionException{
      ConsultaBD<Estudiante> c = new ConsultaBD();
      List<Estudiante> lista = new ArrayList<>();
      // Añadir estudiantes a la lista

      // Insertar los estudiantes a la BD
      for (Estudiante e: lista)
            c.inserta(e);
   }
}
```

En este ejemplo no se trata la excepción y si se produce, la aplicación finaliza con la excepción.

Índice

Exception vs RuntimeException

Hay dos categorías de excepciones en Java:

Comprobadas (ckecked): necesariamente hay que tratarlas (`Exception` y sus subclases)

No comprobadas (unchecked): el lenguaje no requiere que se traten (`RuntimeException`, `Error` y sus subclases)

Las clases que proporciona Java que extienden a `RuntimeException` representan problemas tales como:

1. excepciones aritméticas (dividir por cero),
2. excepciones con referencias (enviar un mensaje a null),
3. excepciones de indexación (acceder a un elemento de un *array* mediante un índice que sobrepasa los límites del array).
4. conversión a número de una cadena que no representa un número, etc.

Estas excepciones ocurren cuando una precondición en un parámetro del método no se cumple.

Exception vs RuntimeException

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEA

get

```
public E get(int index)
```

Returns the element at the specified position in this list.

Specified by:
`get` in interface `List<E>`

Specified by:
`get` in class `AbstractList<E>`

Parameters:
`index` - index of the element to return

Returns:
the element at the specified position in this list

Throws:
`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

Exception vs RuntimeException

Estas excepciones pueden ocurrir en cualquier parte del programa y si se nos forzara a tratarlas tendríamos que envolver todo el código en bloques `try-catch`.

Por ello el compilador **no fuerza** ni a declarar que se lanzan en la signatura del método ni a su tratamiento cuando llamamos a un método que las puede lanzar.

Parte II

Tema 2: Programación concurrente

Tema 2
Programación concurrente

Juan Gutiérrez

Programación
GIT
Universitat de València

Índice

Objetivos

1. Declarar clases que contengan código que se puede ejecutar concurrentemente
2. Ejecutar código de forma concurrente (tanto con `Thread` como con `Runnable`)
3. Reunificar tareas que se están ejecutando concurrentemente
4. Identificar la sección crítica en código
5. Usar semáforos para controlar el acceso a la sección crítica
6. Usar synchronized para conseguir la exclusión mutua
7. Sincronizar tareas para cooperación mediante un monitor

Índice

Concurrencia

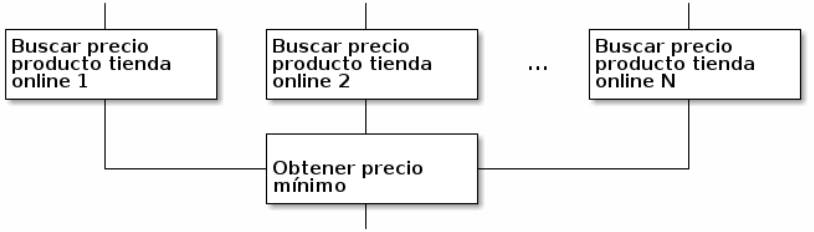
Una máquina puede ejecutar de forma concurrente un conjunto de procesos (programa en ejecución para el que el sistema operativo proporciona recursos tales como memoria, ficheros, etc.).

La posibilidad de ejecutar varios procesos concurrentemente es conveniente ya que ofrece:

- Mejor uso de recursos: si un programa en un proceso está en espera (ej. operación de entrada/salida) otro proceso se puede ejecutar en ese tiempo muerto.
- Reparto del tiempo: se puede realizar un reparto del tiempo asignando intervalos de tiempo a cada proceso (es mejor que esperar a que un proceso finalice para ejecutar otro).
- Conveniencia: en lugar de escribir un programa que realice todas las tareas, puede resultar más fácil escribir varios programas cada uno de los cuales realiza una única tarea y hacer que se coordinen.

Ejemplo: tareas en secuencia

Ejemplo de tareas en paralelo



Concurrencia Física: Existe más de un núcleo por lo que varias tareas se ejecutan realmente de forma simultánea.

Por ejemplo, especificaciones de un procesador Intel Xeon Platinum:

Especificaciones de la CPU

Cantidad de núcleos	56
Cantidad de subprocesos	112
Frecuencia turbo máxima	3.80 GHz
Frecuencia básica del procesador	2.00 GHz
Caché	105 MB

Concurrencia Lógica: Asumir la existencia de varios núcleos (aunque no existan físicamente).

El sistema operativo en última instancia se encargará de «mapear» la concurrencia lógica sobre el hardware realmente disponible.

Un Lenguaje de Programación será concurrente si proporciona las estructuras necesarias para definir y manejar diferentes tareas (hilos de ejecución) dentro de un programa.

El paso de información entre procesos no es especialmente sencillo: sockets, memoria compartida, ficheros, IPC, etc.

Por el contrario, los hilos que se ejecutan en un proceso comparten los recursos del proceso (memoria, ficheros, etc), por tanto, es más fácil compartir datos entre hilos de un proceso que entre diferentes procesos.

Un hilo representa un contexto de ejecución (que almacena cuál es la siguiente instrucción, los valores de las variables, etc). Si un núcleo está ejecutando un hilo y debe continuar con la ejecución de otro hilo, tendrá que guardar el contexto de ejecución actual y cargar el contexto de ejecución del hilo a ejecutar.

Sin embargo, el hecho de que varios hilos accedan a los mismos datos puede dar lugar a resultados inesperados que no ocurren cuando se tiene un único flujo de ejecución.

Interfaces gráficas de usuario

En las interfaces gráficas de usuario hay un hilo llamado *event dispatch thread* que se encarga de ejecutar el código en respuesta a un evento producido en un componente.
Si se ejecutara en el mismo hilo, la interfaz no respondería hasta que no se acabara de ejecutar el código de tratamiento del evento.
Al ejecutarse en un hilo separado, la interfaz sigue respondiendo.

Ejemplos donde se usan hilos

Servidores

Una posibilidad para atender a múltiples clientes desde un servidor es ejecutar el código que trata la conexión/comunicación con cada cliente en un hilo separado.
El cliente envía información, la información es procesada en el servidor y finalmente se devuelve una respuesta desde el servidor.
De este modo un cliente no tiene que esperar a que finalice la petición de otro cliente para que se trate su petición.

Índice

La clase Thread

`Thread` es la clase base de Java para definir hilos de ejecución concurrentes dentro de un mismo programa.

En Java, como lenguaje orientado a objetos, el concepto de concurrencia está asociado a los objetos: instancias de la clase `Thread` pueden ejecutar código concurrentemente.

Desde el punto de vista de la programación, para realizar código que se puede ejecutar concurrentemente hay que hacer una clase que extienda a `Thread`:

```
class ClaseConCodigoConcurrente extends Thread {...}
```


Esqueleto de la clase Thread

Esqueleto de la clase Thread:

```
class Thread{
   ...
   Thread(){...}

   Thread(Runnable r){...}

   public void run(){}

   public void start(){...}

   public void sleep(long tiempo) throws InterruptedException{...}

   public void join() throws InterruptedException{...}
   ...
}
```

Método run()

La clase `Thread` implementa el método `run` vacío:

```
public void run (){}
```

Este método se debe sobreescribir en la subclase y contendrá las sentencias que se pueden ejecutar de forma concurrente.

La ejecución del método `run` de un `Thread` puede realizarse concurrentemente con otros métodos run de otros `Thread` y con el método `main` (el método `main` se ejecuta en un hilo llamado `main`).

Método start()

La ejecución concurrente (en un hilo separado) se realiza cuando se envía el mensaje `start()` (que es un método implementado en la clase `Thread`).

`start()` es un método especial que crea un hilo, comienza a ejecutar el método `run()` y devuelve inmediatamente el control para que se sigan ejecutando las sentencias que van tras la llamada a `start()`.

Resumiendo:

Las tareas que se pueden ejecutar concurrentemente se definen en el método `run()`. Pero para ejecutarlas concurrentemente se invoca al método `start()`.

Ejemplo I

```
class Hilo extends Thread {
   private String url;
   private String file;
   public Hilo (String url, String file) {
      this.url = url;
      this.file = file;
   }

   public void run() {
      System.out.println("Descargando " + url);
      // Descargar el contenido de la URL y guardarlo en el fichero
      System.out.println("Contenido guardado en " + file);
   }
}

public class Demo {
   public static void main (String[] args) {
      Hilo uno, dos;

      uno = new Hilo("https://host1/recurso1.png","r1.png");
      dos = new Hilo("https://host2/recurso2.pdf","r2.pdf");
```

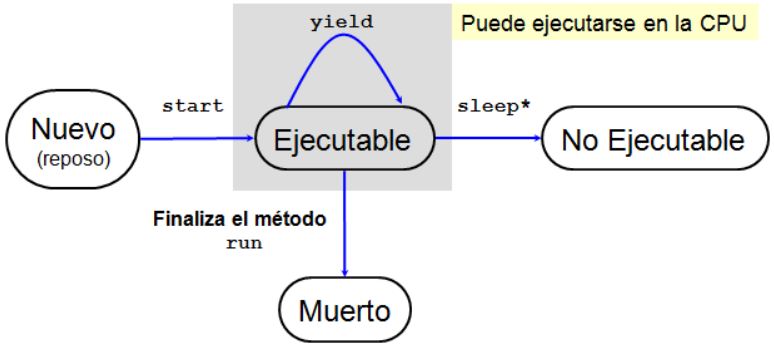

Ejemplo II

```
      uno.start();
      dos.start();

      System.out.println("fin del main");
   }
}
```

Un posible resultado al ejecutar esa aplicación:

```
fin del main
Descargando https://host1/recurso1.png
Descargando https://host2/recurso2.pdf
Contenido guardado en r1.png
Contenido guardado en r2.pdf
```


Ciclo de vida de un hilo



Un hilo está «vivo» si se encuentra en el estado «Ejecutable» o «No ejecutable»:

`boolean isAlive()`

La interfaz Runnable

Problema: ¿Qué ocurre si se desea incluir código que se puede ejecutar de forma concurrente en una clase que ya extiende a otra?.

Ejemplo:

```
class Circulo extends Figura {....}
```

No puede extender también a `Thread` ya que Java prohíbe la herencia múltiple.

Solución: Java proporciona la interfaz `Runnable` (Java admite heredar de una sola clase pero implementar múltiples interfaces).

Esta interfaz declara un único método: `public void run()`

La interfaz Runnable

La clase implementa a `Runnable` y el código que se puede ejecutar concurrentemente se define en el método `run()`:

```
class Circulo extends Figura implements Runnable {
   ...
   public void run(){
      ...
   }
   ...
}
```

Ahora a un objeto de este tipo no le podemos enviar el mensaje `start()` ya que este método está declarado en `Thread`.

Para ejecutar el código concurrentemente hay que crear un `Thread` al que se le pasa en el constructor la instancia de `Runnable`.

Al enviar el mensaje `start()` a la instancia de `Thread` el `run()` que se ejecuta es de `Runnable`.

Ejemplo

El siguiente código ilustra esta situación:

```
class EjemploUsoRunnable{
   public static void main(String[] args){

      Thread h1 = new Thread(new Circulo());
      Thread h2 = new Thread(new Circulo());

      h1.start();
      h2.start();
   }
}
```


Reunificación de tareas

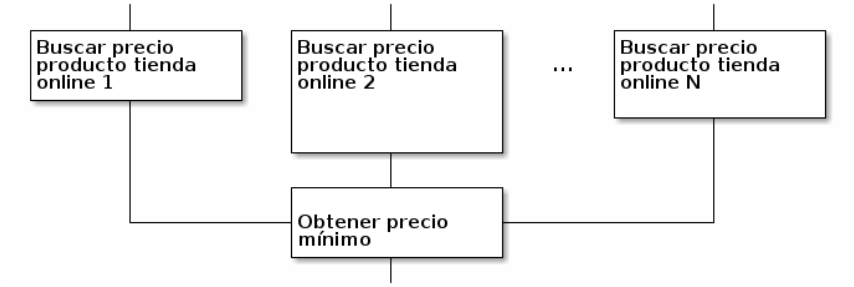
Un programa concurrente, en general, tendrá bloques de ejecución de código concurrente y bloques de ejecución secuencial donde para continuar será necesario que el código concurrente haya finalizado.

Después de bloques concurrentes puede ser preciso reunificar el flujo de control del programa para continuar de forma secuencial.

Para ello se utiliza el método `join()` de `Thread`.

Reunificación de tareas

Unas peticiones pueden tardar más que otras, hasta que no finalicen todas no podemos obtener cuál es la que nos ofrece el precio más bajo:



Ejemplo de reunificación de tareas

Este código suma los elementos del vector que se pasa en el constructor y ofrece un método para consultar la suma:

```
public class SumaVector extends Thread{
   private int[] vector;
   private double suma;

   SumaVector(int[] v){
      vector = v;
      suma = 0;
   }

   public void run(){
      for (int i=0;i<vector.length;i++)
         suma += vector[i];
   }

   public double getSuma(){
      return suma;
   }
}
```

Ejemplo de reunificación de tareas

Queremos sumar cada vector en un hilo y finalmente mostrar la suma de los dos vectores:

```
public class Principal {
   public static void main(String[] args){
      int[] v1 = new int[100000000];
      int[] v2 = new int[100000];

      for (int i=0;i<v1.length;i++)
         v1[i] = (int)(Math.random()*10);

      for (int i=0;i<v2.length;i++)
         v2[i] = (int)(Math.random()*10);

      SumaVector s1 = new SumaVector(v1);
      SumaVector s2 = new SumaVector(v2);

      s1.start();
      s2.start();

      try{
         s1.join();
         s2.join();
      }catch(InterruptedException ex){
         ex.printStackTrace();
      }
      // Es necesario reunificar (esperar a que finalicen).
      // Si no esperamos se mostraría un resultado intermedio incorrecto
      System.out.println(s1.getSuma()+s2.getSuma());
   }
}
```


Índice

Sincronización de tareas

La **sincronización** es el mecanismo que controla el acceso a un recurso compartido desde diferentes tareas concurrentes.

Tipos de sincronización:

- **Exclusión mutua**: un hilo H_1 ejecuta código que accede a un dato x mientras que otro hilo H_2 ejecuta código que está accediendo al mismo dato (los dos hilos comparten un recurso). **Objetivo**: El hilo H_2 debe de esperar que el hilo H_1 finalice el uso de x para poder continuar.
- **Cooperación**: un hilo H_1 ejecuta un código que no puede ejecutarse mientras no se produzca un cambio de estado en el recurso compartido. El cambio de estado en el recurso lo produce la ejecución de otro método del recurso compartido desde otro hilo H_2. Ejemplo: **productor/consumidor**.

Recursos compartidos y secciones críticas

En el caso de la exclusión mutua, deseamos que cada tarea ejecute un conjunto de sentencias como si fuese una operación indivisible o atómica (de forma que la otra tarea no interfiera).

Este conjunto de sentencias se conoce como **sección crítica** (que si se ejecutan simultáneamente desde varios hilos puede dar lugar a errores).

Nótese que los errores se solo se pueden dar si las dos tareas ejecutan código desde el que se accede a las mismas variables.

Si dos tareas ejecutan código que no comparte variables no hay necesidad de exigir exclusión mutua.

Ejemplo I

El siguiente ejemplo ilustra los errores que se pueden producir cuando no se usa exclusión mutua.

```
class Contador{
   private double valor = 0;

   // Método para incrementar el valor del contador
   public void incrementar(){
      valor++;
   }

   // Método para decrementar el valor del contador
   public void decrementar(){
      valor--;
   }

   // Método para consultar el valor del contador
   public double getValor(){
      return valor;
   }
}
```


Ejemplo II

```
// Tarea concurrente que incrementa 2000 veces
class Incrementador extends Thread{
   private Contador cont;

   // Al constructor se le pasa una referencia al recurso compartido
   Incrementador(Contador c){
      cont = c;
   }
   public void run(){
      for (int i=0;i<2000;i++)
         cont.incrementar();
   }
}

// Tarea concurrente que decrementa 2000 veces
class Decrementador extends Thread{
   private Contador cont;

   // Al constructor se le pasa una referencia al recurso compartido
   Decrementador(Contador c){
      cont = c;
   }
   public void run(){
```


Ejemplo III

```
      for (int i=0;i<2000;i++)
         cont.decrementar();
   }
}

// Clase con el método main
class Prueba{

   public static void main(String[ ] args){
      // Se crea una instancia del recurso
      Contador c = new Contador();

      // Esa instancia se comparte en los dos hilos
      Incrementador inc = new Incrementador(c);
      Decrementador dec = new Decrementador(c);

      // Se lanzan los hilos
      inc.start();
      dec.start();

      try{
         inc.join();
         dec.join();
```


Ejemplo IV

```
      }catch(InterruptedException ex){}

      // Mostramos el valor del contador cuando finalizan los dos hilos
      System.out.println(c.getValor());
   }
}
```

Ejemplo

Un hilo incrementa 2000 veces y el otro decrementa 2000 veces, el resultado debería ser cero, pero...

Ejecución 1: -369

Ejecución 2: 286

Ejecución 3: -624

Ejecución 4: -434

Ejecución 5: -1256

Ejecución 6: 35

Ejecución 7: 612

Ejecución 8: 858

Ejecución 9: 1148

Ejecución 10: 0

Algunos mecanismos para conseguir la sincronización

Ejemplos de mecanismos que ofrece Java para la sincronización:

- **Semáforos**: exclusión mutua mediante la clase `Semaphore`
- **synchronized**: exclusión mutua mediante métodos y bloques de código `synchronized`.
- **Monitores**: un mecanismo conseguir la sincronización: detener tareas que no pueden continuar y avisar a tareas que están en espera de que quizá pueden continuar. En Java cualquier objeto puede ser un monitor ya que este comportamiento se define en la clase `Object`

Índice

Semáforos

Fueron propuestos por Dijkstra en 1965. Un semáforo es una estructura de datos que consiste en:

- Un valor entero (número de permisos). Si el valor es cero significa que el hilo no puede continuar.
- Dos operaciones: adquirir y liberar.
- Internamente usa una cola donde se almacenan las tareas que no pueden continuar.

Operaciones del semáforo

El semáforo ofrece dos operaciones que se ejecutan de forma **atómica**:

- **adquirir** Si hay permisos (el valor es mayor que cero) resta (decrementa el valor) y el hilo puede continuar con la ejecución (entrar en la sección crítica). Si no hay permisos (el valor es cero), el hilo se bloquea hasta que hayan permisos.
- **liberar** Si hay algún hilo bloqueado en el semáforo lo activa. En caso contrario devuelve un permiso (incrementa el valor) al semáforo.

Uso de semáforos para exclusión mutua

Los semáforos binarios (donde el valor entero toma el valor 0 o 1) se utilizan para implementar la exclusión mutua de tareas en el acceso a la sección crítica.

Clase `java.util.concurrent.Semaphore`:

```
// Crea un objeto del tipo semáforo con el número máximo de hilos concurrentes
// permitidos en la sección crítica
Semaphore(int permits){}

// Solicita un permiso al semáforo. Si no hay se bloquea hasta que haya uno
// disponible. Lanza una excepción del tipo InterruptedException si la
// tarea es interrumpida mientras está en espera
public void acquire() throws InterruptedException{}

// Libera un permiso incrementando el número de permisos disponibles en
// una unidad o activando un hilo que esté bloqueado.
public void release(){}
```


Semáforos para exclusión mutua

Patrón:

```
class Recurso{
   private Semaphore sem = new Semaphore(1);

   //...
   public void metodo(){
      // Bloque de sentencias con la sección no crítica

      sem.acquire();  // Protocolo de entrada (antes de la sec. crítica)
      // Bloque de sentencias con la sección Crítica;
      sem.release();    // Protocolo de salida (tras la sec. crítica)

      // Bloque de sentencias con la sección no crítica;
   }
}
```


Ejemplo: Implementación del contador con un semáforo I

```
public class Contador{
   private double valor;
   private Semaphore sem;

   public Contador(){
      valor = 0;
      sem = new Semaphore(1);
   }

   public void incrementar(){
      try{
         sem.acquire();
         valor++;
         sem.release();
      }catch(InterruptedException ex){
         ex.printStackTrace();
      }
   }

   public void decrementar(){
      try{
         sem.acquire();
```

Ejemplo: Implementación del contador con un semáforo II

```
            valor--;
            sem.release();
        }catch(InterruptedException ex){
            ex.printStackTrace();
        }
    }

    public double getValor(){
        return valor;
    }
}
```


Índice

synchronized

La palabra reservada `synchronized` se puede utilizar para conseguir la exclusión mutua de tareas.

Se puede utilizar en la signatura del método:

```
public sychronized TipoDevuelto metodo(argumentos){
    // En este caso se usa el objeto referenciado con this
    // y por tanto sólo puede haber una tarea ejecutando
    // un método synchronized
}
```

o en bloques de código (dentro del código de un método):

```
...
synchronized(ref){
    // Solo una tarea puede estar ejecutando este
    // codigo u otro codigo que esté bloqueado
    // mediante el mismo objeto referenciado por ref
}
...
```


Exclusión mutua en el contador con métodos synchronized

```
public class Contador{
    private double valor;
    public Contador(){
        valor = 0;
    }

    public synchronized void incrementar(){
        valor++;
    }

    public synchronized void decrementar(){
        valor--;
    }
}
```

Exclusión mutua en el contador con bloques synchronized

```
public class Contador{
   private double valor;
   private Object lock;
   public Contador(){
      valor = 0;
      lock = new Object();
   }

   public void incrementar(){
      synchronized(lock){
         valor++;
      }
   }

   public void decrementar(){
      synchronized(lock){
         valor--;
      }
   }
}
```


Índice

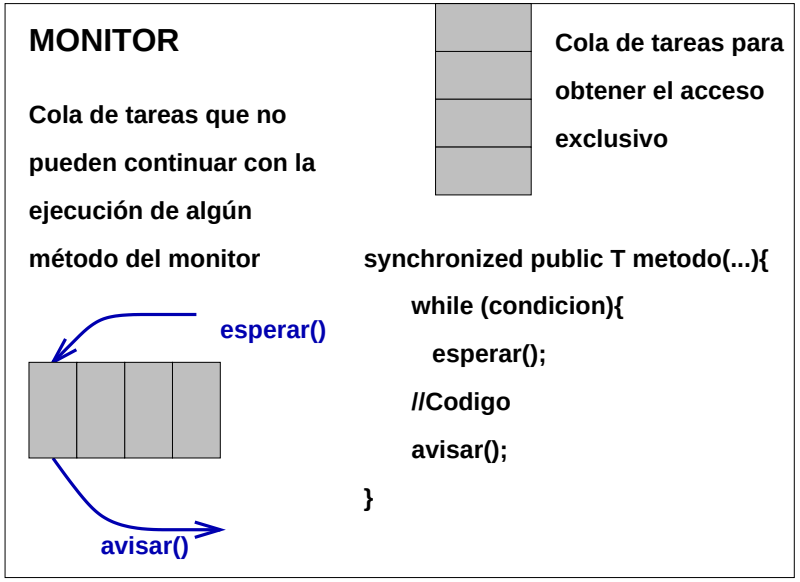
Monitores

Hoare, 1974. Monitor = Encapsula datos + mecanismos de sincronización.

La implementación que hace Java de los monitores incluye dos colas:

- Una cola de espera para conseguir entrar en las secciones críticas. Las tareas se colocan en esta cola cuando hay otra tarea ejecutando un método `synchronized`.
- Una cola de espera a recibir notificaciones. Las tareas se colocan en esta cola cuando no se dan las condiciones para que puedan continuar (no hay datos, el almacén está lleno, ...).

Operaciones sobre las tareas



Operaciones sobre las tareas

El monitor ofrece una serie de operaciones para gestionar las tareas:

- Parar a una tarea que no puede continuar. Esta tarea se coloca en la cola de espera y permanecerá en ella hasta que reciba una notificación.
- Notificar a una tarea para que compruebe si puede reanudar su ejecución. Si no puede continuar (por que no se dan todavía las condiciones) entonces vuelve a la cola.

Además, por el hecho de disponer de una sola cola de espera es necesario que ofrezca un método que permita que todas las tareas que están a la espera comprueben si pueden continuar con su ejecución.

Estos métodos están declarados en la clase `Object`, por tanto cualquier objeto puede ser un monitor.

Operaciones sobre las tareas

Los métodos son:

`wait()` Detiene a la tarea y la coloca en la cola de espera de notificaciones.

`notify()` Despierta a una tarea de la cola notificaciones y la pone en la cola de espera a obtener el acceso exclusivo

`notifyAll()` Despierta a todas las tareas que están el la cola de espera y las coloca en la cola de espera a obtener el acceso exclusivo.

Ejemplo I

A continuación se muestra el código de un almacén que tiene una capacidad fija y que tiene el siguiente comportamiento: si se llena bloquea a las tareas que quieran escribir y si se vacía bloquea a las tareas que quieran leer.

```
public class Almacen<T> {
   private T[] datos;
   private int cap;
   private int pos;
   private int numEl;

   public Almacen(int max) {
      cap = max;
      datos = (T[]) new Object[cap];
      pos = -1;
      numEl = 0;
   }

   synchronized public void almacenar(T dato) {
      while (numEl == cap)
         try {
            wait();
```


Ejemplo II

```
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      pos++;
      datos[pos] = dato;
      numEl++;
      notifyAll();
   }

   synchronized public T extraer() {
      T dato = null;
      while (numEl == 0)
         try {
            wait();
         } catch (InterruptedException ex) {
            ex.printStackTrace();
         }
      dato = datos[pos];
      pos--;
      numEl--;
      notifyAll();
      return dato;
```

Ejemplo III

```
    }
}
```

Parte III

Tema 3: Entrada/Salida y serialización

Tema 3
Entrada/Salida
Serialización

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Clasificar la categoría (entrada/salida, orientada a bytes/orientada a caracteres, bajo nivel/filtrada) a la que pertenece una clase.
2. Enumerar las clases principales de una categoría dada.
3. Describir para qué se utiliza el patrón decorador.
4. Aplicar el patrón decorador para construir flujos según una especificación dada.
5. Construir aplicaciones que usen correctamente las clases de entrada/salida.
6. Enumerar las interfaces que deben implementar los objetos que pueden ser serializados.
7. Describir los pasos necesarios para realizar la serialización y la recuperación.
8. Construir clases cuyas instancias puedan ser serializadas según las especificaciones dadas.
9. Comparar la serialización mediante Serializable y mediante Externalizable.
10. Elegir y justificar la mejor alternativa para serializar un objeto.
11. Construir aplicaciones que usen correctamente la serialización.

Índice

¿Por qué entrada/salida?

- **Almacenamiento/Recuperación de información**: una aplicación puede obtener o guardar datos en un fichero
- **Comunicación entre máquinas**: se realiza recibiendo o enviando información (usando la entrada/salida).

Estas dos situaciones son similares: los datos fluyen desde un lugar a otro (sólo cambia la fuente y el destino).

¿Por qué serialización?

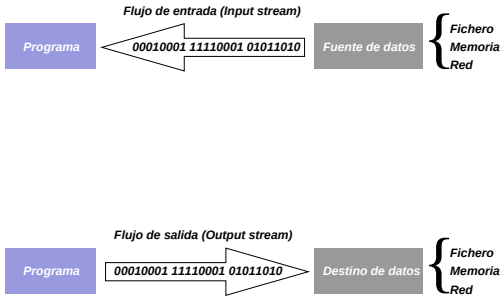
La serialización es el proceso de transformación de un objeto (o conjunto de objetos) en una secuencia de bits. Se puede aplicar en dos situaciones:

1. ¿Qué ocurre con un objeto cuando finaliza la máquina virtual? Se destruye. Mediante la serialización se puede conseguir la persistencia de un objeto guardándolo para ser recuperado posteriormente.
2. El objeto que está en la memoria de una máquina virtual se puede serializar para enviarlo a otra máquina virtual.

Índice

Flujos (streams)

- Un flujo es una abstracción de cualquier secuencia de datos que va desde una fuente (o productor de datos) hacia un destino (o consumidor de datos).
- Se puede interpretar como una serie de datos circulando a través de un canal que une a la fuente y al destino.



Flujos (streams)

Algunos ejemplos de fuentes o destinos de datos a los que se pueden asociar flujos de E/S:

- A la entrada estándar `System.in`, a la salida estándar `System.out`, a la salida de error estándar `System.err`.
- Fichero binarios o ficheros de texto.
- Conexiones de red
- A bloques de memoria (lectura o escritura de datos en la memoria).

Flujos (streams)

Una taxonomía (clasificación) de las clases de entrada/salida:

- Orientados a bytes
  - De bajo nivel
  - Filtrados
- Orientados a caracteres
  - De bajo nivel
  - Filtrados

Flujos (streams)

- Los de bajo nivel orientados a bytes ofrecen funcionalidad básica para lectura o escritura de bytes o arrays de bytes.
- Los de bajo nivel orientados a caracteres ofrecen funcionalidad básica para lectura o escritura de caracteres (con diferentes codificaciones).
- Los filtrados orientados a bytes ofrecen funcionalidad adicional (por ejemplo utilizando un buffer intermedio o permitiendo el trabajo con tipos primitivos) para la lectura o escritura orientada a bytes.
- Los filtrados orientados a caracteres ofrecen funcionalidad adicional para la lectura o escritura orientada a caracteres.

Índice

Entrada de bajo nivel orientada a bytes

- Los flujos de lectura extienden a la clase abstracta `InputStream`
- En esta clase se define el método abstracto: `int read() throws IOException` que lee el siguiente byte y lo devuelve como un `int` en el rango 0 a 255. Si se ha llegado al final del flujo devuelve un -1.
- El método `read()` bloquea la ejecución del hilo hasta que el siguiente dato esté disponible para ser leído, se llega al final del flujo o se lanza una excepción.
- Puesto que es abstracto, cada subclase debe proporcionar una implementación de este método.

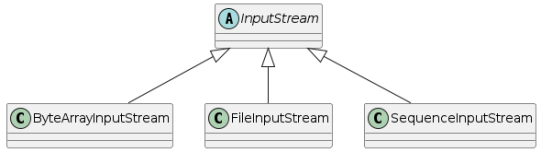
Algunas clases que representan flujos de entrada de bajo nivel en `java.io`

Hay varias clases que representan flujos de entrada de bajo nivel y que extienden a `InputStream`.

Cada clase representa un flujo asociado a la fuente donde están los datos. Ejemplos:

- `ByteArrayInputStream`: Permite leer bytes de un bloque de memoria.
- `FileInputStream`: Permite leer bytes de un fichero.
- `SequenceInputStream`: Permite leer bytes de datos desde dos o más flujos de bajo nivel, continuando con el siguiente flujo al finalizar la lectura de cada uno de ellos.

La jerarquía que siguen estas clases



Ejemplo: uso de flujo de entrada de bajo nivel orientado a bytes

```
import java.io.*;

public class InputDemo{
   public static void main(String[] args){
      int dato;
      try{
         // Se supone que se pasa el nombre del fichero a
         // mostrar como un argumento al programa
         InputStream entrada = new FileInputStream( args[0]);

         // Si el byte leído es -1 significa que no hay más datos
         while ( (dato=entrada.read())!=-1 )
            System.out.print( dato );

         System.out.println();

         entrada.close();
      }catch(IOException e){
         System.out.println("Error leyendo del fichero" + args[0]);
      }
   }
}
```

Esta no es la forma más eficiente de leer un fichero (**ya que es lento**) pero sirve para ilustrar la utilización de los flujos de entrada de bajo nivel.

Entrada filtrada orientada a bytes

- Los flujos de bajo nivel permiten leer bytes pero su flexibilidad es limitada.
- ¿Qué ocurre si deseamos leer un entero?, ¿o un double?, ¿o . . . ?.
- Los flujos filtrados añaden funcionalidad (usando un buffer, permitiendo la lectura de tipos primitivos, etc) al flujo pasado en su constructor.
- Los flujos filtrados de entrada extienden a la clase `FilterInputStream` que a su vez extiende a la clase `InputStream`.
- Para crear un flujo filtrado hay que disponer en última instancia de un flujo de bajo nivel. Por ejemplo: leer enteros **de memoria**, leer enteros **de un fichero**,. . .

Entrada filtrada orientada a bytes

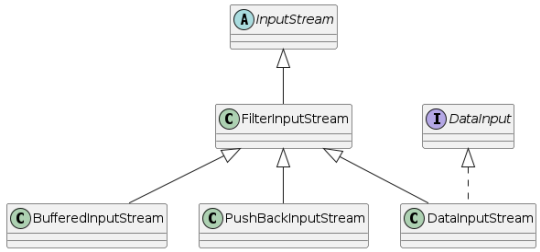
Ejemplos de clases que se definen en el paquete `java.io` para entrada filtrada orientada a bytes son:

- `BufferedInputStream`: Lee bloques de bytes y los almacena en un buffer para mejorar la eficiencia.
- `DataInputStream`: Lee tipos de datos primitivos, tales como `int`, `float`, `double`, etc.
- `LineNumberInputStream`: Mantiene un contador sobre el número de línea que se está leyendo.
- `PushBackInputStream`: Añade la capacidad de volver a colocar un byte (o array de bytes) ya leído otra vez en el flujo.

En otros paquetes `java.security`, `java.util`, `javax.crypto` o `java.util.zip` se definen más flujos de entrada filtrados.

Entrada filtrada orientada a bytes

La jerarquía que siguen estas clases es:



Entrada filtrada orientada a bytes

- Como se puede ver en el diagrama anterior, estas clases heredan de la clase `FilterInputStream`.
- El constructor de esta clase admite un flujo de entrada del tipo `InputStream` que es utilizado como fuente de datos (la lectura de bytes se realiza a través de éste).
- La clase `FilterInputStream` implementa los métodos abstractos de `InputStream` con versiones que pasan las peticiones al flujo de entrada pasado en el constructor.
- Las subclases de `FilterInputStream` pueden ocultar algunos de estos métodos y pueden proporcionar métodos o atributos adicionales.

Entrada filtrada orientada a bytes

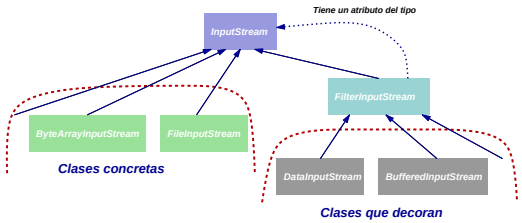
Esta solución se conoce como patrón de diseño **decorator** y pertenece al grupo de patrones estructurales.

¿Qué es un patrón?

- Un patrón de diseño es una solución que aparece frecuentemente en problemas de diseño de aplicaciones.
- Un patrón afronta un problema habitual de diseño que aparece en situaciones específicas y presenta una solución a él.
- Un patrón identifica abstracciones que están por encima del nivel de clases aisladas, instancias o de componentes.

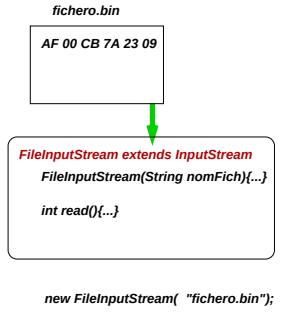
Esta patrón evita la proliferación de clases combinando los flujos para obtener la funcionalidad buscada sin necesidad de crear nuevas clases.

Entrada filtrada orientada a bytes

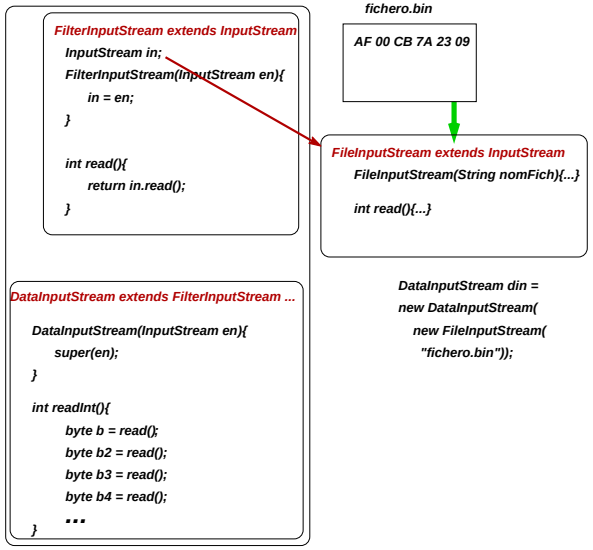


¿Qué se puede decorar?: cualquier clase que sea del tipo `InputStream`.

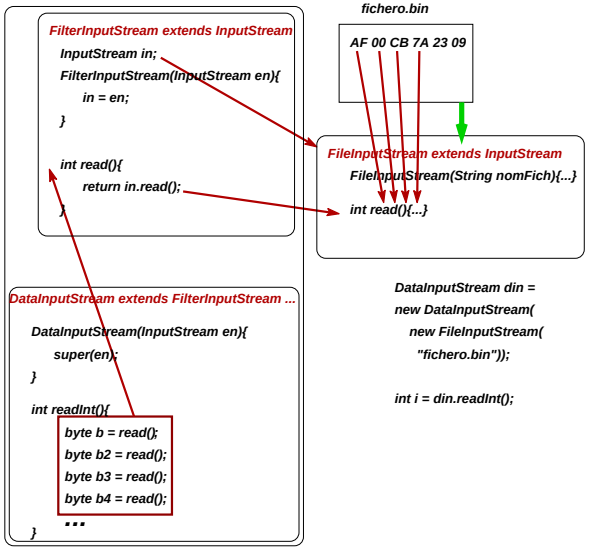
Entrada filtrada orientada a bytes



Entrada filtrada orientada a bytes



Entrada filtrada orientada a bytes



Ejemplo: uso de un flujo de entrada filtrado orientado a bytes

```
import java.io.*;

class DemoBufferedInputStream{
   public static void main(String[] args){
      try{
         BufferedInputStream in = new BufferedInputStream(new
↪  FileInputStream(args[0]));
         int dato;
         while ( (dato=in.read()) != -1)
            // Procesar dato
      }catch(IOException e){
         System.out.println("Error de lectura del fichero " + args[0]);
      }
   }
}
```

Esta lectura del fichero es mucho más eficiente que la versión que usa `FileInputStream`. Cada operación `read()` consume un `byte` del buffer y cuando se vacía este buffer, se llena automáticamente usando `read(byte[])`.

Índice

Salida orientada a bytes

- Los flujos de salida extienden a la clase abstracta `java.io.OutputStream`.
- Esta clase define el método abstracto:
  `public abstract void write(int b) throws IOException`
  que escribe un byte en este flujo de salida. El byte que se escribe se obtiene como los 8 bits de más bajo orden del argumento. Los restantes 24 bits de se desechan.
- Cada subclase debe proporcionar una implementación de este método.
- El paquete `java.io` proporciona unos cuantos flujos de salida y hay que saber cual es el apropiado para realizar la escritura.

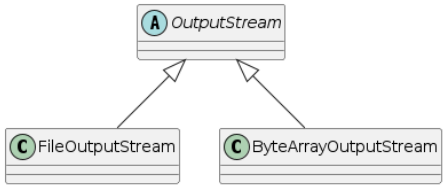
Salida de bajo nivel orientada a bytes

Ejemplos de clases que representan flujos de salida de bajo nivel:

- `ByteArrayOutputStream`: Escribe bytes en un bloque de memoria.
- `FileOutputStream`: Escribe bytes a un fichero.

Salida de bajo nivel orientada a bytes

La jerarquía que siguen estas clases de salida de bajo nivel orientadas a bytes es:



Ejemplo: uso de un flujo de salida de bajo nivel orientado a bytes

```
import java.io.*;

public class OutputDemo{
   public static void main(String[] args){
      int dato;
      try{
         // Se supone que se pasa el nombre del fichero a
         // crear como un argumento al programa
         OutputStream salida = new FileOutputStream(args[0]);

         // En este array están los datos a guardar en el fichero
         byte[] data = ...;

         for (int i=0; i<data.length; i++)
            salida.write(data[i]);

         salida.close();
      }catch(IOException e){
         System.out.println("Error escribiendo en el fichero" + args[0]);
      }
   }
}
```

Este código se proporciona para ilustrar el uso de un flujo de bajo nivel, pero no escribiríamos a un fichero de este modo (**es lento si hay que escribir muchos datos**).

Salida filtrada orientada a bytes

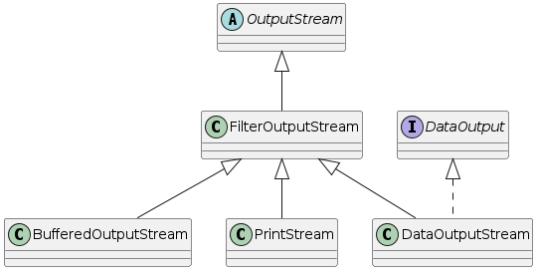
- Las clases de salida también utilizan el patrón decorador mediante la clase `FilterOutputStream` y sus subclases.
- El objetivo es el mismo que anteriormente: añadir una determinada funcionalidad.
- En el paquete `java.io` se declaran un conjunto de flujos filtrados.

Salida filtrada orientada a bytes

- `BufferedOutputStream`: Esta clase implementa un flujo de salida con buffer,de forma que una aplicación puede escribir bytes en el flujo de salida subyacente sin que necesariamente se produzcan llamadas para que se escriba cada byte (se almacenan en el buffer para ser escritos en bloque).
- `DataOutputStream`: Permite escribir datos primitivos en un flujo de salida.Los datos se pueden leer usando `DataInputStream`.
- `PrintStream`: Añade la capacidad de escribir representaciones de diversos datos de forma conveniente. Los métodos de esta clase no lanzan una excepción del tipo `IOException`, en su lugar establecen una bandera interna que puede ser consultada mediante el método `checkError()`. La salida estándar (a la que se puede acceder a través del atributo estático `out` de la clases =System) es de este tipo.

Salida filtrada orientada a bytes

La jerarquía que siguen estas clases filtradas orientadas a bytes es:



Ejemplo: uso de un flujo de salida filtrado orientado a bytes I

```
import java.io.*;

class DemoDataOutputInputStream{
   public static void main(String[] args){
      // Primero se escriben datos de diverso tipo a un fichero
      try{
         DataOutputStream out = new DataOutputStream(
                                    new FileOutputStream("borrame.bin"));
         out.writeChar('A');
         out.writeBoolean(true);
         out.writeInt(256);
         out.close();
      }catch(IOException e){};

      // A continuación se leen del fichero y se muestran
      try{
         DataInputStream in = new DataInputStream(
                                      new FileInputStream("borrame.bin"));
         // Obviamente hay que leerlos en el mismo orden que se han escrito
         System.out.println(in.readChar());
         System.out.println(in.readBoolean());
         System.out.println(in.readInt());
```

Ejemplo: uso de un flujo de salida filtrado orientado a bytes II

```
            in.close();
        }catch(IOException e){};
    }
}
```


Índice

Introducción

- Los flujos vistos anteriormente se pueden utilizar para leer y escribir texto además de bytes y tipos primitivos de datos.
- Sin embargo, para trabajar con texto es mejor utilizar las clases derivadas `Reader` de y `Writer`.
- Estas clases fueron introducidas a partir de la versión 1.1 del JDK para dar soporte a flujos que trabajan con diferentes codificaciones de caracteres.
- Estas clases tienen los mismos métodos que las clases orientadas a bytes salvo que los distintos métodos trabajan con 2 bytes en lugar de un byte.

Codificación de caracteres

- Un sistema de codificación es una tabla en la que se especifica el número correspondiente a cada carácter.
- Puede ocurrir que un determinado carácter no exista en una determinada codificación (por ejemplo US-ASCII no soporta los acentos).
- O que se use una codificación en la máquina que genera el texto y una codificación diferente en la máquina que lo muestra, existe el peligro de que los datos se corrompan al pasar de un sistema de codificación a otro.
- Unicode proporciona un único número a cada carácter independientemente de la plataforma, del programa o del lenguaje.

Codificación de caracteres

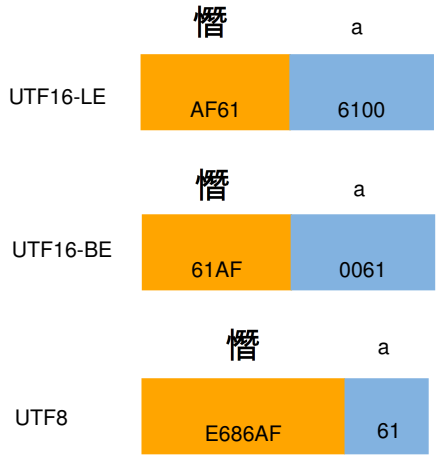
A continuación se describe el conjunto mínimo de codificaciones que debe soportar toda máquina virtual

- **US-ASCII**: código de 7 bits.
- **ISO-8859-X**: familia de codificaciones para soportar caracteres en otros lenguajes. ISO-8859-1 (Latin-1), ISO-8859-2 (Latin-2),...
- **UTF-16** utiliza un código de 16 bits para representar los caracteres.
- **UTF-8** codifica con un número variable de bytes (desde un byte hasta cuatro bytes). Esta codificación es eficiente si codifica documentos que utilizan caracteres US-ASCII ya que éstos se codifican con un único byte. UTF-8 es la codificación por defecto para XML y en los sistemas operativos actuales.

Algunas máquinas virtuales ofrecen soporte para otros conjuntos de caracteres (dependiendo del sistema operativo sobre el que se instale).

Codificación de caracteres

La siguiente ilustración muestra los bytes (en hexadecimal) de un fichero con dos caracteres para diferentes formatos.



Entrada de bajo nivel orientada a caracteres

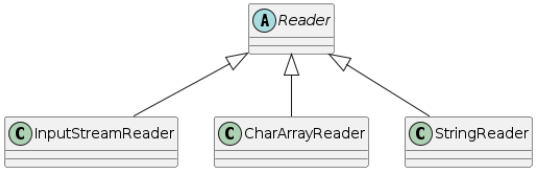
- La clase abstracta `java.io.Reader` es la superclase de las clases de entrada que trabajan con caracteres.
- Una clase que la extienda debe implementar los métodos abstractos `read(char[], int, int)` y `close()`.
- Tiene el método (entre otros) `int read() throws IOException` para leer un único carácter devuelto como un entero en el rango 0 a 65535 (0x0 - 0xFFFF) o -1 si se ha llegado al final del fichero.
- Este método bloquea la ejecución hasta que el siguiente carácter esté disponible, se lance una excepción o se alcance el final del fichero.

Entrada de bajo nivel orientada a caracteres

Las clases que extienden a `Reader` para la lectura de caracteres de bajo nivel en el paquete `java.io` son:

- `CharArrayReader` lee caracteres de un array de caracteres.
- `FileReader` lee caracteres de un fichero.
- `StringReader` lee caracteres de un `String`
- `InputStreamReader` recibe un flujo orientado a bytes como primer argumento y como segundo argumento la codificación de caracteres.

Entrada de bajo nivel orientada a caracteres



Ejemplo: uso de un flujo de bajo nivel orientado a caracteres

En este ejemplo se lee un fichero de caracteres asumiendo la codificación de caracteres por defecto:

```
import java.io.*;

class DemoFileReader{

   public static void main(String[] args){
      int c;

      try{
         FileReader fr = new FileReader(args[0]);

         while ((c=fr.read())!=-1)
            System.out.print((char)c);
         fr.close();
      }catch(IOException e){
         System.out.println("Error E/S");
         e.printStackTrace();
      }
   }
}
```


Ejemplo: uso de un flujo de bajo nivel orientado a caracteres

En este ejemplo se lee un fichero de caracteres especificando la codificación.

```
import java.io.*;

class DemoInputStreamReaderCodificacion{
   public static void main(String[] args){
      int c;

      try{
         InputStreamReader fr = new InputStreamReader(new
                                FileInputStream(args[0]),"UTF-16");
         while ((c=fr.read())!=-1)
            System.out.print((char)c);
         fr.close();
      }catch(IOException e){
         System.out.println("Error E/S");
         e.printStackTrace();
      }
   }
}
```

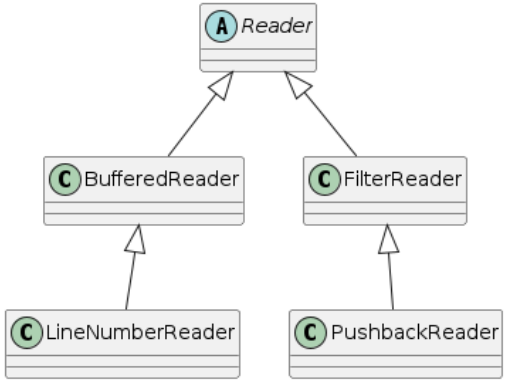

Entrada filtrada orientada a caracteres

Las clases que se ofrecen para entrada filtrada son:

- `BufferedReader` flujo de lectura que almacena los caracteres en un *buffer* para que la lectura sea eficiente.
- `PushbackReader` permite volver a poner en el flujo caracteres leídos.
- `LineNumberReader` lee caracteres de un fichero utilizando un *buffer* y permitiendo consultar el número de líneas leídas.

Entrada filtrada orientada a caracteres

La jerarquía que siguen estas clases es:



Índice

Salida de bajo nivel orientada a caracteres

- La clase de la que heredan las clases de salida que trabajan con caracteres es la clase `Writer` que está definida como abstracta.
- Una clase que la extienda debe implementar los métodos `write(char[], int, int)`, `flush()` y `close()`.
- Tiene el método (entre otros) `public write(int) throws IOException` para escribir un único carácter.
- El caracter a escribir está contenido en los 16 bits de bajo orden del valor entero proporcionado, los 16 bits de más alto orden son desechados.

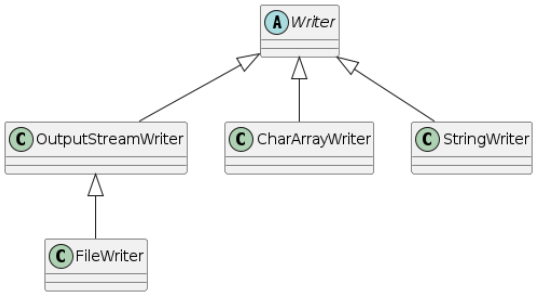
Salida de bajo nivel orientada a caracteres

Ejemplos de clases que extienden a `Writer` para la escritura de caracteres de bajo nivel en el paquete `java.io` son:

- `CharArrayWriter` Escribe en un array de caracteres.
- `FileWriter` Escribe en un fichero.
- `StringWriter` Escribe caracteres en un String.
- `OutputStreamWriter` recibe un flujo de salida orientado a bytes y permite especificar la codificación de caracteres.

Salida de bajo nivel orientada a caracteres

La jerarquía que siguen estas clases es:



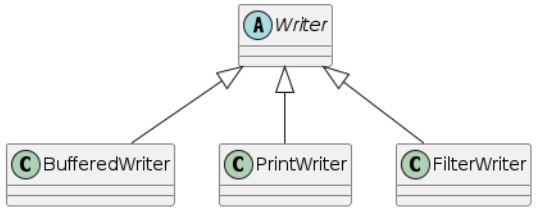
Salida filtrada orientada a caracteres

Las clases que se ofrecen para salida filtrada son:

- `BufferedWriter` flujo de salida que almacena los caracteres en un *buffer* para que la escritura sea eficiente.
- `PrintWriter` ofrece los métodos `print` y `println` que aceptan tipos primitivos y referencias y escribe a un flujo de salida orientado a caracteres.

Salida filtrada orientada a caracteres

La jerarquía que siguen estas clases es:



Ejemplo: uso de `PrintWriter` para generar un fichero de texto I

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

class Punto{
   private int x;
   private int y;
   Punto(int cx, int cy){
      x = cx;
      y = cy;
   }

   public String toString(){
      return x + ";" + y;
   }
}

class DemoPrintWriter{
   public static void main(String[] args){
      try{
         int n=100;
         List<Punto> lista = new LinkedList<Punto>();
         for (int i=0;i<n;i++)
```

Ejemplo: uso de `PrintWriter` para generar un fichero de texto II

```
            lista.add(new Punto(i,i+1));

         PrintWriter pw = new PrintWriter(new FileWriter("puntos.txt"),true);

         for (Punto p: lista)
            pw.println(p);

         pw.close();
      }catch(IOException e){}
   }
}
```


Ejemplo de lectura del fichero anterior usando `BufferedReader`

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

class DemoBufferedReader{
   public static void main(String[] args) throws Exception{
      BufferedReader br = new BufferedReader(new FileReader("puntos.txt"));

      String linea;
      List<Punto> lista = new LinkedList<Punto>();

      while ( (linea=br.readLine()) != null){
         // Se ha usado el punto y coma como delimitador
         String[] coordenadas = linea.split(";");
         lista.add(new Punto(Integer.parseInt(coordenadas[0]),
                             Integer.parseInt(coordenadas[1])));
      }
      br.close();

      System.out.println(lista);
   }
}
```


Índice

Introducción

- La serialización permite convertir cualquier objeto que implemente a la interfaz `Serializable` o la interfaz `Externalizable` en una secuencia de bits que puede ser utilizada posteriormente para reconstruir el objeto original.
- Esta secuencia de bits puede guardarse en un fichero o puede enviarse a otra máquina virtual (que puede estar ejecutándose en otro sistema operativo) para reconstruir el objeto en otro instante o en otra máquina virtual.
- La posibilidad de guardar un objeto de forma que pueda existir incluso cuando la aplicación haya finalizado se conoce como `persistencia`.

Introducción

- Los objetos mantienen referencias a otros objetos. Estos otros objetos deben ser también almacenados y recuperados con el fin de mantener las relaciones originales. Por supuesto, todos estos objetos deben ser serializables ya que de lo contrario se lanzará una excepción del tipo `NotSerializableException`.
- Para reconstruir un objeto (o conjunto de objetos) que ha sido serializado es necesario que la clase (o clases) (archivo .class) esté en el *classpath* (en caso contrario se lanzará una excepción del tipo `ClassNotFoundException`).
- La serialización se introdujo en Java para implementar la Invocación Remota de Métodos (RMI) que permite simplificar el desarrollo de aplicaciones distribuidas.

Índice

Flujos para serializar y deserializar

Pasos para realizar la serialización (con ObjectOutputStream)

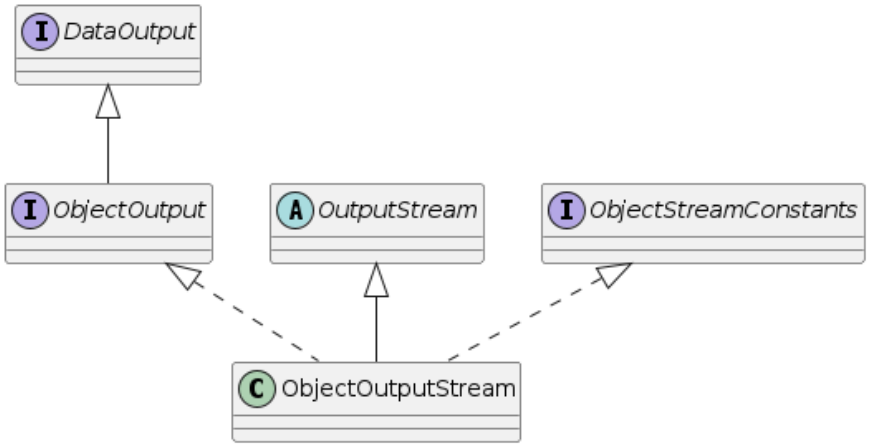
1. Se crea el flujo de salida a donde se van a enviar el objeto serializado.
2. Este flujo se pasa como argumento al constructor de `ObjectOutputStream`
3. Esta clase tiene (entre otros) el método `writeObject(Object o)` que serializa el objeto.

Pasos para realizar la recuperación (con ObjectInputStream)

1. Se crea el flujo de entrada de donde se va a leer el objeto serializado.
2. Este objeto se pasa como argumento al constructor de `ObjectInputStream`.
3. Esta clase tiene (entre otros) el método `Object readObject()` que permite recuperar el objeto.

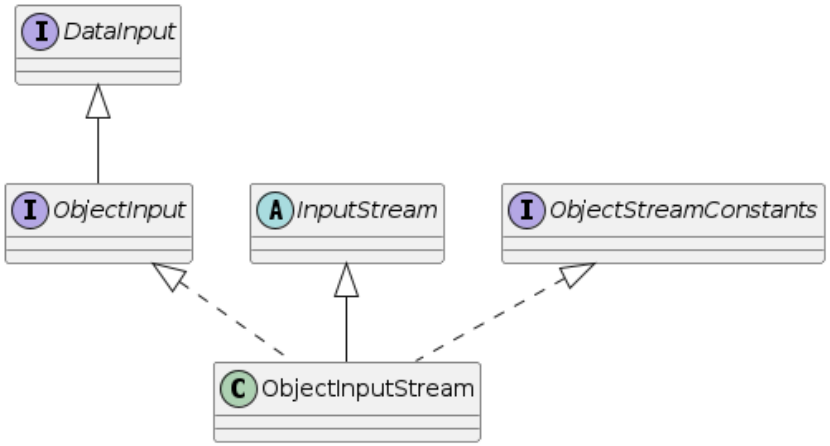
Flujos para serializar y deserializar

La jerarquía que siguen las clases para la serialización de objetos es:



Flujos para serializar y deserializar

La jerarquía que sigue la clase para la recuperación de objetos serializados es:



Índice

Serializable

- Una clase se convierte en serializable implementando la interface `Serializable`.
- Cualquier subclase de una clase que implemente a esta interface se convierte automáticamente en serializable.
- Esta interface no define ningún método ni ningún atributo y sirve únicamente para indicar que la clase es serializable.
- Si al recorrer el grafo de referencias se encuentra que un objeto no implementa la interface `Serializable` se se lanza una excepción del tipo `NotSerializableException`.
- Solo permite un control básico sobre los atributos a serializar: los atributos `transient` no se seralizan.

Serializable

- Una clase serializable debe declarar el atributo `serialVersionUID` del tipo `long` que sea `static` y `final`.
- Al realizar la deserialización se comprueba que el objeto y la clase tienen el mismo `serialVersionUID`.
- Si la clase cargada define un número de versión diferente del de la instancia que se intenta recuperar, entonces se produce una excepción `InvalidClassException`.

Ejemplo Serializable I

En este ejemplo hay 3 clases. Las clases `Punto` y `ListaPuntos` que implementan a `Serializable` y la clase `DemoSerializacion` que contiene el `main`.

En el `main` se crea un objeto del tipo `ListaPuntos` (que contiene objetos de tipo `Punto`) y se serializa a un fichero.

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

/** Clase que representa un punto 2d*/
class Punto implements Serializable{
   private int x;
   private int y;

   Punto(int cx, int cy){
      System.out.println("Creando el punto (" + cx + ", " + cy + ")");
      x = cx;
      y = cy;
   }
```


Ejemplo Serializable II

```
   public int getX(){
      return x;
   }

   public int getY(){
      return y;
   }
}

/** Crea y almacena en una lista objetos del tipo Punto */
class ListaPuntos implements Serializable{
   private LinkedList<Punto> lista = new LinkedList<Punto>();

   ListaPuntos(int nPuntos){
      System.out.println("Constructor de ListaPuntos");
      int x,y;

      for (int i=0; i<nPuntos; i++){
         x = (int) (Math.random()*10);
         y = (int) (Math.random()*10);
         lista.add(new Punto(x,y));
      }
   }
```


Ejemplo Serializable III

```
   public void muestraPuntos(){
      for (Punto p: lista)
         System.out.println("x = " + p.getX() + ", y = " + p.getY());
   }
}


public class DemoSerializacion{
   public static void main(String[] args){
      ListaPuntos lista = new ListaPuntos(5);

      try{
         ObjectOutputStream salida = new ObjectOutputStream(
                                    new FileOutputStream("objeto.bin"));

         salida.writeObject(lista);
         salida.close();
      }catch(Exception ex){
         System.err.println(ex.toString());
      }
```


Ejemplo Serializable IV

```
   }
}
```

A continuación se recupera el objeto del fichero y se le envían mensajes para comprobar que se ha reconstruido correctamente.

```
import java.io.*;

public class DemoRecuperacion{
   public static void main(String[] args){
      try{
         System.out.println("Recuperando el objeto...");

         ObjectInputStream entrada = new ObjectInputStream(
                                                                  new
↪  FileInputStream("objeto.bin"));

         // Downcast ya que readObject() devuelve una ref del tipo Object
         ListaPuntos lp = (ListaPuntos)entrada.readObject();

         entrada.close();
```

Ejemplo Serializable V

```
            lp.muestraPuntos();
        }catch(Exception ex){
            System.err.println(ex.toString());
        }
    }
}
```


Ejemplo Serializable

La ejecución del programa anterior produce el siguiente resultado:

```
Constructor de ListaPuntos
   Creando el punto (7, 4)
   Creando el punto (8, 7)
   Creando el punto (3, 2)
   Creando el punto (0, 3)
   Creando el punto (6, 4)
   Recuperando el objeto...
   x = 7, y = 4
   x = 8, y = 7
   x = 3, y = 2
   x = 0, y = 3
   x = 6, y = 4
```

Como puede observarse, durante la reconstrucción de los objetos no se llama a ninguno de los constructores, ésta se realiza únicamente a partir de los bytes almacenados.

Índice

Externalizable

- Si queremos tener control sobre cómo se serializa un objeto entonces hay que implementar a `Externalizable`.
- La interfaz `Externalizable` extiende a `Serializable` añadiendo dos métodos:

```
public void writeExternal(ObjectOutput out)
public void readExternal(ObjectInput in)
```

  que son llamados automáticamente durante la serialización y la recuperación.
- En estos métodos especificamos qué atributos se serializan/deserializan y cómo se realiza ese proceso de serialización y deserialización..

Ejemplo Externalizable I

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

class Player implements Externalizable{
   private String user;
   private int score;

   public Player(){
      System.out.println("Creando Player vacio");
   }

   Player(String u, int s){
      System.out.println("Creando Player (" + u + ", " + s + ")");
      user = u;
      score = s;
   }

   public void writeExternal(ObjectOutput out)
      throws IOException {
      System.out.println("Player.writeExternal");
      // Explicitamente indicamos cómo serializar los atributos
      out.writeObject(user+";"+score);
```


Ejemplo Externalizable II

```
   }

   public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
      System.out.println("Player.readExternal");
      // Explicitamente indicamos cómo deserializar los atributos
      String player = (String)in.readObject();
      String[] data = player.split(";");
      // Asignamos la información recuperada a los atributos
      user = data[0];
      score = Integer.parseInt(data[1]);
   }

   public String toString(){
     return "User: " + user + ", score: " + score;
   }
}

class DemoExternalizable{
   public static void main(String[] args)
      throws IOException, ClassNotFoundException {

      String[] users={"A","B","C"};
```


Ejemplo Externalizable III

```
      int[] scores={10,2,30};

      List<Player> lp = new LinkedList<Player>();
      for (int i=0; i<3; i++)
         lp.add(new Player(users[i],scores[i]));

      System.out.println("\nAlmacenando objeto");
      ObjectOutputStream o =
         new ObjectOutputStream(
                              new FileOutputStream("/tmp/objetos.out"));

      o.writeObject(lp);
      o.close();

      System.out.println("\nRecuperando objeto");
      ObjectInputStream in =
         new ObjectInputStream(
                             new FileInputStream("/tmp/objetos.out"));

      List<Player> lpr = (List<Player>)in.readObject();
      System.out.println(lpr);
```


Ejemplo Externalizable IV

```
   }
}
```

```
Creando Player (A, 10)
Creando Player (B, 2)
Creando Player (C, 30)

Almacenando objeto
Player.writeExternal
Player.writeExternal
Player.writeExternal

Recuperando objeto
Creando Player vacio
Player.readExternal
Creando Player vacio
Player.readExternal
Creando Player vacio
Player.readExternal
[User: A, score: 10, User: B, score: 2, User: C, score: 30]
```

Como puede observarse, (al contrario que al implementar a `Serializable`) al recuperar un objeto que ha sido *externalizado* se llama al constructor por defecto así que este debe ser accesible.

Índice

Para saber más

- Java I/O, Elliotte Rusty Harold. 1999 O'Reilly.
- Tutorial dedicado a entrada/salida:
  http://docs.oracle.com/javase/tutorial/essential/io/index.html

Parte IV

Tema 4: Programación en red (UDP, TCP y HTTP)

Tema 4
Programación en red

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Identificar la clase que sirve para representar direcciones IP
2. Identificar las clases que sirven para trabajar con el protocolo UDP (tanto *unicast* como *multicast*).
3. Describir cómo se usan estas clases en el servidor y en el cliente
4. Desarrollar aplicaciones que usen el protocolo UDP.
5. Identificar las clases que sirven para trabajar con el protocolo TCP.
6. Describir cómo se usan estas clases en el servidor y en el cliente
7. Desarrollar aplicaciones que usen el protocolo TCP.
8. Comparar las características que presenta el desarrollo de una aplicación con UDP y con TCP.
9. Identificar las clases que sirven para obtener y/o clocar recursos y que pueden trabajar con diferentes protocolos en la capa de aplicación.
10. Desarrollar aplicaciones que obtengan y/o coloquen recursos usando estas clases.
11. Identificar la clase que sirve para trabajar de forma específica con el protocolo HTTP.
12. Desarrollar aplicaciones que trabajen con el protocolo HTTP.

Índice

Índice

Direcciones IP

Una instancia de la clase `InetAddress` sirve para representar una dirección IP.

Esta clase define una serie de métodos estáticos para obtener instancias de este tipo:

```
public static InetAddress getLocalHost()
```

Devuelve la dirección de la máquina local.

```
public static InetAddress getLoopbackAddress()
```

Devuelve la dirección de loopback

Direcciones IP

```
public static InetAddress getByName(String host)throws
 UnknownHostException
```

Devuelve un objeto del tipo `InetAddress` a partir del nombre. El nombre puede ser un nombre de máquina, como por ejemplo `sweb.uv.es` o una representación como cadena de su dirección IP como por ejemplo `147.156.16.46`

```
public static InetAddress[] getAllByName(String host)throws
UnknownHostException
```

Dado un nombre de servidor, devuelve un vector con todas sus direcciones IP.

Direcciones IP

Una vez se tiene una instancia de este tipo se pueden usar algunos de sus métodos:

`public byte[] getAddress()`

Devuelve la dirección IP que representa en un vector de `byte`.

`public String getHostName()`

Devuelve el nombre de la máquina para la dirección que representa

`public boolean isReachable()`

Comprueba si se puede acceder a la dirección que representa

Direcciones IP

El siguiente código muestra un ejemplo de aplicación de esta clase:

```
import java.net.*;

public class DireccionesIP {
   public static void main(String[] args) {
      try{
         // Si se pasa un argumento obtenemos la direccion IP
         if (args.length>0){
            String host = args[0];
            InetAddress[] direcciones = InetAddress.getAllByName(host);

            for (int i=0;i<direcciones.length;i++){
               System.out.println(direcciones[i]);
            }
         }
         // Si no, obtenemos la direccion IP de la maquina local
         else{
            InetAddress direccion = InetAddress.getLocalHost();
            System.out.println(direccion);
         }
      }catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

```
(message-box (prin1 c))
```


Índice

NetworkInterface

Una instancia del tipo `NetworkInterface` representa a una interfaz de red.

La clase ofrece una serie de métodos estáticos para obtener instancias de este tipo:

```
// Permite obtener la interfaz de red correspondiente a una dirección IP
// de la máquina
public static NetworkInterface getByInetAddress(InetAddress addr)

// Permite obtener una interfaz de red a partir del nombre pasado
// como argumento (por ejemplo "eth0")
public static NetworkInterface getByName(String nombre)

// Permite obtener todas las interfaces de red de la máquina
public static Enumeration<NetworkInterfaces> getNetworkInterfaces()
```

NetworkInterface

Una vez tenemos un objeto de este tipo usando los métodos estáticos anteriores, podemos obtener información sobre la interfaz:

```
// Obtener su nombre
String getName()

// Obtener el MTU
int getMTU()

// Obtención de todas las direcciones IP asociadas a la interfaz
Enumeration<InetAddress> getInetAddresses()

// Comprobación de si es la interfaz lo
boolean isLoopback()

// Dirección MAC
byte[] getHardwareAddress

// ...
```


NetworkInterface I

```
import java.net.*;
import java.io.*;
import java.util.*;
import static java.lang.System.out;
import java.util.Enumeration;

class NICInfo {
   static void displayInterfaceInformation(NetworkInterface netint) throws
↪  Exception {
      out.printf("Name: %s\n", netint.getName());
      out.printf("Loopback: %b\n", netint.isLoopback());
      StringBuffer sb = new StringBuffer();
      for (byte b: netint.getHardwareAddress())
         sb.append(String.format("%02X ", b));
      out.printf("MAC: %s\n",sb.toString());

      // Obtención de todas las direcciones IP asociadas a la interfaz de red
      Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();

      // Bucle para mostrar las direcciones IP de la interface
      while (inetAddresses.hasMoreElements())
         out.printf("InetAddress:
↪  %s\n",inetAddresses.nextElement().toString());
```


NetworkInterface II

```
      out.printf("\n");
   }
   public static void main(String[] args) throws Exception {
      try {
         // Indicamos que queremos trabajar con IPV4
         System.setProperty("java.net.preferIPv4Stack" , "true");

         Enumeration<NetworkInterface> nics =
↪  NetworkInterface.getNetworkInterfaces();
         while (nics.hasMoreElements())
            displayInterfaceInformation(nics.nextElement());

      } catch(Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

```
(message-box (prin1 c))
```


Índice

Índice

UDP Unicast

Es un protocolo de la capa de transporte que las aplicaciones pueden utilizar para enviar paquetes de datos.

UDP soporta la especificación de un número de puerto por lo que puede ser utilizado para enviar datagramas a una aplicación o servicio concreto (que está escuchando en ese puerto).

- **Inconvenientes**: no garantiza la entrega de los paquetes ni que estos lleguen en el orden en el que han sido enviados.
- **Ventajas**: es eficiente, causa poca sobrecarga, es conveniente para aplicaciones en tiempo real y un socket UDP puede enviar y recibir datos de diferentes máquinas.

Se utiliza en aplicaciones en las que se puede tolerar la pérdida de algún paquete (*streaming* de audio y vídeo, *VoIP*, juegos).

UDP unicast

Java proporciona dos clases para el trabajo con el protocolo UDP:

- `java.net.DatagramPacket`
- `java.net.DatagramSocket`

¿Cómo se utilizan estas clases en una aplicación?

1. Crear un objeto del tipo `DatagramSocket` y
2. A través de éste socket enviar y/o recibir objetos del tipo `DatagramPacket`

UDP unicast: DatagramPacket

La clase **DatagramPacket** representa un paquete de datos que va a ser enviado o recibido utilizando el protocolo UDP. Los paquetes contienen la dirección IP, el número de puerto y los datos.

Hay dos situaciones en las que se debe crear un objeto del tipo `DatagramPacket`:

- para enviar un paquete a una máquina remota utilizando UDP (la dirección IP y el puerto se refieren a la máquina destino).
- para recibir un paquete enviado por una máquina remota utilizando UDP (la dirección IP y el número de puerto se refieren a la máquina que ha enviado el paquete).

UDP unicast: DatagramPacket

La clase `DatagramPacket` dispone de varios constructores (solo se muestran algunos):

`DatagramPacket(byte[] buf, int long)`

Crea un paquete UDP para **recibir** paquetes de longitud long, almacenando los datos en `buf`.

`DatagramPacket(byte[] buf, int long, InetAddress dir, int port)`

Construye un paquete UDP para **enviar** la información contenida en `buf` cuya longitud es `long` a la máquina y puertos especificados

UDP unicast

Algunos de los métodos que ofrece esta clase son:

`public InetAddress getAddress()`

Devuelve la dirección IP asociada al paquete UDP.

`public byte[] getData()`

Devuelve el buffer con los datos.

`public int getLength()`

Devuelve el número de bytes de datos almacenado en el datagrama. Este número puede ser menor que el tamaño del buffer.

UDP unicast: DatagramPacket

`public void setAddress(InetAddress dir)`

Sirve para asignar una dirección IP al paquete UDP.

`public setData(byte[] buffer)`

Asigna la información al *payload* del paquete UDP.

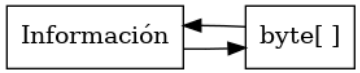
`public setLength(int long)`

Sirve para asignar la longitud real de los datos que contiene el paquete UDP. Este número deber ser menor o igual que el tamaño del buffer.

UDP unicast: DatagramPacket

Hemos visto que tenemos el método `setData(byte[] b)` para asignar la información al `DatagramPacket` y el método `byte[] getData()` para recuperarla.

Pero en nuestras aplicaciones trabajamos con otros tipos de datos, así que tendremos que usar un mecanismo que nos permita transforma la información a enviar en un `byte[]` y el mecanismo inverso que nos permita pasar los datos que llegan en un `byte[]` a la información que necesitamos:



Podemos usar los flujos `ByteArrayOutputStream` y `ByteArrayInputStream` para realizar estas transformaciones.

También se puede usar la clase `java.nio.ByteBuffer` para este propósito.

UDP unicast: DatagramSocket

La clase `DatagramSocket` representa un socket UDP para permitir el envío y recepción de paquetes UDP.

Una misma instancia de la clase `DatagramSocket` puede ser utilizada tanto para recibir paquetes como para enviarlos.

La clase `DatagramSocket` proporciona entre otros los siguientes constructores:

`DatagramSocket()`

Contruye un socket para el trabajo con paquetes UDP y le asigna un puerto libre en la máquina local.

`DatagramSocket(int puerto)`

Contruye un socket para el trabajo con paquetes UDP y le asigna el puerto especificado.

UDP unicast: DatagramSocket

Algunos de los métodos que ofrece esta clase son:

`public void send(DatagramPacket datagrama)throws IOException`

Envía el datagrama UDP.

`public void receive(DatagramPacket datagrama)throws IOException`

Espera a que llegue un paquete UDP y lo almacena en el `DatagramPacket` pasado como argumento. Una vez recibido, los datos sobre la dirección y puerto serán los correspondientes a la máquina origen. Si no se ha establecido el tiempo de espera, este método bloquea la ejecución del programa hasta que se reciba el paquete UDP.

`public void setReceiveBufferSize(int long)throws SocketException`

Establece el tamaño del buffer de recepción de paquetes UDP.

UDP unicast: DatagramSocket

`public void setSendBufferSize(int long)throws SocketException`

Establece el tamaño del buffer de envío de datagramas UDP.

`public void setSoTimeout(int duracion)`

Establece el tiempo de espera del socket. El valor es el número de milisegundos que el método `receive()` se esperará antes de lanzar una excepción del tipo `SocketTimeoutException`.

`public void close()`

Cierra el socket y libera el puerto.

UDP unicast

La estructura básica que debe tener una aplicación para la creación y el envío de paquetes UDP se muestra en el siguiente código.

```
String maquina = ...;
int puerto = ...;
int tamDatos = ...;

DatagramSocket socket = new DatagramSocket();
DatagramPacket paquete = new DatagramPacket(new
↪  byte[tamDatos],tamDatos,InetAddress.getByName(maquina),puerto);

boolean finalizar = false;

while(!finalizar){
   // Asignar los datos al buffer del paquete UDP
   //...
   socket.send(paquete);
   //...
}
socket.close();
```

UDP unicast

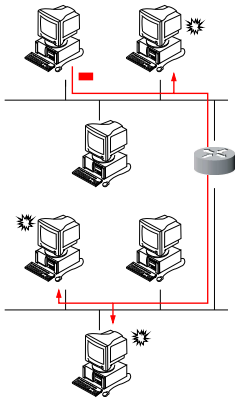
La estructura básica que debe tener una aplicación para la recepción de paquetes UDP se muestra en el siguiente código.

```
int puerto = ...;
int tamDatos = ...;
DatagramSocket socket = new DatagramSocket(puerto);
DatagramPacket paquete = new DatagramPacket(new byte[tamDatos],tamDatos);

boolean finalizar = false;

while(!finalizar){
   socket.receive(paquete);

   // Extraer los datos del paquete UDP y procesarlos
   // ...
}

socket.close();
```


Índice

UDP multicast

Multicast es similar a UDP, salvo que en lugar de enviar el paquete a un único destinatario, éste es enviado a múltiples receptores que hayan manifestado su interés.



https://es.wikipedia.org/wiki/Internet_Group_Management_Protocol

UDP multicast

- Un grupo multicast se especifica mediante una dirección IP de tipo D y un número de puerto.
- Las direcciones IP de tipo D están en el rango desde 224.0.0.0 hasta 239.255.255.255.
- Para enviar un paquete a un grupo multicast, el cliente simplemente envía el paquete en la red con la dirección IP y puerto apropiados.

UDP Multicast

- Los paquetes multicast incluyen un campo TTL (*Time To Live*) que limita la propagación de los paquetes a través de internet.
- Cada vez que un paquete es propagado a través de router o un túnel, se decrementa el TTL. Por tanto un paquete con valor pequeño de TTL será repartido en redes *cercanas*.

TTL	Restricciones
1	No se propaga a través del router
<=64	No se propaga a través de túneles
>64	No se aplica ninguna restricción

UDP multicast: MulticastSocket

La clase que soporta multicast es `MulticastSocket` y extiende a la clase `DatagramSocket`.

Ofrece los siguientes constructores:

```
MulticastSocket()
```

Crea una instancia que escucha y transmite a través de un puerto UDP libre.

```
MulticastSocket(int puerto)
```

Crea una instancia que escucha y transmite a través de un puerto UDP especificado.

UDP multicast: MulticastSocket

Y ofrece los siguientes métodos:

```
void joinGroup(InetAddress grupo)throws IOException
```

Anuncia el interés en un grupo multicast (los datagramas se enviarán a esa dirección y los datagramas dirigidos a esa dirección serán recibidos).

```
void leaveGroup(InetAddress grupo)throws IOException
```

Anuncia que se deja un grupo multicast.

UDP multicast: MulticastSocket

```
void setTimeToLive(int ttl)throws IOException
```

```
int getTimeToLive()throws IOException
```

Métodos para establecer o consultar el TTL (comentado anteriormente).

```
void send(DatagramPacket p)throws IOException
```

```
void receive(DatagramPacket p)throws IOException
```

Estos dos métodos los hereda de la clase `DatagramSocket`

UDP multicast

Ejemplo: envío de un paquete UDP multicast.

```
//Dirección IP tipo D y puerto para realizar multicast
InetAddress multicastGroup = InetAddress.getByName("239.1.2.3");
int puerto = 6789;

MulticastSocket ms = new MulticastSocket(puerto);

ms.joinGroup(multicastGroup);

String mensaje = "Hola, este es un mensaje de prueba";

DatagramPacket dp = new DatagramPacket(mensaje.getBytes(), mensaje.length(),
↪  multicastGroup, puerto);

ms.setTimeToLive(64);

ms.send(dp);
```


UDP multicast

Ejemplo: recepción de un paquete UDP multicast.

```
//Dirección IP tipo D y puerto para realizar multicast
InetAddress multicastGroup = InetAddress.getByName("239.1.2.3");
int puerto = 6789;

MulticastSocket ms = new MulticastSocket(puerto);
ms.joinGroup(multicastGroup);
//...

// Obtención de la respuesta

byte[] buf = new byte[2000];
DatagramPacket resp = new DatagramPacket(buf, buf.length);

ms.receive(resp);
// Obtención y procesado del payload
//...

ms.leaveGroup();
ms.close();
```


Índice

TCP

Es un protocolo de la capa de transporte que las aplicaciones pueden utilizar para enviar paquetes de datos y que garantiza la entrega en el mismo orden en el que se han enviado.

Desde el punto de vista de la aplicación, TCP transfiere un flujo continuo de bytes (por lo que se pueden utilizar los flujos de Entrada/Salida) a través de Internet. La aplicación no ha de preocuparse de dividir los datos en bloques.

El coste de la garantía de entrega y de la ordenación de los datos provoca una reducción del rendimiento en la red.

Java proporciona dos clases para el trabajo con el protocolo TCP:

- `java.net.Socket`: para crear el cliente (que se conectará a un determinado servicio: máquina y puerto).
- `java.net.ServerSocket`: para crear el servidor (servicio asociado a un puerto).

TCP: Socket

La clase Socket representa un canal de comunicación entre dos puertos de comunicación TCP que pertenecen a una o dos máquinas. Un socket TCP no puede comunicar más de dos máquinas.

Si se requiere esta funcionalidad el cliente debe establecer múltiples conexiones, una por cada máquina.

Esta clase ofrece diversos constructores entre los que cabe destacar:

```
Socket(InetAddress dir, int puerto)
```

Crea un socket conectado a la IP y puerto especificados.

```
Socket(InetAddress dir, int port,InetAddress localdir, int
localPort)
```

Crea un socket conectado la IP y puerto especificados y es asociado a la IP local y al puerto proporcionados.

TCP: Socket

```
Socket(String maquina, int puerto)
```

Crea un socket conectado la máquina y puerto especificados.

```
Socket(String maquina, int port,InetAddress local, int localPort)
```

Crea un socket conectado la maquina y puerto especificados y es asociado a la IP local y al puerto proporcionados.

Una vez que se ha creado el socket las operaciones habituales que se pueden realizar con el objeto son: **leer datos**, **escribir datos** , y al finalizar **cerrar la conexión**.

TCP: Socket

Algunos de los métodos que ofrece esta clase.

```
public OutputStream getOutputStream()throws IOException
```

Devuelve un flujo de salida que se puede utilizar para escribir datos que se envían a la aplicación a la que está conectado este socket.

```
public InputStream getInputStream()throws IOException
```

Devuelve un flujo de entrada que se puede utilizar para leer datos de se reciben de la aplicación a la que está conectado este socket.

TCP: Socket

```
public void setReceiveBufferSize(int tam)throws SocketException
```

```
public int getReceiveBufferSize()
```

Asigna o consulta el valor de `SO_RCVBUF`. Esta opción es utilizada para establecer el tamaño del buffer de red para la entrada.

La llamada al método que establece el tamaño del buffer se debe realizar realizar antes de conectar el socket.

```
public void setSendBufferSize(int tam)throws SocketException
```

```
public int getSendBufferSize()
```

Asigna o consulta el valor de `SO_SNDBUF`. Esta opción es utilizada para establecer el tamaño del buffer de red para la salida.

TCP: Socket

```
public void setSoTimeout(int duracion)throws SocketException
```

Establece el valor de la opción `SO_TIMEOUT` que controla el tiempo en milisegundos que bloquearán las operaciones de lectura. Un valor de cero bloquea de forma indefinida, en otro caso al transcurrir el tiempo especificado y no recibir datos, se lanza una excepción del tipo `SocketTimeoutException`.

```
public void setTcpNoDelay(boolean flag)throws SocketException
```

Establece el valor de la opción `TCP_NODELAY` que determina si se deben acumular secuencias de pequeños mensajes en paquetes TCP más grandes antes de ser enviados, impidiendo de este modo el envío de grandes cantidades de paquetes de pequeño tamaño.

TCP: Socket

```
public void shutdownInput()throws IOException
```

Cierra el flujo de entrada de el socket, de forma que las operaciones de lectura devolverán el marcador de final de fichero.

```
public void shutdownOutput()throws IOException
```

Cierra el flujo de salida de el socket, de forma que las operaciones de escritura lanzarán una excepción del tipo `IOException`.

```
public void close()throws IOException
```

Cierra el socket. Una vez cerrado, no se puede volver a conectar y por tanto se necesitaría crear un nuevo socket.

TCP: ServerSocket

En cuanto a la clase `ServerSocket` el constructor más sencillo es:

```
ServerSocket(int puerto)throws IOException
```

Crea un socket servidor asociado al puerto que se pasa como argumento de forma que los clientes puedan localizar el servicio TCP que ofrece.

Si el puerto ya está asociado se lanza una excepción.

TCP: ServerSocket

Una vez que se tiene un objeto de este tipo, se puede obtener un `Socket` para realizar la conexión y comunicación entre el cliente y el servidor mediante el método:

```
public Socket accept()throws IOException
```

Espera a que el socket cliente realice una petición de conexión y devuelve el socket correspondiente. Es una operación que bloquea hasta que no se produzca una conexión (a no ser que se haya establecido la opción `SO_TIMEOUT`

TCP: Servidor que atiende de forma concurrente a múltiples clientes

Si queremos tratar de forma simultánea la conexión de múltiples clientes, podemos tratar la conexión de cada cliente en un hilo.

```
class TrataConexion implements Runnable{
   private Socket s;
   TrataConexion(Socket s){
      this.s=s;
   }
   public void run(){
      // Obtención de flujos de E/S y comunicación
   }
}
```


TCP: Servidor que atiende de forma concurrente a múltiples clientes

En el servidor tendremos un bucle infinito aceptando las conexiones y creando una instancia de `TrataConexion` por cliente:

```
// En este código faltaría el tratamiento de las excepciones
class Server{
   public static void main(String[] args){
      SeverSocket s = new ServerSocket(9000);
      while (true){
         Socket canal = s.accept();
         new Thread(new TrataConexion(canal)).start();
      }
   }
}
```


Índice

Índice

URL

Esta clase sirve para representar uno de los tipos más usados de dirección en Internet: el *Uniform Resource Locator*.

Las direcciones URL pueden referirse a ficheros, sitios Web, sitios FTP, direcciones de correo electrónico, etc.

HTTP	http://maquina:puerto/recurso#referencia
Ejemplo	http://java.sun.com:80/downloads/ea/index.html#j2se
FTP	ftp://usuario:password@maquina:puerto/recurso
Ejemplo	ftp://anonymous:anonymous@ftp.sunite.dk:21/pub
MAIL	mailto:dirección
Ejemplo	mailto:foo@uv.es
ARCHIVOS	file://path_recurso
Ejemplo	c:/programas

URL

La clase `URL` dispone de varios constructores entre los que cabe destacar:

```
URL(String cadena)throws MalformedURLException
```

Crea un objeto URL a partir de la dirección pasada en forma de String.

```
URL(String proto, String host, String rec)throws
MalformedURLException
```

Crea un objeto URL a partir del protocolo, la máquina y el recurso pasados en forma de String.

URL

```
URL(String proto, String host, int puerto, String rec)throws
MalformedURLException
```

Crea un objeto URL a partir del protocolo, la máquina y el recurso pasados en forma de String, y el puerto pasado como un entero.

```
URL(URL contexto, String relativo)throws MalformedURLException
```

Crea un objeto URL a partir una ruta relativa a la URL pasada como primer argumento.

URL

Algunos de los **métodos** de los que dispone esta clase son:

```
String getProtocol()
```

Devuelve el protocolo asociado al objeto URL.

```
String getHost()
```

Devuelve la máquina asociada al objeto URL.

```
int getPort()
```

Devuelve el puerto asociado al objeto URL

URL

`InputStream openStream()` **throws** `IOException`

Devuelve un objeto del tipo `InputStream` para leer lo que nos devuelva el recurso que representa la URL.

Sin embargo, si lo queremos es escribir hacia la URL (por ejemplo para subir un fichero si es el protocolo TCP) no es suficiente con la clase `URL` y necesitamos un objeto del tipo `URLConnection`.

`URLConnection openConnection()` **throws** `IOException`

Devuelve un objeto `URLConnection` que representa una conexión con el recurso remoto al que hace referencia el objeto URL. Se utiliza cuando necesitamos controlar parámetros de la petición y escribir hacia el recurso.

Índice

URLConnection

La clase `URLConnection` es la superclase abstracta de todas las clases que representan la comunicación entre la aplicación y una URL.

Un objeto de este tipo se puede utilizar para establecer parámetros de la petición, leer desde y para escribir hacia el recurso al que hace referencia el objeto URL.

En general la creación de una conexión implica los siguientes pasos:

1. Se obtiene un objeto `URLConnection` utilizando el método `openConnection()` de un objeto .
2. Se establecen los parámetros y las propiedades de la petición.
3. Se establece la conexión utilizando el método `connect()`
4. Se puede obtener parámetros de la respuesta, obtener el recurso remoto o escribir hacia el recurso que representa esta URL.

URLConnection

Algunos de los métodos de esta clase son:

void `setRequestProperty(String key,String value)`

Establece una propiedad de la petición. Por ejemplo (válido para el protocolo HTTP):

```
conn.setRequestProperty ("User-agent","Mozilla/5.0 (X11; Linux x86_64)...");
```

void `connect()` **throws** `IOException`

Establece una conexión entre la aplicación (el cliente) y el recurso (servidor) enviando las propiedades de la petición que se hayan establecido con el método anterior.

URLConnection

```
void setDoOutput(boolean b)
```

```
void setDoInput(boolean b)
```

Métodos que sirven para establecer si se va a utilizar esta `URLConnection` para salida y/o entrada de datos.

```
OutputStream getOutputStream()throws IOException
```

Devuelve un objeto `OutputStream` para escribir datos hacia el servidor (en el caso de FTP podría ser un fichero que subimos al servidor, en el caso de HTTP sería el cuerpo de la petición).

URLConnection

```
int getContentLength()
```

Obtiene de la respuesta el valor del campo de cabecera *content-length* o -1 si no está definido.

```
String getContentType()
```

Devuelve de la respuesta el valor del campo de cabecera *content-type* o null si no está definido.

```
String getHeaderField(String campo)
```

Devuelve de la respuesta el valor del campo de cabecera especificado o `null` si no está definido.

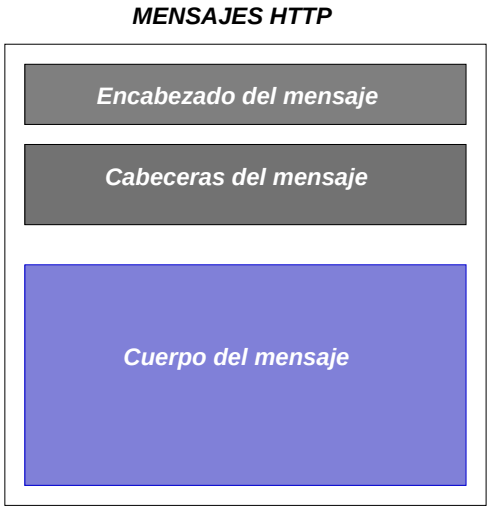
URLConnection

```
InputStream getInputStream()throws IOException
```

Devuelve un objeto `InputStream` para leer la respuesta. En el caso de HTTP sería el cuerpo de la respuesta HTTP que se envía desde el servidor.

Índice

HTTP: estructura de los mensajes



HTTP: mensajes de petición

Ejemplos de mensajes de petición: GET, POST, HEAD, DELETE, PUT.

- `GET` codifica los parámetros de la petición en la URL. El cuerpo del mensaje está vacío. Creado para peticiones que no modifican el estado del servidor.
- `POST` codifica los parámetros de la petición en el cuerpo del mensaje. En mensajes *multipart* es posible enviar ficheros al servidor.
- `HEAD` solicita las cabeceras del mensaje de respuesta.
- `PUT` para reemplazar un recurso del servidor.
- `DELETE` para borrar un recurso del servidor.

HTTP: Códigos de respuesta

- **1xx**: Informan al navegador
- **2xx**: Indican que la respuesta es correcta (200: respuesta correcta).
- **3xx**: Indican redirección (300 y 301: Indican el que navegador debe ir a otra página).
- **4xx**: Indican problemas en la petición (400: error en la petición, 401: no autorizado, 404: no encontrado)
- **5xx**: Indican errores en el servidor (500: error interno en el servidor, 501: no implementado).

HTTP: ejemplos de campos de cabecera

- `Content-Type` (petición y respuesta): describe el contenido del cuerpo del mensaje
- `Date` (petición y respuesta): fecha y hora en la que se ha generado el mensaje.
- `User-Agent` (petición): información sobre la aplicación que realiza la petición.
- `Set-Cookie` (respuesta): identificador de la sesión enviada por el servidor.
- `Cookie` (petición): identificador de la sesión enviada por el cliente.
- `WWW-Authenticate` (respuesta): el servidor informa al cliente de que se necesita usuario y contraseña para acceder al recurso.
- `Authentication` (petición): usuario y contraseña enviados por el cliente.

HttpURLConnection

Es una subclase de `URLConnection` que soporta características propias del protocolo HTTP.

Esta clase no tiene constructor, la forma de obtener una referencia a un objeto de este tipo es:

```
URL direccion = new URL("http://maquina:puerto/recurso");
URLConnection conexion = url.openConnection();

// Comprobamos su tipo
if (conexion instanceof HttpURLConnection ){

   // Se realiza un cast
   HttpURLConnection conexionHTTP = (HttpURLConnection) conexion;

   // Resto del codigo
}
```


HttpURLConnection

Algunos de los métodos que define esta clase (además de los de `URLConnection`) son:

`void disconnect()`

Si la conexión con el servidor está activa se cierra

`void setRequestMethod(String metodo)throws ProtocolException`

Establece el método de petición que se utilizará en esta conexión (`POST`, `GET`, `HEAD`,...). Debe ser utilizado antes de llamar a `connect()`.

`boolean usingProxy()`

Indica si se está utilizando un servidor *proxy* para la conexión.

HttpURLConnection

`int getResponseCode()`

Obtiene el código de estado del mensaje de respuesta HTTP. Por ejemplo para las siguientes líneas de estado:

```
HTTP/1.0 200 OK
HTTP/1.0 401 Unauthorized
```

Se obtendría 200 y 401 respectivamente.

Devuelve -1 si no se puede obtener el código a partir de la respuesta.

HTTP Client

HTTP Client de Apache es una librería bastante conocida que se usa para realizar peticiones desde Java a servidores HTTP.

http://hc.apache.org/httpcomponents-client-ga/index.html

https://mvnrepository.com/artifact/org.apache.httpcomponents.client5/httpclient5/5.2.1

Parte V

Tema 5: Programación distribuida (RMI)

Tema 5
Middleware, RMI

Juan Gutiérrez

Programación
GIT
Universitat de València

Outline

Índice

Objetivos

1. Explicar qué papel juega el middleware y qué ofrece.
2. Identificar la interfaz que debe extender la interfaz que define los servicios.
3. Identificar la clase que deben extender los objetos remotos.
4. Explicar la función del *stub*, dónde se crea, dónde se almacena y dónde se usa.
5. Identificar la clase que sirve para registrar en el registro RMI el stub y para obtenerlo desde el cliente.
6. Desarrollar el código correspondiente al servidor RMI.
7. Ejecutar el registro RMI
8. Desarrollar el código correspondiente al cliente y al servidor RMI.

Índice

Motivación

En uno de los boletines vimos que podíamos hacer RPC sobre TCP.

Supongamos que el servicio remoto ofrece las siguientes operaciones:

```
public interface ServicioRemoto{
   public void addVM(String host, VM vm);
   public void List<VM> getMVs(String host);
}
```

Para invocar al método `addVM` desde el cliente sobre TCP deberíamos:

1. Establecer una conexión TCP con el servidor
2. Obtener flujo de salida que permita serializar
3. Escribir qué método (de los dos que ofrece el servidor) estamos invocando
4. Escribir los dos argumentos
5. Cerrar los flujos
6. Cerrar la conexión socket

Motivación

```
public interface ServicioRemoto{
   public void addVM(String host, VM vm);
   public void List<VM> getMVs(String host);
}
```

Para invocar al método `getVMs` desde el cliente sobre TCP deberíamos:

1. Establecer una conexión TCP con el servidor
2. Obtener flujo de salida que permita serializar
3. Escribir qué método (de los dos que ofrece el servidor) estamos invocando
4. Escribir el argumento
5. Leer el resultado que envía el servidor
6. Cerrar los flujos
7. Cerrar la conexión socket

Motivación

Por supuesto, también tendremos que escribir el código del lado del servidor que:

1. Crea un `ServerSocket`
2. Acepta conexiones de clientes
   1. Lee el método invocado
   2. En función del método lee los argumentos
   3. **Realiza la lógica de negocio de la aplicación**
   4. En función del método escribe el resultado
   5. Cierra los flujos
   6. Cierra el socket

De todos estos pasos, lo que tiene valor para la aplicación es el únicamente el punto 3.

Motivación

Por tanto, si tenemos que escribir otra aplicación distribuida con otros métodos, tendremos que volver a escribir y adaptar todo este código de bajo nivel.

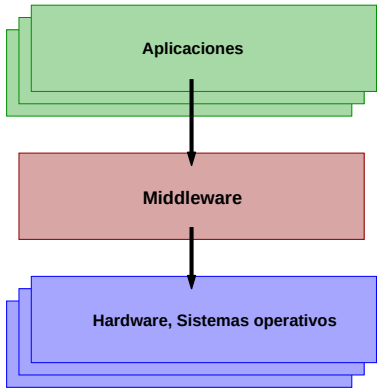
¿A qué problemas nos enfrentaremos?

- Ciclo de desarrollo lento (hay que escribir código de bajo nivel: conexiones, entrada/salida, definición del protocolo sobre TCP o HTTP, etc.) y propenso a errores.
- El código estará muy vinculado a un protocolo y migrar a otro (por ejemplo sobre HTTP) requiere modificar mucho código.
- No es posible mezclar componentes desarrollados en diferentes lenguajes de programación si usamos características propias de un lenguaje (por ejemplo la serialización en Java).
- No hay servicios de alto nivel (seguridad, transacciones, balanceo de carga, etc.) disponibles y por lo tanto hay que desarrollarlos para cada aplicación.

Middleware

Estos problemas motivaron el desarrollo de *middleware* y de arquitecturas basadas en *middleware*.

El *middleware* es una capa software que está entre las aplicaciones y los sistemas operativos, pilas de protocolos de red y hardware.



Middleware

La función principal es la de ofrecer una capa de abstracción que pueden usar las aplicaciones.

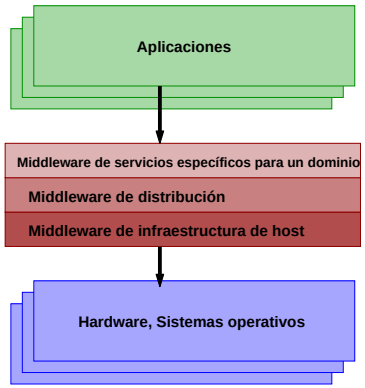
Ejemplos de *middleware*:

- Remote Procedure Call (procedimientos remotos: JAX-RPC)
- Object-Oriented Middleware (orientado a objetos: RMI, CORBA)
- Message-Oriented Middleware (orientado a mensajes: RabbitMQ, JMS)
- Event-Based Middleware (orientado a eventos y asíncrono: Vert.x)

Índice

Capas

Al igual que la pila de protocolos de red se descompone en múltiples capas (física, enlace de datos, red, transporte, sesión, presentación y aplicación) el *middleware* orientado a la computación con objetos distribuidos se puede descomponer en diferentes capas:



Capa de infraestructura de host

Capa middleware de infraestructura de host

Abstraen las particularidades de sistemas operativos y ayudan a evitar aspectos que son de bajo nivel, susceptibles de errores y no portables como por ejemplo la creación se *sockets* o hilos que son dependientes del sistema operativo. Por ejemplo la **máquina virtual** de Java proporciona una plataforma sobre la que ejecutar aplicaciones que abstrae de las diferencias entre sistemas operativos (es la máquina virtual la que se encarga de comunicarse con el sistema operativo).

Capa de middleware de distribución

Capa middleware de distribución

Define modelos de programación distribuida de alto nivel que extienden las capacidades de programación en red encapsuladas por el *middleware* de infraestructura de host. El *middleware* de distribución permite el desarrollo de aplicaciones distribuidas sin necesidad de escribir código de bajo nivel abstrayendo aspectos como: localización del servidor, lenguajes, sistemas operativos, protocolos de comunicación y hardware.

Ejemplos

- *Remote Method Invocation (RMI)*: que permite crear aplicaciones distribuidas desarrolladas en el lenguaje Java.
- *gRPC*: declaración del servicio y de los mensajes que se intercambian usando un lenguaje (*protobuf*) independiente del lenguaje de programación. El framework permite generar el código de comunicación entre el cliente y el servidor para facilitar el desarrollo de aplicaciones distribuidas.
- *Common Object Request Broker Architecture (CORBA)*: es un estándar abierto que permite a los objetos comunicarse por red independientemente del lenguaje en el que estén desarrollados o de la plataforma en la que están ejecutándose.
- *Distributed Component Object Model (DCOM)*: es una tecnología propietaria de Microsoft para desarrollar componentes software distribuidos que se comunican entre sí.

Capa middleware de servicios específicos

Capa middleware de servicios específicos para un dominio

Se ajustan a los requisitos de dominios concretos como telecomunicaciones, comercio electrónico, salud, automatización de procesos o industria aeroespacial. A diferencia de las otras dos capas, que ofrecen mecanismos y servicios «horizontales» reusables, este *middleware* está dirigido a mercados específicos (verticales).

Índice

Remote Method Invocation (RMI)

RMI es un middleware de distribución de objetos para Java donde tanto el servidor como el cliente se desarrollan en este lenguaje.

Usando RMI se pueden desarrollar aplicaciones distribuidas sin necesidad de escribir una sola línea de código relacionado con los detalles de comunicación entre las diferentes máquinas.

Para ello este framework ofrece:

- un conjunto de tipos y
- un conjunto de herramientas (aplicaciones).

RMI

Este middleware permite realizar aplicaciones distribuidas «casi» como si fueran aplicaciones locales. Esto es así porque **de forma transparente** permite:

- realizar la llamada alguno de los servicios que ofrece el servidor,
- enviar los datos que necesita ese procedimiento (*marshalling* o serialización)y
- recoger el resultado devuelto (*unmarshalling* o deserialización).

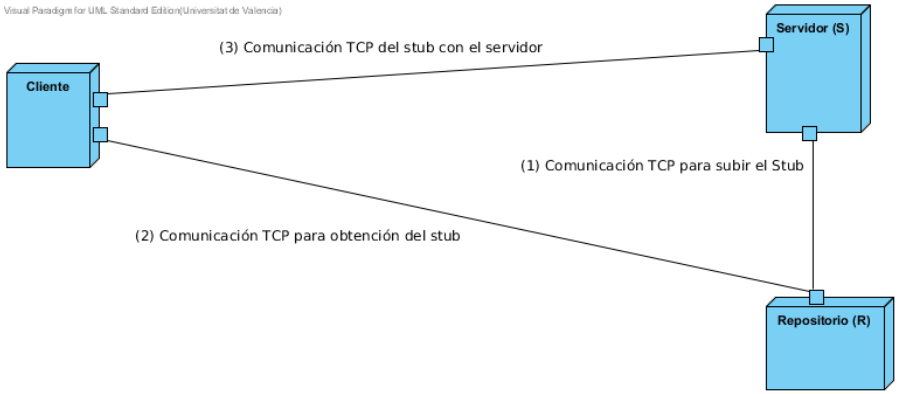
RMI

Para desarrollar la aplicación:

- En el lado del servidor:
  - Declarar una interfaz con la definición de los servicios que se podrán invocar remotamente y que extiende a `Remote`. Los métodos que se pueden invocar remotamente deben lanzar una excepción del tipo `RemoteException`.
  - Implementar la interfaz con la lógica de la aplicación.
  - Usar la clase `UnicastRemoteObject` que se encarga de exponer el servicio y de crear un *stub* (que se encarga de realizar las peticiones desde el cliente)
  - Registrar el *stub* en un registro de stubs.
- En el lado del cliente:
  - Conectar con el registro (que está en el servidor) para obtener el *stub*
  - Enviar mensajes al *stub* como si fuera un objeto local (aunque en la implementación de sus métodos realiza la comunicación con el servicio de forma transparente).

RMI

El *stub* es un objeto que ofrece las mismas operaciones que el objeto remoto (que son las que se han especificado en la interfaz) y en la implementación de estas operaciones se realiza la comunicación con el servidor.



RMI

- El servidor almacena el *stub* en un repositorio de *stubs* llamado registro RMI.
- El cliente obtiene el *stub* del registro RMI.
- El stub será usado para realizar las llamadas al servidor desde el cliente.
- El stub se lleva de una máquina a otra usando serialización y deserialización sobre TCP.
- El registro RMI no es más que una aplicación que almacena y provee de stubs.

Desarrollo

El primer paso para desarrollar una aplicación con RMI es declarar una interface que define los métodos que se podrán invocar de forma remota:

```
import java.rmi.*;

public interface RemoteCache extends Remote{
   public int addPair(String key, String value) throws RemoteException;
   public String getValue(String key) throws RemoteException;
}
```

El paquete `java.rmi` define la interface `Remote` y la excepción `RemoteException`.

Desarrollo I

Todos los argumentos de los métodos y los valores devueltos por los métodos que se pueden invocar de forma remota deben ser serializables (ya que se van a pasar de una máquina a otra a través de conexiones TCP).

En este caso tenemos `String` y un tipo primitivo que son serializables.

Desarrollo en el servidor

A continuación hay que declarar una clase que implemente a esa interface y que **representa la lógica de negocio de la aplicación** en el servidor.

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.util.concurrent.*;

public class RemoteCacheImpl implements RemoteCache{
   private ConcurrentHashMap<String,String> cache;

   public RemoteCacheImpl() throws RemoteException{
      cache = new ConcurrentHashMap<String,String>();
   }

   public int addPair(String key, String value){
      cache.put(key,value);
      return cache.size();
   }

   public String getValue(String key){
      return cache.get(key);
   }
}
```


Desarrollo en el servidor

Ahora hay que crear la clase con el `main` en la que:

1. Se crea el stub usando el método estático `exportObject` de la clase `UnicastRemoteObject` y se inicia el servidor (se expone el servicio)
2. Se registra el *stub* en el registro RMI:

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

class ServidorRMI{
   public static void main(String[] args){
      try{
         RemoteCache cache = new RemoteCacheImpl();

         // Exporta el servicio en un puerto libre y obtiene el stub
         RemoteCache stub = (RemoteCache)
↪  UnicastRemoteObject.exportObject(cache, 0);

         // Obtiene el registro RMI (en la misma máquina que el servicio)
         Registry registry = LocateRegistry.getRegistry();
         // y sube el stub al registro
         registry.rebind("cache", stub);

      }catch(Exception ex){
         ex.printStackTrace();
      }
   }
}
```


Desarrollo en el cliente

Para el cliente crearemos una clase que tenga el método `main` y en la que

1. Se obtiene el *stub* del registro RMI
2. Se le envían mensajes al *stub* como si fuese un objeto local (aunque sabemos que en la implementación de los métodos se encapsula la comunicación con el servidor).

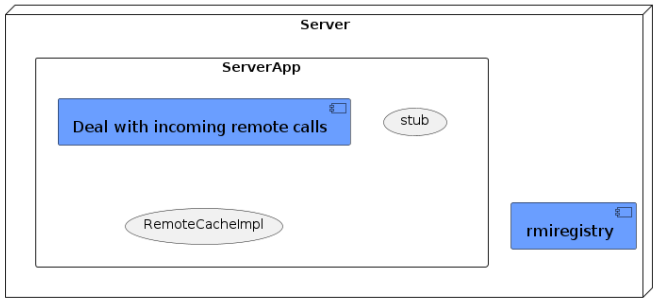
Desarrollo en el cliente

```
import java.rmi.*;
import java.rmi.registry.*;

class ClienteRMI{
   public static void main(String[] args){
      try{
         // El host donde está el registro se pasa como argumento
         String regHost = args[0];

         // Se obtiene el stub del registro RMI
         Registry registry = LocateRegistry.getRegistry(regHost);
         RemoteCache cacheStub = (RemoteCache) registry.lookup("cache");

         // Envío de mensajes al stub que se comunicará con el servidor
         System.out.println(cacheStub.addPair("host1","10.200.20.20"));
         System.out.println(cacheStub.addPair("port1","1002"));

         System.out.println(cacheStub.getValue("host1"));
         System.out.println(cacheStub.getValue("port1"));
      }catch(Exception ex){
         ex.printStackTrace();
      }
   }
}
```

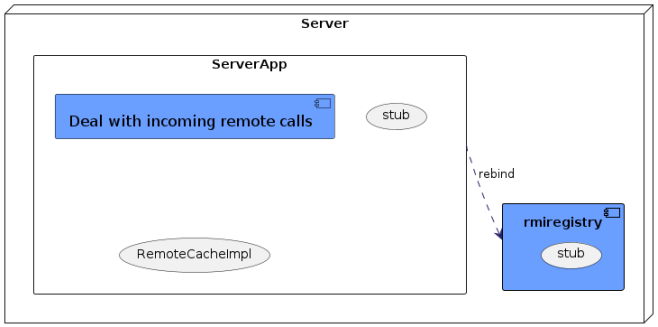

Compilación y despliegue

```
cd servidor
javac *.java

cd ../cliente
cp ../servidor/RemoteCatalog.java .
javac *.java

cd ../servidor
# Ejecución del registro RMI
rmiregistry &
# Ejecucución del servidor
java ServidorRMI &

cd ../cliente
# Ejecución del cliente (pasando el host del registro)
java ClienteRMI localhost
```

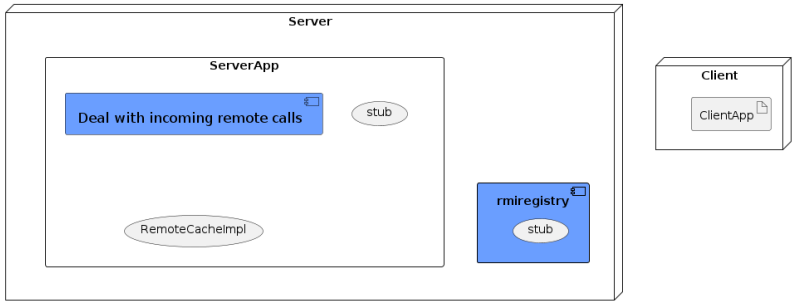

Despliegue de la aplicación



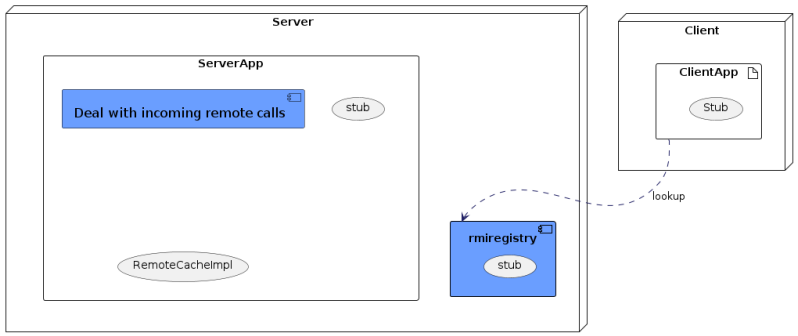
Despliegue de la aplicación



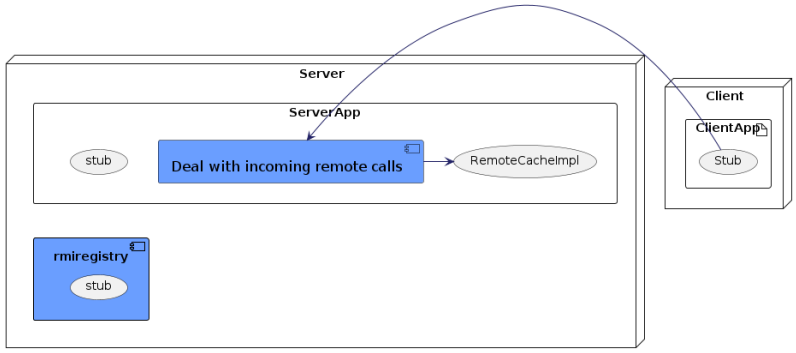
Despliegue de la aplicación



Despliegue de la aplicación



Despliegue de la aplicación



Enlaces

https://docs.oracle.com/javase/tutorial/rmi/overview.html