v 1.8

Tema 4
El paquete lang de Java

Programación Orientada a Objetos

José Francisco Vélez Serrano

Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Índice

- Cadenas de caracteres
- Las excepciones
- La clase Object
- Algunos métodos importantes de Object
  - Equals y Hashcode
  - compareTo (Comparable)
  - clone (Cloneable)
- Concurrencia
- Tipos enumerados

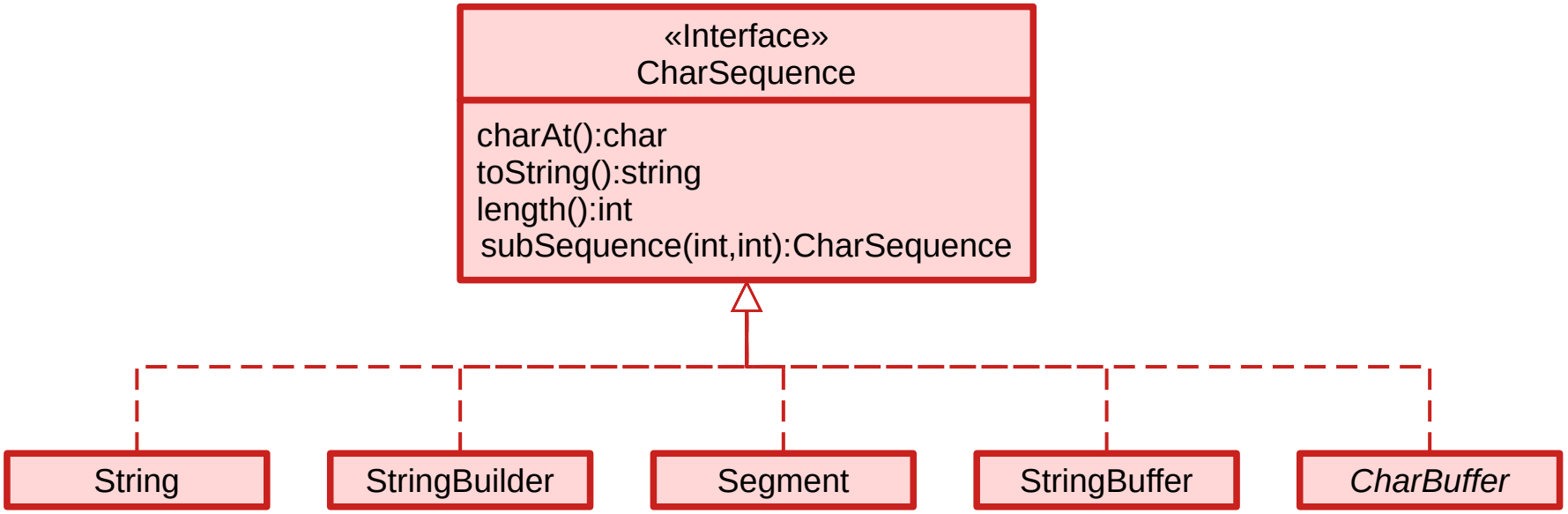
Cadenas de caracteres

java.lang es el paquete más importante de Java. Aporta interfaces y clases tan importantes que están integradas en el propio lenguaje y no es preciso importarlas. Esta primera sección se dedica a las cadenas de caracteres.

Clases para manejar cadenas de caracteres

Hay varias clases para manipular cadenas de caracteres frente al uso directo de arrays de caracteres en Java.

Todas ellas implementan la interfaz CharSequence que obliga a ofrecer charAt(), length() y subsequence.



Cadenas de caracteres

String

Proporciona métodos para el tratamiento de cadenas de caracteres: substring(), trim(), compareTo(), toCharArray().

Además, String proporciona métodos estáticos (como valueOf) para realizar operaciones sin tener que crear objetos de la clase String.

También sobrecarga + y += para tener una notación simple al concatenar Strings y tipos primitivos.

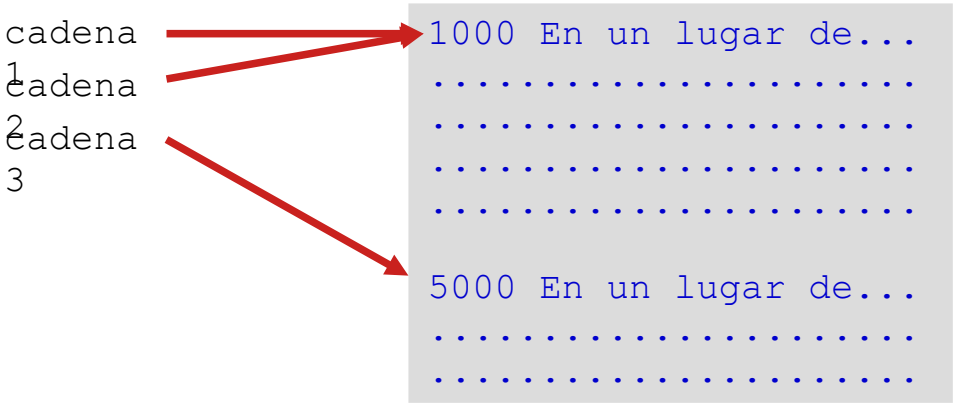
String
+ charAt():char + codePointAt():int + compareTo(String):int + concat(String):String + contains(CharSequence):boolean + contentEquals(CharSequence):boolean + copyValueOf(char[]):String + endWith(String):boolean + equals(Object):boolean + getBytes():byte[] + hashCode():int + indexOf(int):int + intern():String + isEmpty():boolean + lastIndexOf(int):int + length():int + matches(String):boolean + offsetByCodePoint(int):int + replace(char,char):String + split(String):String[] + startWith(String,int):boolean + subSequence(int,int):CharSequence + substring(int):String + toLowerCase():String + toUpperCase():String + trim():String +valueOf(double):String...

Creación de Strings

Cuando se declara una cadena entre comillas Java crea un String en el pool privado de la clase String que reutiliza. A estos String se les llama literales.

Al crear un String utilizando new se hace en el heap.

```
String cadena1 = "En un lugar de la Mancha";
String cadena2 = "En un lugar de la Mancha";
String cadena3 = new String("En un lugar de la Mancha");
```



Comparando dos String

El operador == compara las referencias a los objetos, por ello es posible que == devuelva false ante dos String que contienen la misma cadena.

Para comparar Strings se debe usar el método equals.

```
String cadena1 = "En un lugar de la Mancha";
String cadena2 = "En un lugar de la Mancha";
System.out.println(String.valueOf(cadena1 == cadena2));
System.out.println(String.valueOf(cadena1.equals(cadena2)));

String cadena3 = new String("En un lugar de la Mancha");
System.out.println(String.valueOf(cadena1 == cadena3));
System.out.println(String.valueOf(cadena1.equals(cadena3)));

Scanner sc = new Scanner(System.in);
String cadena4 = sc.nextLine();
System.out.println(String.valueOf(cadena1 == cadena4));
System.out.println(String.valueOf(cadena1.equals(cadena4)));
```

Características de String

String es inmutable. Todos los métodos de String que operan sobre el String devuelven un nuevo String. Un String no se puede modificar.

String está etiquetada como final para evitar que se pueda heredar de String y crear una versión mutable.

Como consecuencia, para cadenas que crecen no es conveniente usar String.

```
String cadena = "En un lugar de la Mancha ... del todo sin duda alguna. VALE";

cadena = cadena + ".";
```

StringBuilder

StringBuilder es una clase mutable de manejo de cadenas de caracteres. Por ello, está especialmente indicada para cadenas que pueden crecer.

Algunos métodos devuelven this, permitiendo la concatenación de métodos.

```
StringBuilder cadena = new StringBuilder();

cadena.append("En un lugar de la Mancha");

cadena.append("...")
  .append("del")
  .append("todo sin duda alguna. VALE");
```

StringBuilder
+ append(char):StringBuilder + append(double):StringBuilder + append(String):StringBuilder ... + capacity():int + charAt():char + codePointAt():int + delete(int, int):StringBuilder + equals(Object):boolean + getBytes():byte[] + hashCode():int + insert(int, char):StringBuilder + indexOf(int):int + lastIndexOf(int):int + length():int + offsetByCodePoint(int):int + replace(char,char):StringBuilder + reverse():StringBuilder + setCharAt(int,char) + setLength(int) + subSequence(int,int):CharSequence + substring(int):String + toString():String

Cadenas de caracteres

Segment

La clase Segment se utiliza para manipular segmentos de otras cadenas de caracteres sin necesidad de copia en un nuevo objeto.

Los objetos Segment son inmutables, aunque el array de char que contiene sea mutable.

Segment
+ charAt(int):char + clone():Object + current():char + first():char + getBeginIndex():int + getEndIndex():int + getIndex():int + isPartialReturn():boolean + last():char + length():int + next():char + previous():char + setIndex(int):char + setPartialReturn(boolean) + subSequence(int,int) + toString():String

```
char [] charArray = "En un lugar de la mancha".toCharArray();
Segment seq = new Segment(charArray, 5, 10);

char c = seq.current();
while (c != CharacterIterator.DONE) {
  System.out.print(c);
  c = seq.next();
}
```

Cadenas de caracteres

CharBuffer

La clase CharBuffer gestiona un buffer de caracteres. Principalmente, se usa para crear buffers de comunicación con dispositivos.

El método get devuelve un char y mueve la posición actual del buffer.

El método put inserta elementos.

CharBuffer abstracta y para crear objetos de ella hay que llamar al método allocate.

CharBuffer
+ allocate(int):CharBuffer + append(char c) + array():char[] + arrayOffset():int + asReadOnyBuffer():CharBuffer + charAt(int):char + compact():CharBuffer + compareTo(CharBuffer):int + duplicate():CharBuffer + equals(Object):boolean + get():char + hasArray() + hashCode() + isDirect() + length() + order() + put(char):CharBuffer + read(CharBuffer):int + slice():SharSequence + subSequence(int,int) + toString():String + wrap(char[]):CharBuffer

Cadenas de caracteres

StringBuffer

StringBuffer es una clase mutable de manejo de cadenas de caracteres.

La diferencia con StringBuilder es que StringBuffer es thread-safe. Por ello, está indicada para cadenas que pueden crecer en entornos multihilo y desaconsejada en cualquier otro caso por los retardos que introduce.

StringBuffer
+ append(char):StringBuffer + append(double):StringBuffer + append(String):StringBuffer ... + capacity():int + charAt():char + codePointAt():int + delete(int, int):StringBuffer + equals(Object):boolean + getBytes():byte[] + hashCode():int + insert(int, char):StringBuffer + indexOf(int):int + lastIndexOf(int):int + length():int + offsetByCodePoint(int):int + replace(char,char):StringBuffer + reverse():StringBuffer + setCharAt(int,char) + setLength(int) + subSequence(int,int):CharSequence + substring(int):String + toString():String

Las excepciones

En los lenguajes de programación orientados a objetos el concepto de error se cambia por el de la situación excepcional que se produce cuando un objeto no puede cumplir su contrato.

Concepto de excepción

Una excepción es un objeto que se lanza en el momento en el que ocurre una situación excepcional y que se captura cuando esa situación excepcional puede tratarse.

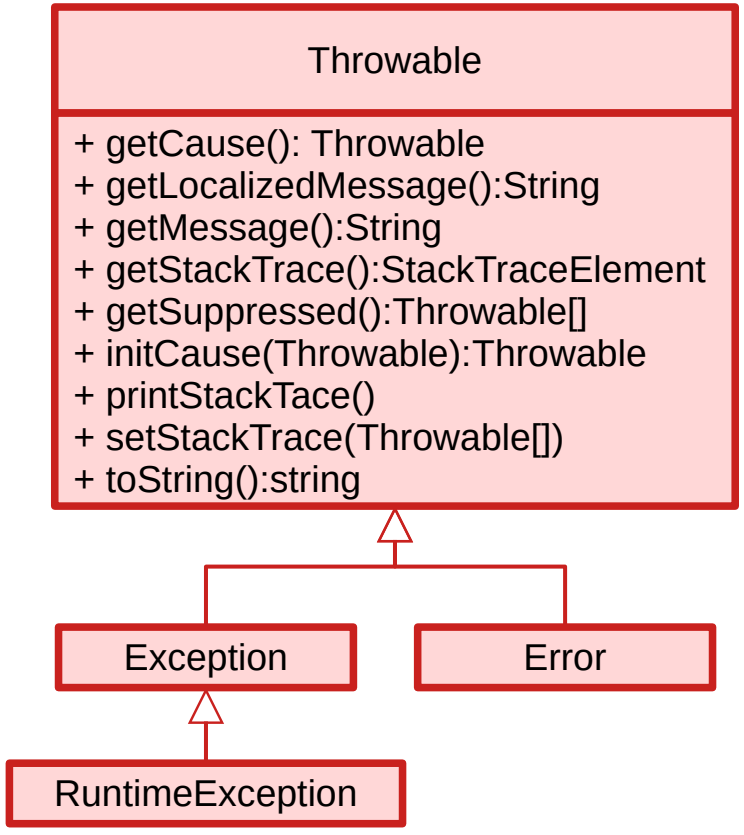
Cuando se captura una excepción se puede:

- Reintentar la operación,
- Realizar una acción alternativa
- Interrumpir la ejecución del programa.

Tipos de excepciones en Java

En Java, las excepciones derivan de la clase Throwable y hay dos tipos de excepciones: controladas (checked exceptions) y no controladas (unchecked exceptions).

- Cualquier clase que derive de Error o de RuntimeException genera excepciones no controladas.
- Por otro lado, las clases que deriven directamente de Throwable o Exception crean excepciones controladas.



Sintaxis de las excepciones en Java

En Java, para lanzar una excepción se usa la palabra reservada throw, y el objeto que se lanza debe ser heredero de Throwable o derivadas de esta.

Para definir el ámbito de captura de una excepción y el de su tratamiento se usan bloques try-catch-finally.

Los métodos que lanzan excepciones pueden indicarlo en su declaración, utilizando la palabra reservada throws seguida de los tipos de objetos que lanzaría.

Si un método puede lanzar excepciones controladas es obligatorio que lo indique en su declaración.

Uso práctico de las excepciones

```
class PilaEnteros {
          ...

  public int cima() throws RuntimeException {
    if (cima != null)
      return cima.valor;
    else
      throw new RuntimeException("Pila vacía");
  }

  public int desapilar() throws Exception  {
    Nodo aux = cima;
    if (cima == null)
      throw new Exception("Pila vacía");
    cima = cima.siguiente;
    return aux.valor;
  }
}
```

```
class Ejemplo {
  public static void main (String [ ] arg)   {
    PilaEnteros pila = new PilaEnteros();
    pila.apilar(10);

    try {
      int v = pila.desapilar();
      System.out.println (v);
    }
    catch (Exception e) {
      System.out.println (e.getMessage());
    }
  }
}
```

Más tipos de excepciones en Java

Con herencia se pueden crear nuevas clases de excepciones, pero lo normal es usar RuntimeException o reutilizar alguna de sus derivadas.

Derivadas de RuntimeException predefinidas en Java:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DataBindingException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IllformedLocaleException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MirroredTypesException, MalformedParameterizedTypeException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, ProviderNotFoundException, RasterFormatException, SecurityException, SystemException, RejectedExecutionException, TypeConstraintException, TypeNotPresentException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException.

Sobre los tipos de excepciones

Inicialmente se pensó que las excepciones controladas eran una buena idea, ya que los programadores de C++ solían ignorar el tratamiento de excepciones.

Posteriormente ningún lenguaje de programación ha replicado las excepciones controladas.

Actualmente, no se suelen usar las excepciones controladas por la sobrecarga de formalismo.

Normalmente las excepciones deben romper el programa a la espera de que el programador arregle un problema, por eso suele derivarse siempre de RuntimeException.

Excepciones vs valores de error

Una excepción no es un valor devuelto por una función. Por ello, el tratamiento de errores se simplifica pues no es necesario comprobar errores tras cada llamada a cada método.

Además, una excepción no puede ser ignorada, lo que garantiza que la situación excepcional será tratada antes o después.

```
// Ejemplo en C++ que trata el error
// después de cada print
if (printf("Menu") < 0)
  return -1;
if (printf("=======") < 0)
  return -1;
if (printf("a) Cargar") < 0)
  return -1;
if (printf("b) Grabar") < 0)
  return -1;
if (printf("c) Salir") < 0)
  return -1;
```

```
// Ejemplo en C++ que ignora el tratamiento
// de error después de cada print
printf("Menu");
printf("=======");
printf("a) Cargar");
printf("b) Grabar");
printf("c) Salir");
```

Excepciones vs valores de retorno

En Java se aconseja no utilizar excepciones para gestionar el flujo de control normal de un programa.

Hay veces que es difícil elegir entre lanzar excepción o devolver valores imposibles.

```
// Ejemplo que muestra el lanzamiento de una excecpción cuando el fichero no existe
// pero la devolución de -1 cuando el fichero alcanza el EOF
try {
  RandomAccessFile f = RandomAccessFile("ruta/Fichero.txt","r");
  int n = f.read();
  while (n != -1) {
    n = f.read();
  }
}
catch (FileNotFoundException e) {
  // Tratamiento del error
}
```

El log y las excepciones

El tratamiento de excepciones se suele combinar con la generación de ficheros de log.

Existen diferentes bibliotecas para la gestión de log en Java, aunque Java incluye Logger.

Logger permite definir el fichero dónde se guarda la información o el nivel del problema (severo, aviso, info...).

```
try {
  ...
} catch (Exception ex) {
  Logger.getLogger(Coche.class.getName()).log(Level.SEVERE, null, ex);
}
```

La clase Object

En esta sección se tratan los detalles de la clase Object, que es la clase báse de herencia de todas las clases en Java.

La clase Object

Jeraquía única

Toda clase que se declara en Java, y que no se especifica de que clase deriva, lo hace de la clase Object. Esto tiene como consecuencia que todas las clases de Java tienen como tipo común la clase Object.

Se pueden destacar dos ventajas:

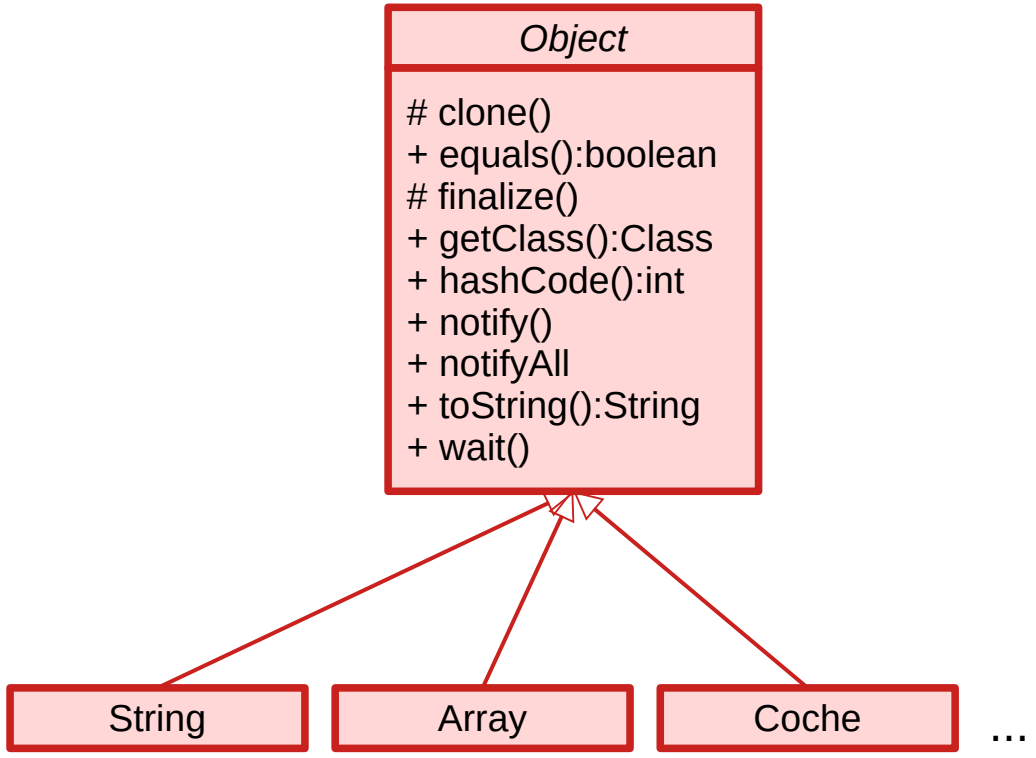
- Todos los objetos en última instancia son del mismo tipo y por lo tanto puede garantizarse ciertas operaciones sobre todos ellos.
- Permite definir estructuras de datos que almacenen objetos de tipo Object (y por tanto cualquier clase de objetos, pues todas derivan de Object).

La clase Object

Métodos de object

La clase Object define una serie de métodos que pueden utilizarse sobre todos los objetos que se creen en Java. Estos métodos son:

- equals() y hashCode()
- toString()
- clone()
- getClass()
- wait(), notify() y notifyAll()
- finalize



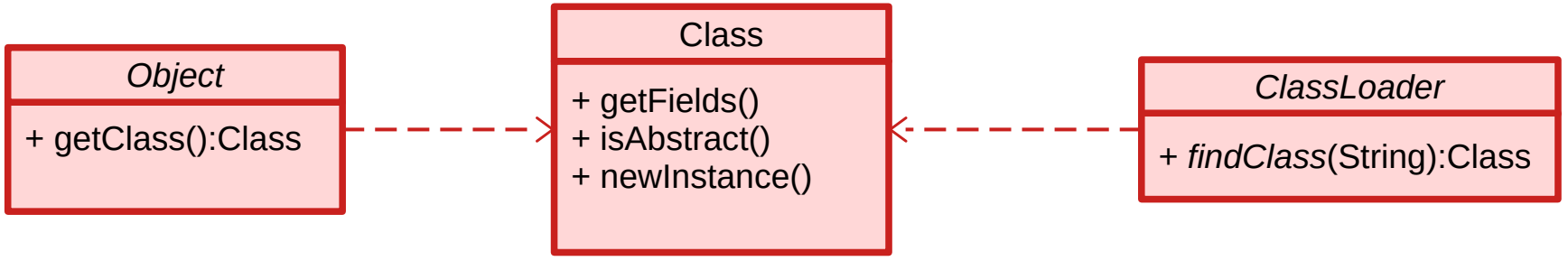
https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

La clase Object

Reflexión

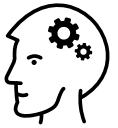
La reflexión es una propiedad de un lenguaje que permite a un programa revisar su propio código de manera dinámica. Incluso permite añadir clases y miembros nuevos durante la fase de ejecución.

- La clase Class permite inspeccionar una clase dinámicamente.
- La clase abstracta ClassLoader define la forma de cargar clases dinámicamente.



La clase Object

Un ejemplo de reflexión: el ClassLoader



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

public class Cargador extends ClassLoader
{
  public Class <?> findClass(String name) {
    byte[] b = loadClassData(name);
    return defineClass(name, b, 0, b.length);
  }
  private byte[] loadClassData(String name) {
    try {
      String nombre=".\\"+name+".class";
      InputStream in = new FileInputStream(nombre);
      Vector <Byte> buffer = new Vector<Byte>();

      int i = in.read();
      while(i != -1) {
        buffer.add((byte)i);
        i = in.read();
      }
      byte [] aux = new byte[buffer .size()];
      for (int c = 0; c < buffer.size(); c++)
        aux[c]=buffer.get(c);
      return aux;
    }
    catch(Exception e) {
         return null;
    }
  }
}
```

```
public static void main(String[] args) {
  try {
    ClassLoader loader = new Cargador();
    Class un_coche = loader.loadClass("Deportivo");
    Coche c = (Coche) un_coche.newInstance();
    c.acelerar();
  }
  catch(Exception e) {
    System.out.print(e.getMessage());
  }
}
```

Las anotaciones

Las anotaciones consiste en la posibilidad de definir etiquetas que empiezan con el carácter @.

Las anotaciones se pueden aplicar antes de cualquier clase, método o propiedad.

Las anotaciones no añaden ninguna funcionalidad por sí mismas, pero permiten que otras clases las encuentren y realicen acciones al respecto.

Las anotaciones se pueden consultar mediante reflexión, en ejecución, o durante la compilación usando un "Anotation Processor".

Ejemplo de creación y uso de anotaciones

El siguiente ejemplo define la anotación @Dividido.

La clase Motor, cuando recibe un objeto invoca el método “dame25”. Si está anotado con @Dividido imprime el resultado dividido por 2.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Dividido {
}

public class Controlado {
    @Dividido
    public int dame25() {
        return 25;
    }
}

public class Motor {
    void ejecutar(Object c) throws Exception {
        try {
            Class<? extends Object> o = c.getClass();
            Method method = o.getMethod("dame25");
            Integer i = (Integer) method.invoke(c);
            if (method.isAnnotationPresent(Dividido.class))
                System.out.println(i/2);
            else
                System.out.println(i);
    }
}

public class EjemploAnotaciones {
    public static void main(String[] args) throws Exception {
        Motor m = new Motor();
        m.ejecutar(new Controlado());
    }
}
```

Equals y Hashcode

Para comparar si dos objetos son iguales no basta con comparar sus referencias. En esta sección hablaremos de los mecanismos que proporciona Java para ello.

Importancia del método equals

Para comparar dos objetos la clase Object proporciona el método equals. Para comparar dos objetos debe invocarse equals sobre un objeto y pasarle el otro.

Así, si dos objetos son iguales equals devuelve true, y false en caso contrario. Formalmente, equals cumplirá:

- Identidad.- a.equals(a) = true ∀ a.
- Simétrica.- a.equals(b) ⇔ b.equals(a) ∀ a, b.
- Transitiva.- Si a.equals(b) y b.equals(c) ⇒ a.equals(c) ∀ a, b, c.
- Elemento neutro.- a.equals(null) = false ∀ a.

```
Coche c1 = new Coche("B-1212");
Coche c2 = new Coche("B-1212");
c1.acelerar();

boolean iguales = c1.equals(c2);
```

Equals y Hashcode

Reimplementación de equals

Desafortunadamente, su implementación por defecto solo compara las referencias (igual que hace ==).

Queda en manos de las diferentes clases la adecuada implementación de equals.

La sobrescritura de equals suele consistir en comparar cada una de las propiedades del objeto.

```
public class Coche {
  private int velocidad;
  final private String matricula;

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) {
      return false;
    }
    if (getClass() != obj.getClass()) {
      return false;
    }
    final Coche other = (Coche) obj;
    return matricula.equals(other.matricula);
  }
  public Coche(String matricula) {
    this.matricula = matricula;
  }
  public int acelerar() {
    velocidad++;
    return velocidad;
  }
}
```

Equals y Hashcode

Importancia del método hashCode

La clase Object proporciona el método hashCode cuya misión es devolver un resumen de cualquier objeto en forma de número entero.

hashCode se utiliza en diferentes bibliotecas de Java para, por ejemplo, indexar objetos. Por ello, siempre que se sobreescriba equals debe sobrescribirse hashCode para que si entre dos objetos:

- equals da false, sus hashCode sean diferentes.
- equals da true, sus hashCode sean iguales.

```
Coche c1 = new Coche("B-1212");
Coche c2 = new Coche("B-1212");
c1.acelerar();

System.out.println(c1.hashCode());
System.out.println(c2.hashCode());
```

Equals y Hashcode

Reimplementación de hashCode

Por defecto, hashCode devuelve el valor de la referencia al objeto.

Queda para las diferentes clases la correcta implementación de hashCode.

```
public class Coche {
  private int velocidad;
  final private String matricula;

  @Override
  public int hashCode() {
    int hash = 5;
    hash = 71 * hash + matricula.hashCode();
    return hash;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null) || (getClass() != obj.getClass()) {
      return false;
    }
    final Coche other = (Coche) obj;
    return matricula.equals(other.matricula);
  }

  public int acelerar() {
    velocidad++;
    return velocidad;
  }
}
```

Comparable

Para comparar si un objeto es mayor que otro Java proporciona la interfaz Comparable y los Comparators. En esta sección hablaremos de estos mecanismos.

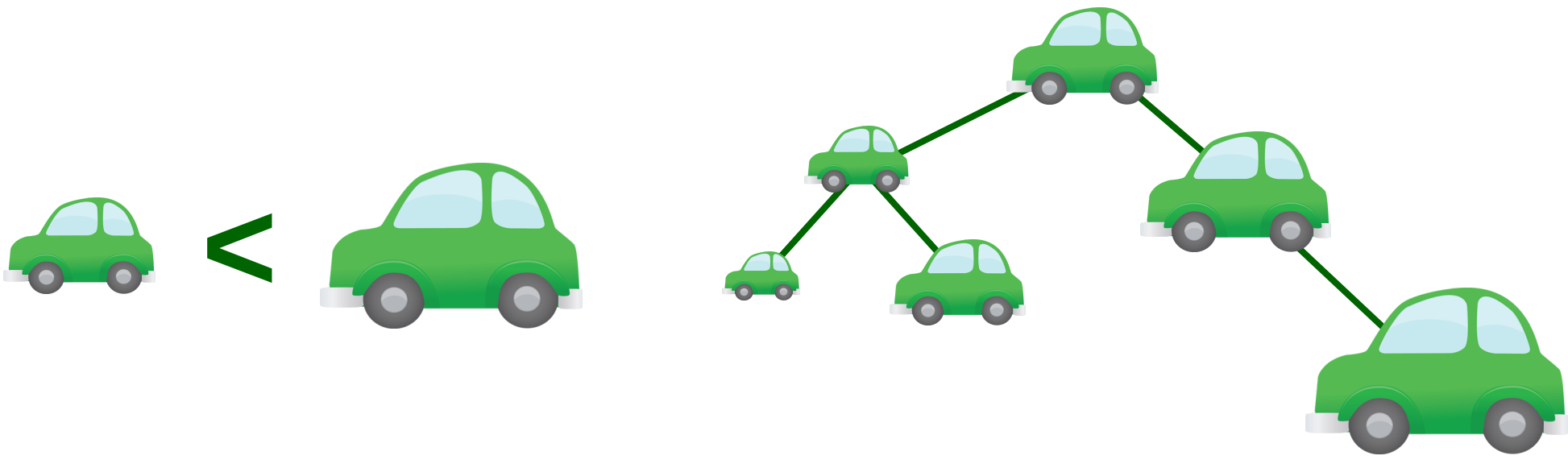
Comparable

El problema de comparar objetos

En multitud de situaciones es preciso comparar dos objetos del mismo tipo para saber cuál es mayor.

Por ejemplo, al ordenar objetos en un árbol binario.

Los lenguajes de POO suelen definir mecanismos para que estas comparaciones se realicen siempre de la misma forma.



La interfaz Comparable

Java considera que los objetos que cumplen la interfaz Comparable pueden ser comparados.

La interfaz Comparable solo define el método compareTo, que devuelve:

- Cero si el resultado de la comparación es la igualdad
- Un valor negativo si la instancia es menor que el parámetro
- Un valor positivo si la instancia es mayor que el parámetro

Si compareTo entre dos objetos da 0, es recomendable (pero no obligatorio), que equals dé true.

Implementando Comparable

En principio no tiene sentido comparar objetos de clases diferentes.

Para comparar dos objetos solo deben utilizarse sus propiedades inmutables.

```
public class Coche implements Comparable {
    private final String matricula;
    private int velocidad;

    public Coche(String matricula) {
        this.matricula = matricula;
    }

    public int acelerar() {
        velocidad++;
        return velocidad;
    }

    @Override
    public int compareTo(Object o) {
        Coche c = (Coche) o;
        return (matricula.compareTo(c.matricula));
    }
}
```

Comparable

La interfaz Comparator

La interfaz Comparable introduce un único medio de comparación.

En ocasiones es necesario comparar objetos utilizando diferentes criterios.

Por ejemplo, comparar dos frases por su orden lexicográfico, o por el número de caracteres, o por el número de palabras.

Para desligar el criterio de comparación del objeto, Java introduce la interfaz Comparator.

Comparable

Creando un Comparator

```
public class Coche {
    private final String matricula;
    private int velocidad;

    public Coche(String matricula) {
        this.matricula = matricula;
    }

    public int acelerar() {
        velocidad++;
        return velocidad;
    }

    String getMatricula() {
        return matricula;
    }

    String getVelocidad() {
        return velocidad;
    }
}
```

```
public class CarComparator implements Comparator {
   @Override
   public int compare(Object o1, Object o2) {
     Coche c1 = (Coche) o1;
     Coche c2 = (Coche) o2;
     return c1.getMatricula().compareTo(c2.getMatricula());
   }
}
```

```
public class SpeedComparator implements Comparator {
   @Override
   public int compare(Object o1, Object o2) {
     Coche c1 = (Coche) o1;
     Coche c2 = (Coche) o2;
     return c1.getVelocidad() - c2.getVelocidad();
   }
}
```

```
public static void main(String[] args) {
  Coche c1 = new Coche("BMQ-2112");
  Coche c2 = new Coche("ABC-1234");
  c1.acelear();

  CarComparator cc = new CarComparator();
  SpeedComparator sc = new SpeedComparator();

  if (cc.compare(c1,c2) > 0)
    System.out.println("c1 tiene mayor matrícula");

  if (sc.compare(c1,c2) > 0)
    System.out.println("c1 es más rápido");
}
```

Clonado de objetos

En esta sección se analizan las posibilidades que proporciona Java para realizar la copia de objetos.

Clonado de objetos

El problema

En Java, el operador de asignación entre dos referencias no copia el objeto, sino que proporciona dos referencias al mismo objeto.

Para poder realizar copias de un objeto Java propone dos alternativas:

- Crear un constructor copia.
- Usar la interfaz Cloneable y el método clone de Object.

Clonado de objetos

El constructor copia

El constructor copia es un constructor que recibe un objeto del mismo tipo como parámetro y configura el estado del objeto de manera adecuada para que parezca una copia del parámetro.

```
public class Coche {
  private int velocidad;

  //Constructor copia
  public Coche(Coche c) {
    this.velocidad = c.velocidad;
  }

  public int acelerar() {
    velocidad++;
    return velocidad;
  }
}

...

Coche c = new Coche();
c.acelerar();
Coche copia = new Coche(c); //Llamada al constructor copia
```

Clonado de objetos

Interfaz cloneable y método clone

La interfaz Cloneable indica que un objeto se puede clonar y posibilita invocar al método clone de Object.

Para lograr clonado se debe:

- Declarar que la clase implementa la interfaz Cloneable.
- Declarar como público un método clone que internamente invoque a super.clone.

```
public class Coche implements Cloneable {
  private int velocidad;

  public int acelerar() {
    velocidad++;
    return velocidad;
  }
  @Override
  public Coche clone() throws CloneNotSupportedException {
    return (Coche) super.clone();
  }
}
```

Clonado de objetos

Clonado profundo

El clonado realizado por Object copia las referencias del objeto clonado, no clona a su vez los objetos contenidos en el objeto clonado. Por eso se llama clonado superficial (shallow).

Para obtener clonado profundo, tanto con clone como con el constructor copia, se debería implementar explícitamente.

```
public class Rueda implements Cloneable {
  final int radio;
  public Rueda(int radio) {
    this.radio = radio;
  }
  public Rueda(Rueda rueda) {
    this.radio = rueda.radio;
  }
  @Override
  public Rueda clone() throws CloneNotSupportedException {
    return (Rueda) super.clone();
  }
}

public class Coche implements Cloneable {
  private int velocidad;
  private Rueda[] rueda = new Rueda[4];
  public Coche() {
    for (int i = 0; i < 4; i++)
      rueda[i] = new Rueda(30);
  }
  public Coche(Coche c) {
    this.velocidad = c.velocidad;
    for (int i = 0; i < 4; i++)
      rueda[i] = new Rueda(c.rueda[i]);
  }
  public int acelerar() {
    velocidad++;
    return velocidad;
  }
  @Override
  public Coche clone() throws CloneNotSupportedException {
    Coche c = super.clone();
    for (int i = 0; i < 4; i++)
      c.rueda[i] = (Rueda) rueda[i].clone();
    return c;
  }
}
```

Clonado de objetos desconocidos

El clonado de objetos desconocidos puede implicar el uso de introspección.

```
Coche car = new Coche("BAV-2121");

Object objeto = car;

Class clase = objeto.getClass();

Method method = clase.getDeclaredMethod("clone", null);

Coche car2 = (Coche) method.invoke(o, null);
```

Clonado de objetos

El constructor copia vs interfaz cloneable

Las ventajas del constructor copia son:

- No obliga a implementar ninguna interfaz.
- No requiere hacer casting.
- Permite copiar propiedades finales.

Las ventajas de usar clone son:

- Sencillo de realizar debido a la mínima codificación.
- La interfaz Cloneable indica a otros objetos que un objeto se puede copiar sin necesidad de conocer su interfaz.

Habitualmente se implementan ambas soluciones porque son complementarias.

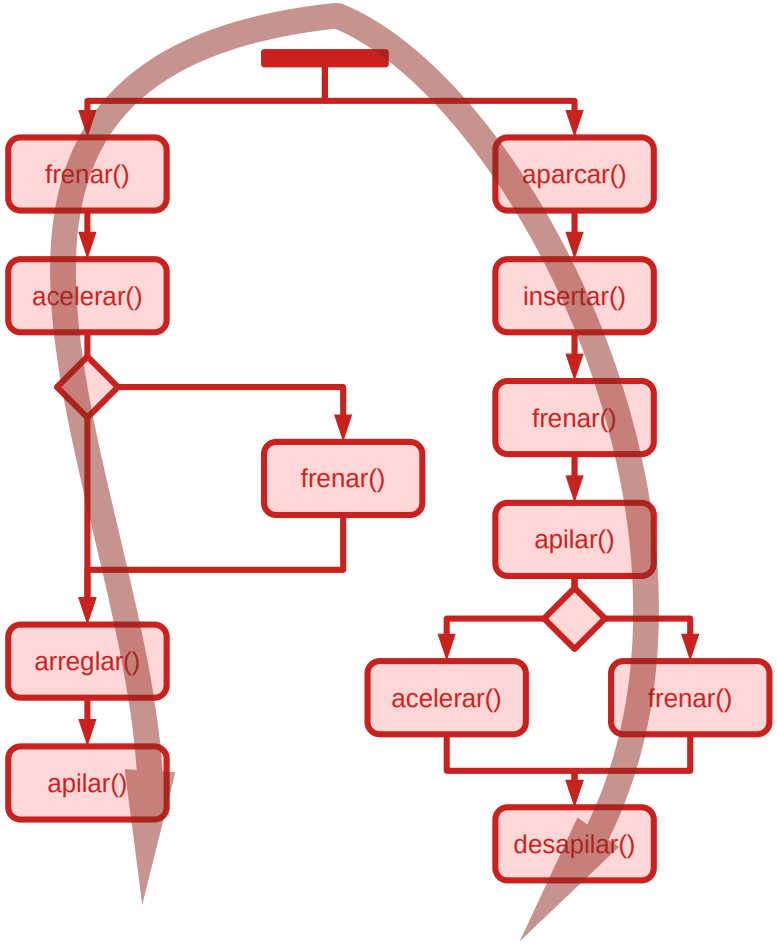
Concurrencia

En esta sección se introducen brevemente los mecanismos que proporciona Java para ejecutar simultáneamente varios hilos de proceso.

Ejecución concurrente de código

Actualmente, los ordenadores suelen disponer de varias unidades de proceso que le permiten ejecutar simultáneamente varias instrucciones.

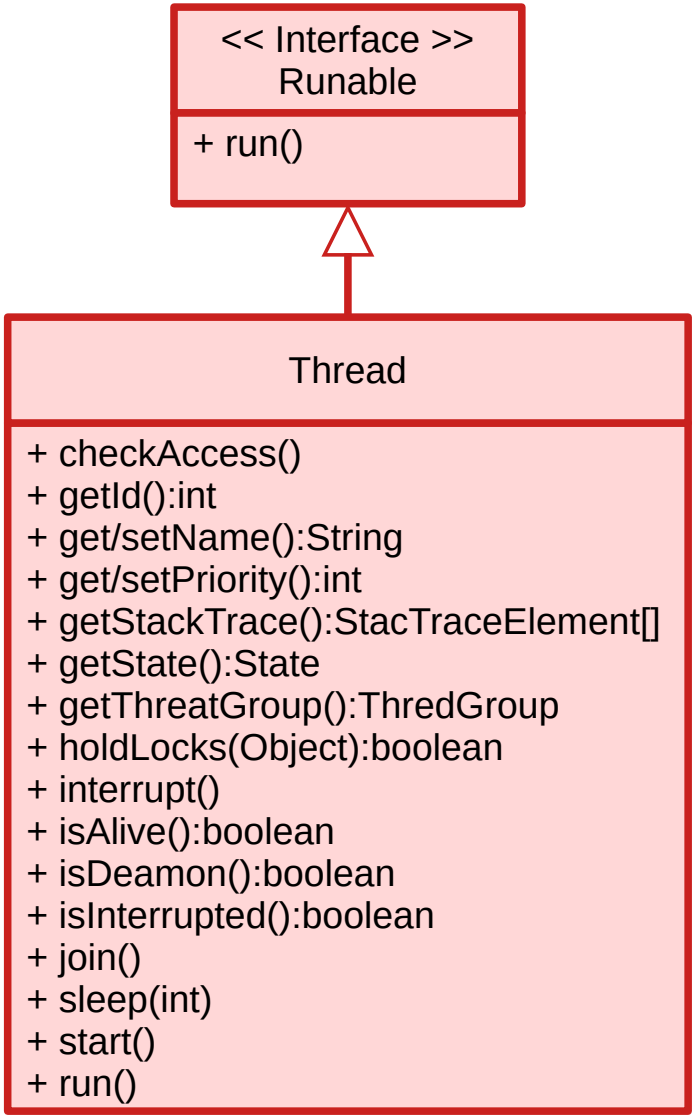
Por ello, la ejecución simultanea de diferentes secuencias de líneas de código es una característica cada vez más utilizada en los lenguajes de programación.



La clase Thread

Para crear un hilo separado de ejecución en Java basta con heredar de la clase Thread e implementar el método run() que en la clase Thread es abstracto.

```
public class Coche extends Thread {
  public void run() {
    for (int cont = 0; cont < 100; cont++) {
      System.out.print("BRUM ");
    }
  }
}
public class HelloWorld {
  public static void main(String[] args) {
    Coche c = new Coche();
    c.start();
    for (int cont = 0; cont < 1000; cont++) {
      System.out.print("PARA ");
    }
  }
}
```



Synchronized y la región de exclusión mutua

Cada objeto de Java dispone de un cerrojo (lock).

En las clases se pueden definir regiones (métodos o bloques) etiquetadas como synchronized.

Si un hilo intenta ejecutar una región synchronized mira el cerrojo: si está abierto, lo cierra, ejecuta la región, y al final, lo abre; si está cerrado se bloquea hasta que se abra.

Todos los bloques synchronized de un objeto forman una región de exclusión mutua. Dichas regiones se usan para evitar que desde varios hilos se puedan modificar simultáneamente partes claves del objeto, lo que daría lugar a estados no deseados.

El estado del threat

Java define 4 posibilidades para el estado de un thread dentro de un objeto o de la parte estática de una clase:

- nuevo, cuando se ha iniciado pero no ha comenzado a ejecutarse.
- ejecutable, cuando se está ejecutando, o no lo está pero nada impide que lo estuviese.
- muerto, cuando ha finalizado su método `run()`.
- bloqueado, si ha llamado a `sleep()`, si está esperando para ejecutar un bloque etiquetado synchronized, si está esperando eventos de E/S o si ha llamado a `wait()` esperando una llamada a `notify()` desde ese objeto.

Tipos enumerados

En esta sección se explica cómo se creaban tipos enumerados en Java antes de la versión 5, y como se pueden crear ahora.

Antes de los enumerados

Antes del JDK 5 Java no disponía de tipos enumerados.

Para simular los enumerados, algunos programadores usaban propiedades enteras estáticas y finales. Esto tenía el problema de no asegurar el tipo, permitiendo mezclas con enteros.

```
class Dia {
  public final static int lunes = 0;
  public final static int martes = 1;
  public final static int miercoles = 2;
  public final static int jueves = 3;
  public final static int viernes = 4;
  public final static int sabado = 5;
  public final static int domingo = 6;
}
```

Otra posibilidad más compleja era usar una clase de la que solo se creaban un número limitado de objetos (los valores enumerados). Esta es la base de los enumerados que se introducen con JDK 5.

```
class Dia {
  private Dia() {}
  public final static Dia lunes = new Dia();
  public final static Dia martes = new Dia();
  public final static Dia miercoles = new Dia();
  public final static Dia jueves = new Dia();
  public final static Dia viernes = new Dia();
  public final static Dia sabado = new Dia();
  public final static Dia domingo = new Dia();
}
```

Los enumerados en Java

En la versión 5 del JDK se introduce los tipos enumerados con la siguiente sintaxis.

```
enum <Identificador del tipo> {[identificadores]*};
```

Por ejemplo:

```
enum Dia {lunes, martes, miercoles, jueves, viernes, sabado, domingo};

...

Dia d = lunes;
```

Los enumerados y los switches

Java permite utilizar los tipos enumerados en switches.

```
public class EjemploSemanal {

  enum Dia {L, M, X, J, V, S, D}

  public static void main(String[] args) {
    Dia d = Dia.S;

    switch (d) {
      case S :
      case D : System.out.println("Findeee");
               break;
      default: System.out.println("A trabajar");
    }

  }
}
```

Tipos enumerados

Un tipo enumerado es una clase

Un enumerado es una clase, por tanto:

- Puede tener su propio fichero.
- Hereda de Object y por tanto implementa equals, hashCode…
- Implementa Comparable.
- Se pueden sobreescribir sus métodos.
- Se pueden añadir nuevos constructores, métodos y propiedades.

```
public enum Dia {
  L, M, X, J, V, S, D;

  public String toString() {
    switch (this) {
      case L: return "Lunes";
      case M: return "Martes";
      case X: return "Miercoles";
      case J: return "Jueves";
      case V: return "Viernes";
      case S: return "Sabado";
      default: return "Domingo";
    }
  }
}
```

```
public class Ejemplo {

  public static void main(String[] args) {
    Dia d1 = Dia.S;
    Dia d2 = Dia.L;
    System.out.println(d1.compareTo(d2));
    System.out.println(d1);
  }
}
```

Referencias para saber más

- "El lenguaje de programación Java", 3ª Edición, Arnold Gosling, Addison Wesley, 2001.
- "Piensa en Java". Eckel, 2ª Edición, Addison Wesley, 2002.
- "Introducción a la programación orientada a objetos con Java", C. Thomas Wu, Mc Graw Hill, 2001.
- "Proceso de Anotaciones de Java en tiempo de compilación", Nadum De Silva (web)