Grado en Ingeniería Informática
Metodología de la Programación

UAL. Área de Lenguajes y Sistemas Informáticos. Mercedes Martínez Durbán

Tema 2. Genéricos, Clases Abstractas e Interfaces

- Genéricos
- Clases abstractas
- Ejemplos de clases abstractas: Number, Calendar
- Interfaces
- Ejemplos de interfaces: Comparable, Cloneable, Iterable, Iterator, Comparator
- Interface vs. Clases abstractas
- Guía en el diseño de clases
- Resumen

Genéricos

Los genéricos permiten **detectar errores en tiempo de compilación** en lugar de en tiempo de ejecución.

Permiten parametrizar tipos. Con esta capacidad, definimos clases o métodos con tipos genéricos que el compilador puede sustituir por un tipo concreto. Por ejemplo, Java define la clase **ArrayList** para almacenar elementos de tipo genérico. Para esta clase genérica, creamos un objeto **ArrayList** para tener string y un objeto **ArrayList** para tener números. En este caso, strings y números son tipos concretos que sustituyen al tipo genérico.

El beneficio clave de los genéricos es habilitar errores para ser detectados en tiempo de compilación en lugar de en tiempo de ejecución. Una clase o método genérico nos permite especificar los tipos de objetos autorizados con los que la clase o método puede trabajar. Si usamos un objeto incompatible, el compilador detecta el error.

Aprenderemos a definir y usar clases, interfaces y métodos genéricos y demostraremos como los genéricos pueden usarse para mejorar la legibilidad y fiabilidad del software.

El uso de genéricos está permitido a partir del **JDK 1.5.** Muchas clases e interfaces se han tenido que modificar para usar los genéricos. Por ejemplo, la inteface **java.lang. Comparable**.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Antes de JDK 1.5

```
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o)
}
```

(b) JDK 1.5

<T> representa un *tipo genérico formal*, el cual puede ser reemplazado después con un *tipo concreto actual*. Sustituir un tipo genérico se llama *instanciación genérica*. Por convención, una letra mayúscula **E** o **T** se usa para denotar un tipo genérico formal.

Para ver los beneficios del uso de genéricos, analizaremos el código siguiente:

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Antes de JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Un tipo **genérico** puede definirse para una **clase o una interface**. Un tipo **concreto** puede especificarse cuando se usa la clase para **crear un objeto** o usando la clase o la interface para **declarar una variable referencia**.

Un **método genérico** puede definirse para un **método estático**.

Para declarar un método genérico, pondremos el tipo genérico **<E>** inmediatamente después de la palabra reservada **static** en la cabecera del método. Por ejemplo:

```
public static <E> void print(E[] list)
```

Para invocar un método genérico, anteponemos al nombre del método el tipo actual entre paréntesis angulares <>. Por ejemplo:

```
ClaseMetodoGenerico.<Integer>print(enteros);
ClaseMetodoGenerico.<String>print(cadenas);
```

O simplemente:

```
print(enteros);
print(cadenas);
```

En este caso, el tipo actual no se especifica explícitamente. El compilador automáticamente halla el tipo actual.

Un tipo **genérico** puede especificarse como un subtipo de otro tipo. Este tipo genérico se llama *encerrado*. Por ejemplo:

```
1  public class BoundedTypeDemo {
2    public static void main(String[] args ) {
3      Rectangle rectangle = new Rectangle(2, 2);
4      Circle circle = new Circle(2);
5
6      System.out.println("Same area? " +
7        equalArea(rectangle, circle));
8    }
9
10   public static <E extends GeometricObject> boolean equalArea(
11       E object1, E object2) {
12     return object1.getArea() == object2.getArea();
13   }
14 }
```

Haría referencia a las clases Rectangulo, Circulo y ObjetoGeometrico del tema 2.

Compatibilidad de tipos raw y hacia atrás

Una clase o interface usada sin especificar un tipo concreto se llama *raw type (puro, bruto)* y permite compatibilidad hacía atrás con versiones de Java anteriores.

```
ArrayList lista =new ArrayList( )  // tipo raw
ArrayList  <String> ciudades = new ArraysList<String>( );
```

Usando *tipos raw* permitimos compatibilidad con otros versiones de Java.

Una manera mejor de escribir el método **max** es usando un tipo genérico. Por ejemplo:

```
1  package org.mp.tema02;
2  public class MaxUsandoTiposGenericos {
3    /** Devuelve el máximo de dos objetos */
4    public static <E extends Comparable<E>> E max(E o1, E o2) {
5      if (o1.compareTo(o2) > 0)
6        return o1;
7      else
8        return o2;
9    }
10 }
```

Si invocamos el método max(“Hola”, 23) de la clase MaxUsandoTiposGenericos ????

Borrador y restricciones en genéricos

La información en genéricos la usa el compilador pero no está disponible en tiempo de ejecución. Esto se llama *tipo limpiador o borrador (erasure).*

Los genéricos se implementan usando una aproximación que es el *tipo limpiador.* El compilador usa la información del tipo genérico para compilar el código y la borra después. Por tanto, la información no está disponible en tiempo de ejecución. Esto permite que el código con genéricos sea compatible hacia atrás con el código heredado que usa *tipos raw.*

Los tipos genéricos están presentes en tiempo de compilación. Después de que el compilador confirma que un tipo genérico se ha usado satisfactoriamente, convierte el tipo genérico en tipo raw.

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

Cuando una clase, interface o método genérico se compila, el compilador sustituye el tipo genérico por el tipo Object. El compilador convierte el método siguiente de (a) a (b).

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

Si el tipo genérico es encerrado, el compilador sustituye este por el tipo encerrado. El compilador convierte el método siguiente de (a) a (b).

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

Restricciones

Puesto que los tipos genéricos son borrados en tiempo de ejecución, hay ciertas restricciones sobre cómo se pueden usar los genéricos. Algunas son:

Restricción 1: No usar new E( )

No se puede crear una instancia usando un tipo genérico.

E objeto = new E( ); **ERROR!!**

La razón es que **new E( )** se ejecuta en tiempo de ejecución y los genéricos no están disponibles en tiempo de ejecución.

Restricción 2: No usar new E[ ]

No se puede crear un array usando un tipo genérico.

E[ ] array = new E[capacidad]; **ERROR!!**

Podemos crear un array de tipo Object y después hacer un casting a E[ ]:

E[ ] array = (E[ ])new Object[capacidad];

La creación de un array genérico usando una clase genérica no está permitido. Por ejemplo:

ArrayList<String>[ ] lista = new ArrayList<String>[10]; No permitido

ArrayList<String>[ ] lista = (ArrayList<String>[])new ArrayList[10]; permitido

Sin embargo, seguiremos recibiendo una advertencia por el compilador.

Restricción 3: Un tipo genérico parametrizado o una clase no está permitido en un contexto estático

Dado que todas las instancias de una clase genérica tienen la misma clase en tiempo de ejecución, las variables y métodos estáticos de una clase genérica son compartidos por todas las instancias. Por lo tanto, es ilegal referirse a un tipo genérico en una clase en un método estático, campo.

```
public class Test<E> {
  public static void m(E, o1){   //   ilegal
  }
  public static E o1;   //  ilegal
```

Restricción 4: Las clases excepciones no pueden ser genéricas

Una clase genérica no puede extender a java.lang.Throwable.

```
public class MiExcepcion <T> extends Exception {     //  ilegal
}
```

¿Por qué?

Si eso fuese posible, tendríamos código con una cláusula **catch**

```
try {
    ……..
} catch (MiExcepcion<T>  e) {
    ……..
}
```

La JVM tiene que comprobar la excepción lanzada desde el bloque **try** mirando si coincide con el tipo especificado en el bloque **catch**. Esto es imposible porque la información del tipo no está presente en tiempo de ejecución.

Una superclase define un comportamiento común para las subclases relacionadas. Una *interface* se puede usar para definir un comportamiento común de clases no relacionadas.

Antes de hablar de interfaces, introduciremos un tema estrechamente relacionado: las *clases abstractas.*

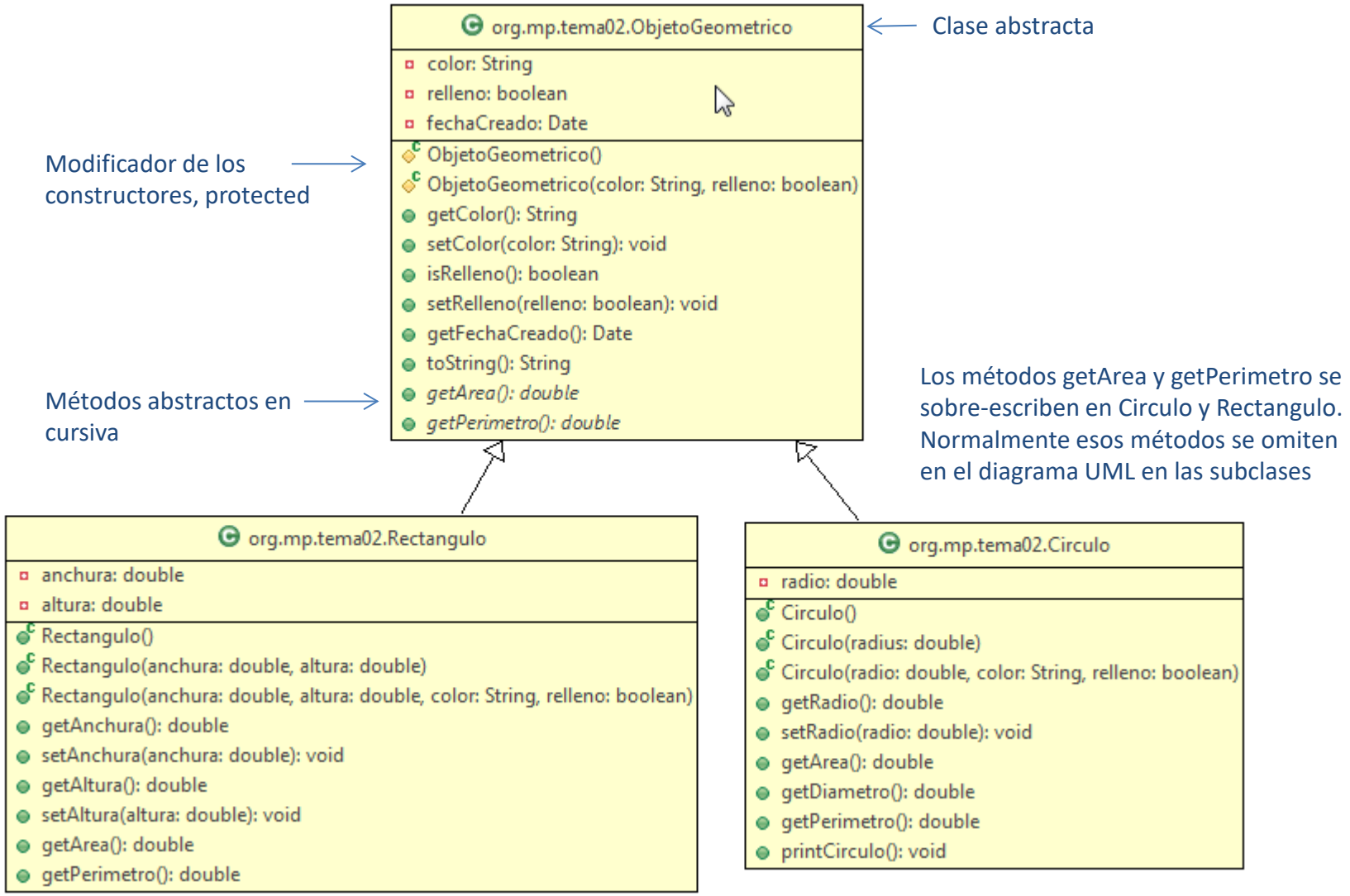
Clases Abstractas

Una clase abstracta no puede usarse para crear objetos. Puede contener métodos abstractas cuya implementación se hace en las subclases concretas.

Si retomamos el ejemplo del tema anterior de las clases ObjetoGeometrico, Circulo y Rectangulo, los métodos getArea() y getPerimetro() pueden declararse como abstractos en la clase base, ObjetoGeometrico y deberán ser implementados en las clases derivadas. De esta manera, la clase ObjetoGeometrico pasa a ser una clase abstracta puesto que contiene dos métodos abstractos.

public abstract class ObjetoGeometrico

En el diagrama UML, los nombres de las *clases abstractas* y de los *métodos abstractos* se indican en cursiva


org.mp.tema02.ObjetoGeometrico
color: String
relleno: boolean
fechaCreado: Date
ObjetoGeometrico()
ObjetoGeometrico(color: String, relleno: boolean)
getColor(): String
setColor(color: String): void
isRelleno(): boolean
setRelleno(relleno: boolean): void
getFechaCreado(): Date
toString(): String
getArea(): double
getPerimetro(): double
Clase abstracta
Modificador de los constructores, protected
Métodos abstractos en cursiva
Los métodos getArea y getPerimetro se sobre-escriben en Circulo y Rectangulo. Normalmente esos métodos se omiten en el diagrama UML en las subclases
org.mp.tema02.Rectangulo
anchura: double
altura: double
Rectangulo()
Rectangulo(anchura: double, altura: double)
Rectangulo(anchura: double, altura: double, color: String, relleno: boolean)
getAnchura(): double
setAnchura(anchura: double): void
getAltura(): double
setAltura(altura: double): void
getArea(): double
getPerimetro(): double
org.mp.tema02.Circulo
radio: double
Circulo()
Circulo(radius: double)
Circulo(radio: double, color: String, relleno: boolean)
getRadio(): double
setRadio(radio: double): void
getArea(): double
getDiametro(): double
getPerimetro(): double
printCirculo(): void

El constructor de la clase abstracta está declarado como **protected,** porque se usa exclusivamente en las clases derivadas. Cuando creamos una instancia de una subclase concreta, el constructor de la superclase se invoca para inicializar las propiedades definidas en la superclase.

Un método abstracto no puede estar en una clase que no sea abstracta. Si una subclase de una superclase abstracta no implementa todos los métodos abstractos, la subclase se debe definir como abstracta. En otras palabras, en una subclase que no sea abstracta que hereda de una clase abstracta, todos los métodos abstractos deben ser implementados aunque no se usen en la subclase.

Una clase abstracta no puede instanciar objetos usando el operador **new** , puedes definir constructores que serán invocados en los constructores de las subclases.

Una clase que contiene un método abstracto pasa a ser una clase abstracta. Ahora bien, es posible definir una clase abstracta que no contenga ningún método abstracto. En este caso no podremos crear instancias de la clase. Esa clase se usa como clase base para definir subclases.

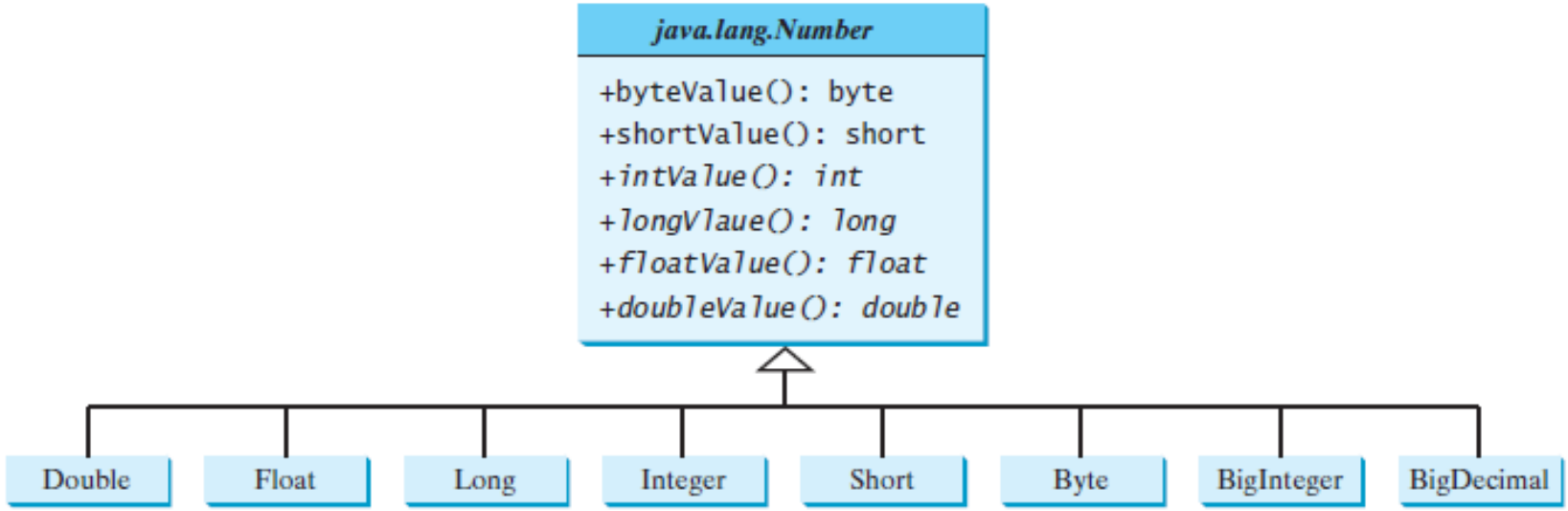
Una subclase puede ser abstracta incluso si la superclase es concreta.

No se puede crear una instancia de una clase abstracta usando el operador **new**, pero una clase abstracta puede ser usada como un tipo de dato. La siguiente sentencia sería correcta:

ObjetoGeometrico [] objetos = new ObjetoGeometrico [10]; objetos[0] = new Circulo();

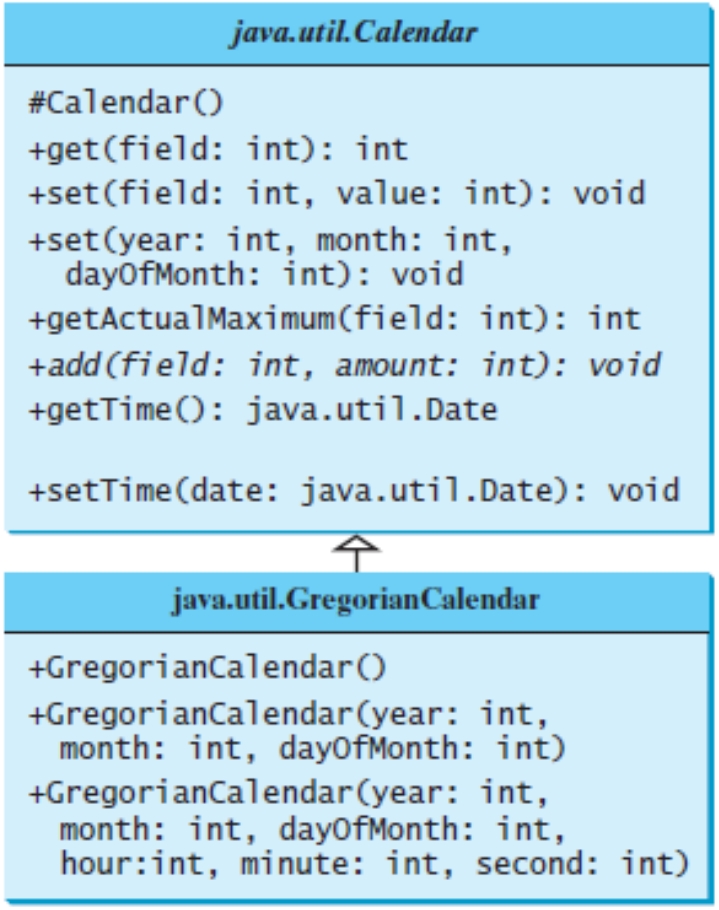
Ejemplos de clases abstractas

La clase abstracta *Number*



Number es una superclase abstracta para las clases envoltorios (wrapper), **BigInteger** y **BigDecimal.** Los métodos *intValue(), longValue(), floatValue()* y *doubleValue()* no se pueden implementar en la clase ***Number*** y se declaran como abstractos, de ahí que la clase pase a ser abstracta.

La clase abstracta *Calendar* y la subclase GregorianCalendar



Constructor por defecto de Calendar.
Devuelve el valor de un campo dado de calendar.
Modifica un campo dado de calendar por el valor especificado
Modifica calendar con año, mes y día especificados. El parámetro mes comienza en 0, por tanto enero es 0.
Devuelve el máximo valor de un campo de calendar que indiquemos.
Añade o elimina la cantidad especificada a la hora a partir de un campo dado de calendar.
Devuelve un objeto Date que representa el valor de la hora de ese calendar (en milisegundos)
Modifica la hora de ese calendar con un objeto Date dado.

Constructor de GregorianCalendar a partir de la hora en curso.
Constructor de GregorianCalendar al que hay que especificarle el año, mes y el día.
Constructor de GregorianCalendar a partir de año, mes día, hora, minuto y segundo. El parámetro mes está basado en 0, por lo tanto enero es 0.

Una instancia de java.util.Date representa un instante especifico de una hora con una precisión de milisegundos.

java.util.Calendar es una clase abstracta base para extraer información detallada tal como el año, mes, día, hora, minuto, segundo de un objeto Date.

Subclases de Calendar pueden implementar sistemas de calendarios específicos tales como el calendario Gregoriano, Lunar, Judío. Actualmente, la API de Java sostiene java.util.GregorianCalendar para el calendario Gregoriano.

El método **get(int field)** definido en la clase Calendar es muy útil para extraer el día y la hora a partir de un objeto Calendar. Los campos se definen como constantes, se puede ver en la diapositiva que sigue.

El método **get(int field)** definido en la clase Calendar es muy útil para extraer el día y la hora a partir de un objeto Calendar. Los campos se definen como constantes, se puede ver a continuación.

Constante	Descripción
YEAR	Año del calendario.
MONTH	Mes del calendario con 0 para enero.
DATE	Día del calendario.
HOUR	Hora del calendario (Notación 12 horas).
HOUR_OF_DAY	Hora del calendario (Notación 24 horas).
MINUTE	Minuto del calendario.
SECOND	Segundo del calendario.
DAY_OF_WEEK	El número del día de la semana con 1 para el Domingo.
DAY_OF_MONTH	Igual para el mes.
DAY_OF_YEAR	El número del día en el año,con 1 para el primer día de año.
WEEK_OF_MONTH	El número de semana en el mes.
WEEK_OF_YEAR	El número de semana en el año.
AM_PM	Indicador de AM y PM(0 para AM y 1 para PM).

Interfaces

¿Qué es una interface? ¿Porqué son útiles las interfaces?

Una interface es una clase que solo contiene constantes y métodos abstractos. En muchos aspectos, una interface es similar a una clase abstracta, pero el propósito de una interface es especificar características comunes para los objetos. Por ejemplo, podemos especificar que los objetos son comparables, clonables… usando las interfaces apropiadas.

Definir una interface

Para distinguir una interface de una clase, Java usa la sintaxis que sigue:

```
public interface NombreInterface {
  declaración de constantes;
  cabeceras de métodos;
}
```

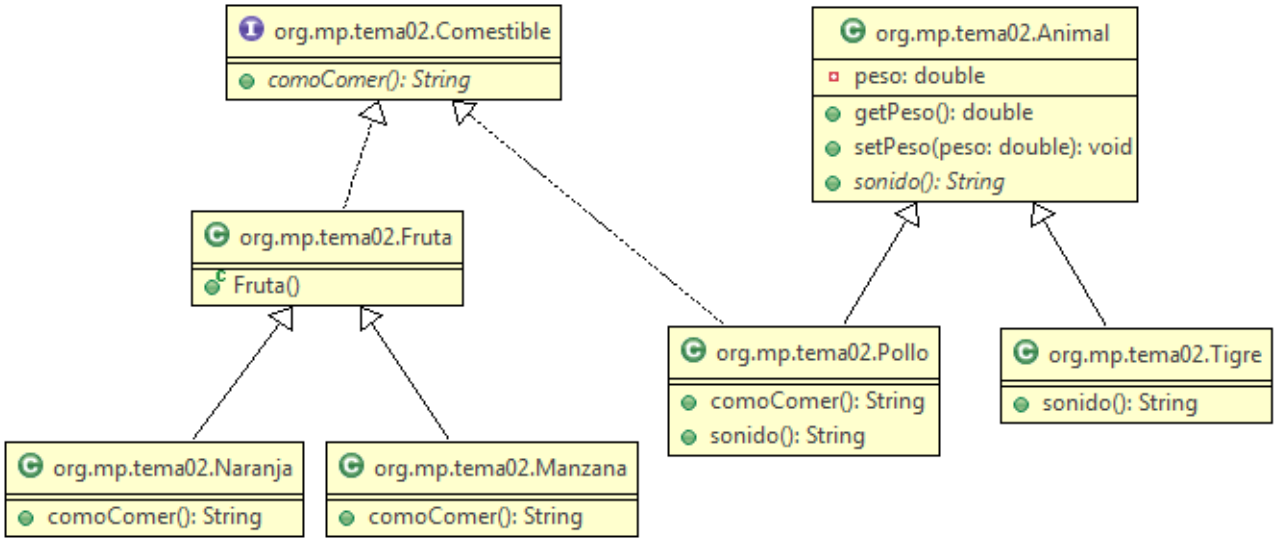
Ejemplo:

```
public interface Comparable {
  public int compareTo(Object o);
}
```

Una interface se trata como una clase especial en Java. Cada interface se compila en un archivo bytecode separado como una clase normal. Igual que en una clase abstracta, no se puede crear una instancia de una interface usando el operador **new**. Se puede utilizar una interface más o menos de la misma manera que una clase abstracta. Por ejemplo, se puede usar como un tipo de dato para declarar una variable, como el resultado de un casting y así sucesivamente.

Ejemplo

Utilizaremos la interface **Comestible** para especificar cuando un objeto es comestible. Esto se logra haciendo que la clase del objeto implemente la interface usando la palabra reservada **implements**. Por ejemplo, la clase Fruta y Pollo implementan la interface Comestible



```java
package org.mp.tema02;
public interface Comestible {
  /** Describe cómo comer */
  public abstract String comoComer();
}
```

```java
package org.mp.tema02;
public abstract class Animal {
  private double peso;

  public double getPeso() {
    return peso;
  }

  public void setPeso(double peso) {
    this.peso = peso;
  }

  /** Devuelve el sonido de un animal*/
  public abstract String sonido();
}
```

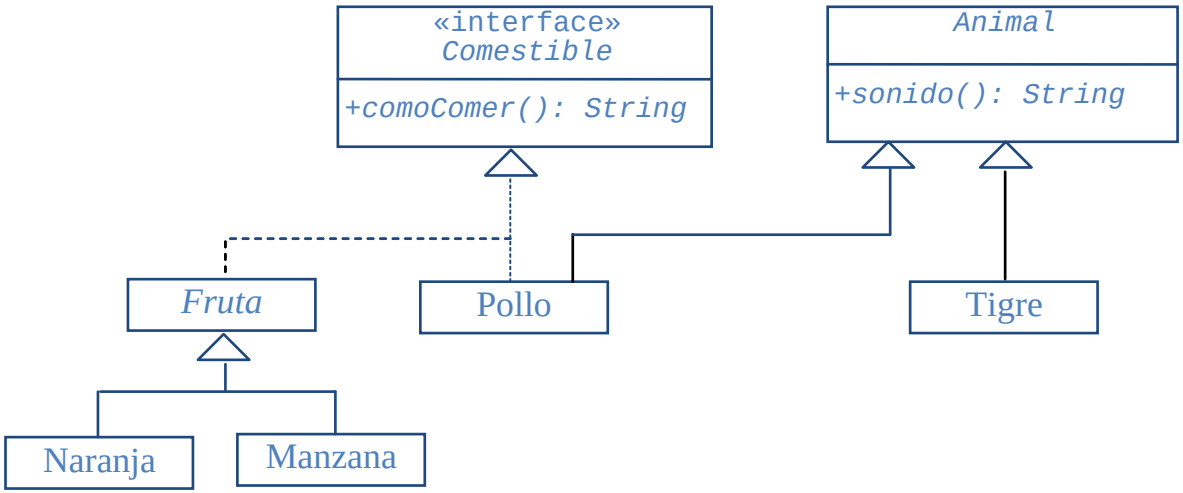
```java
package org.mp.tema02;
public abstract class Fruta implements Comestible {
    //se han omitido las propiedades y los métodos
    public Fruta(){

    }
}
```

```java
package org.mp.tema02;
public class Pollo extends Animal implements Comestible {
  @Override
  public String comoComer() {
    return "Pollo: Freírlo";
  }
  @Override
  public String sonido() {
    return "Pollo: Pío, pío, pío";
  }
}
```

```java
package org.mp.tema02;
public class Tigre extends Animal {
  @Override
  public String sonido() {
    return "Tigre: RGRGRGRGR";
  }
}
```

```java
package org.mp.tema02;
public class Naranja extends Fruta {
  @Override
  public String comoComer() {
    return "Naranja: Hacer un zumo";
  }
}
```

```java
package org.mp.tema02;
public class Manzana extends Fruta {
  @Override
  public String comoComer() {
    return "Manzana: Hacer sidra";
  }
}
```



Comestible es un supertipo para Pollo y Fruta. Animal es un supertipo para Pollo y Tigre. Fruta es un supertipo para Naranja y Manzana.

La clase Fruta implementa a la clase Comestible pero no implementa el método comoComer por eso debe ser declarada como abstracta. Las subclases concretas de Fruta deben implementar el método comoComer.

En esencia, la interface *Comestible* define comportamientos comunes para objetos comestibles. Todos los objetos comestibles tienen el método **comoComer.**

```java
package org.mp.tema02;
public class TestComestible {
  public static void main(String[] args) {
    Object[] objetos = {new Tigre(), new Pollo(), new Manzana()};
    for (int i = 0; i < objetos.length; i++) {
      if (objetos[i] instanceof Comestible)
        System.out.println(((Comestible)objetos[i]).comoComer());

      if (objetos[i] instanceof Animal) {
        System.out.println(((Animal)objetos[i]).sonido());
      }
    }
  }
}
```

Salida

```
Tigre: RGRGRGRGR
Pollo: Freírlo
Pollo: Pío, pío, pío
Manzana: Hacer sidra
```

Modificadores omitidos en las interfaces

Todas los miembros datos son **public final static** y todos los métodos son **public *abstract*** en una interface. Por este motivo, esos modificadores se pueden omitir, tal y como se muestra:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalente

```
public interface T1 {
  int K = 1;

  void p();
}
```

A una constante definida en una interface se puede acceder usando la sintaxis:

NombreInterface.NOMBRE_CONSTANTE

Por ejemplo: T1.K

Interface Comparable

La interface Comparable define el método **compareTo()** para comparar objetos. Esta interface se define como sigue:

```
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

El método **compareTo()** determina el orden del objeto con el especificado como **o**. Devuelve un entero negativo, cero o positivo si el objeto es menor, igual o mayor que **o**.

La interface Comparable es una **interface genérica**. El tipo genérico E se sustituye por un tipo concreto al implementar esta interface.

Muchas clases de las librerías de Java implementan Comparable para definir un orden natural para los objetos. Las clases Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String y Date todas implementan la interface Comparable.

Clases Integer y BigInteger

```
public class Integer extends Number
    implements Comparable<Integer> {
  // cuerpo de la clase omitido

  @Override
  public int compareTo(Integer o) {
    // Implementación omitida
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // cuerpo de la clase omitido

  @Override
  public int compareTo(BigInteger o) {
    // Implementación omitida
  }
}
```

Clases String y Date

```
public class String extends Object
    implements Comparable<String> {
  // cuerpo de la clase omitido

  @Override
  public int compareTo(String o) {
    // Implementación omitida
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // cuerpo de la clase omitido

  @Override
  public int compareTo(Date o) {
    // Implementación omitida
  }
}
```

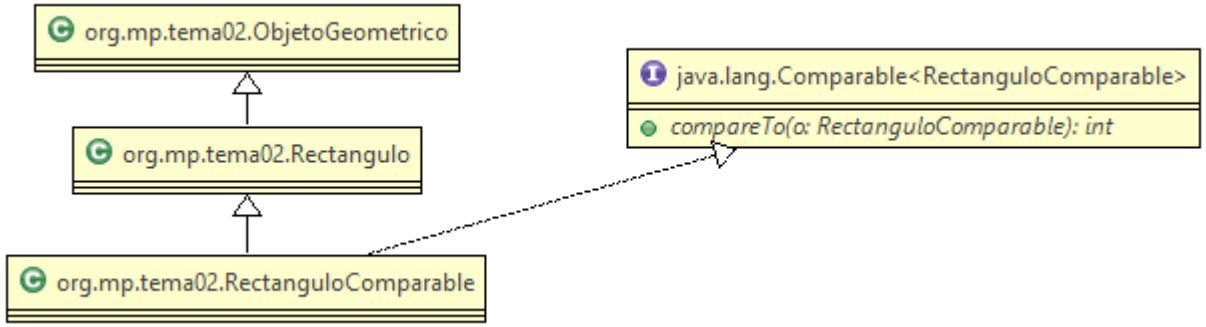
Sea **n** es un objeto de la clase **Integer**, **s** un objeto de la clase **String** y **d** un objeto de la clase **Date**. Las siguientes expresiones son **true.**

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

Definiendo clases que implementen Comparable



Ejemplo de uso

```java
package org.mp.tema02;
public class RectanguloComparable extends Rectangulo implements Comparable <RectanguloComparable>{

    public RectanguloComparable (double anchura, double altura){
        super(anchura, altura);
    }

    /**
     * Método que compara dos rectángulos devolviendo 0 si son iguales, -1 si el
     * primero es menor y 1 si el primero es mayor
     *
     * @param obj
     * @return un entero 0, -1, 1
     */
    public int compareTo(RectanguloComparable o) {
        if (getArea() == o.getArea())
            return 0;
        else if (getArea() < o.getArea())
            return -1;
        else
            return 1;

    }
    @Override
    public String toString() {
        return super.toString()+ "Area: " + getArea();
    }
```

```java
package org.mp.tema02;
public class OrdenacionRectangulos {
  public static void main(String[] args) {
    RectanguloComparable[] rectangulos = {
      new RectanguloComparable(3.4, 5.4),
      new RectanguloComparable(13.24, 55.4),
      new RectanguloComparable(7.4, 35.4),
      new RectanguloComparable(1.4, 25.4)};
    java.util.Arrays.sort(rectangulos);
    for (Rectangulo rectangulo: rectangulos) {
      System.out.print(rectangulo + " ");
      System.out.println();
    }
  }
}
```

Salida

```
Creado el Sun Feb 28 12:30:41 CET 2016
color: blanco y relleno: false Area: 18.36
Creado el Sun Feb 28 12:30:41 CET 2016
color: blanco y relleno: false Area: 35.559999999999995
Creado el Sun Feb 28 12:30:41 CET 2016
color: blanco y relleno: false Area: 261.96
Creado el Sun Feb 28 12:30:41 CET 2016
color: blanco y relleno: false Area: 733.496
```

Interface Cloneable

Una interface contiene constantes y métodos abstractos, sin embargo, la interface Cloneable es un caso especial. Se define como sigue:

```
package java.lang;

public interface Cloneable {
}
```

En ocasiones, necesitamos crear una copia de un objeto. Para hacer esto, necesitamos usar el método clone() y entender la interface Cloneable.

Interface marcadora (*marker interface)* : es una interface vacía.

Una interface de este tipo no tiene ni constantes ni métodos. Se usa para indicar que una clase posee ciertas propiedades deseables.

Una clase que implemente la interface **Cloneable** es marcada clonable y sus objetos pueden clonarse usando el método clone() definido en la clase Object.

Muchas clases en la librería de Java implementan Cloneable. Por ejemplo, Date, Calendar…Por lo tanto instancias de esas clases se pueden clonar.

```
Calendar calendario = new GregorianCalendar(2003, 2, 1);
Calendar calendario1 = calendario;
System.out.println("calendario == calendario1 es " +
  (calendario == calendario1));
Calendar copiaCalendario = (Calendar)calendario.clone();
System.out.println("calendario == copiaCalendario es " +
  (calendario == copiaCalendario));
System.out.println("calendario.equals(copiaCalendario) es " +
  calendario.equals(copiaCalendario));
```

Salida

calendario == calendario1 es true

calendario == copiaCalendario es false

calendario.equals(copiaCalendario) es true

Podemos clonar un array :

```
int[] lista1 = {1, 2};
int[] lista2 = lista1.clone();
lista1[0] = 7;
lista2[1] = 8;
System.out.println("lista1 es " + lista1[0] + ", " + lista1[1]);
System.out.println("lista2 es " + lista2[0] + ", " + lista2[1]);
```

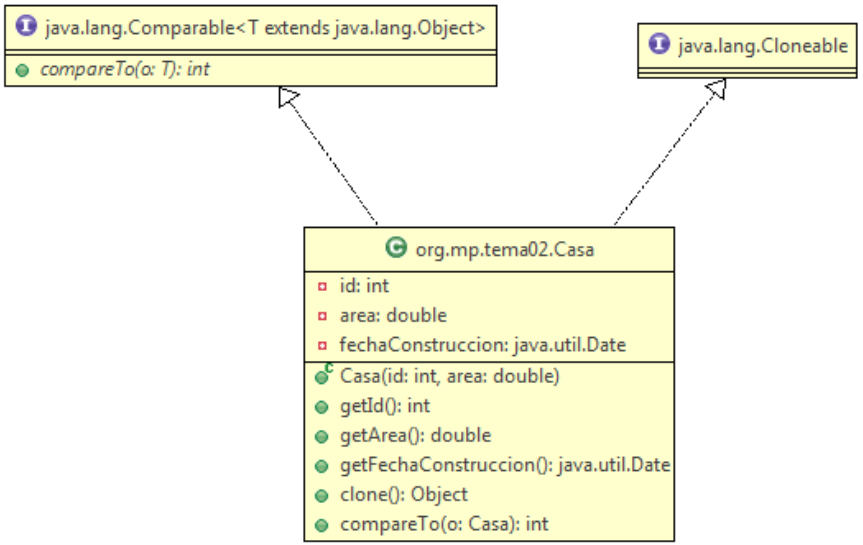
Salida

lista 1 es 7, 2

lista 2 es 1, 8

Implementando la interface Cloneable

Para definir una clase que implemente la interface Cloneable, la clase debe sobre-escribir el método clone() de la clase Object. Las siguientes líneas de código definen una clase llamada **Casa** que implementa Cloneable y Comparable.




```
1  package org.mp.tema02;
2  public class Casa implements Cloneable, Comparable<Casa> {
3    private int id;
4    private double area;
5    private java.util.Date fechaConstruccion;
6
7    public Casa(int id, double area) {
8      this.id = id;
9      this.area = area;
10     fechaConstruccion = new java.util.Date();
11   }
12
13   public int getId() {
14     return id;
15   }
16     public double getArea() {
17     return area;
18   }
19
20   public java.util.Date getFechaConstruccion() {
21     return fechaConstruccion;
22   }
23
24   @Override /** Sobre-escribe el método clone protected  definido en
25     la clase Object y refuerza su accesibilidad */
26   public Object clone() {
27     try {
28       return super.clone();
29     }
30     catch (CloneNotSupportedException ex) {
31       return null;
32     }
33   }
34
35   @Override // Implementa el método compareTo definido en Comparable
36   public int compareTo(Casa o) {
37     if (area > o.area)
38       return 1;
39     else if (area < o.area)
40       return -1;
41     else
42       return 0;
43   }
44 }
```

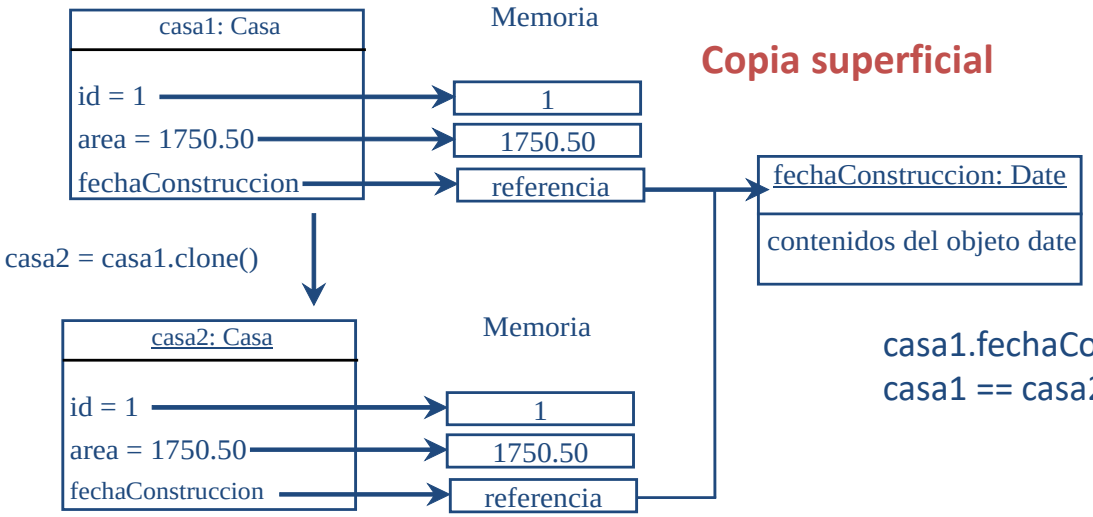
Excepción que se lanza si se usa el método clone() en una clase que no lo ha implementado .

Copia superficial vs. copia profunda

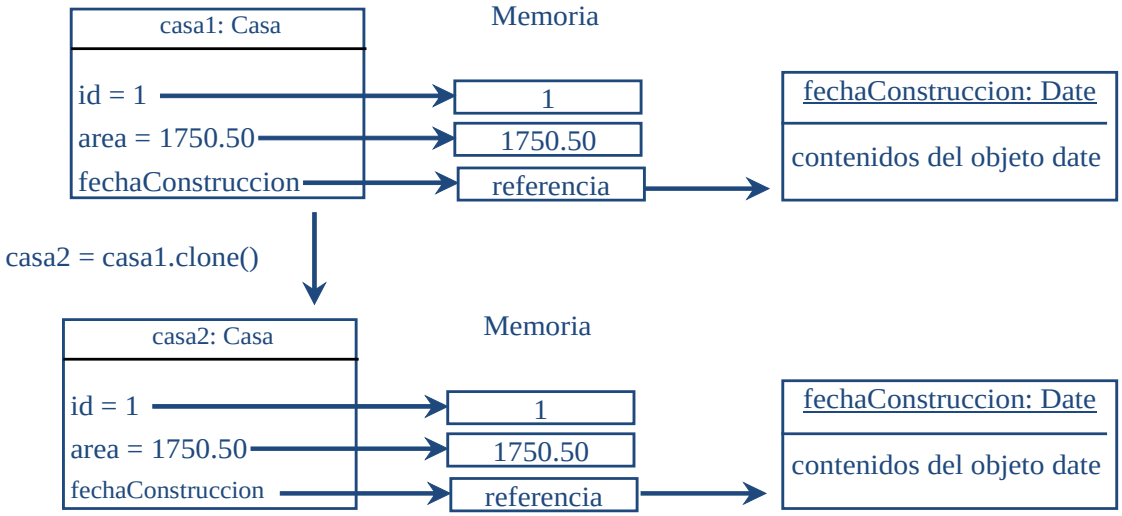
Vamos a crear un objeto de la clase Casa y también una copia idéntica de él:

```
Casa casa1 = new Casa(1, 1750.50);
Casa casa2 = (Casa)casa1.clone();
```

casa1 y casa2 son dos objetos diferentes con idénticos contenidos. El método clone() de la clase Object copia cada propiedad del objeto original en el objeto objetivo. Si los propiedades son de tipo primitivo, sus valores se copian. Por ejemplo, el valor del área (tipo double) se copia desde casa1 a casa2. Si las propiedades son objetos, entonces las referencias se copian en casa2 como se muestra en la figura. Esto se conoce como una una copia **superficial** frente a una copia **profunda** que significa que si una propiedad es un objeto, se copia el contenido del objeto.



casa1.fechaConstruccion == casa2.fechaConstruccion es true
casa1 == casa2 es false



Copia profunda

Para hacer una copia en profundidad de un objeto, el método clone() sería:

```
public Object clone() throws CloneNotSupportedException {
     try{
          // Realiza una copia superficial
          Casa casaClonada = (Casa)super.clone();
          // Copia profunda de fechaConstruccion
          casaClonada.fechaConstruccion = (java.util.Date)(fechaConstruccion.clone());
          return casaClonada;
     }catch (CloneNotSupportedException ex) {
          return null;
     }
}
```

Interfaces Iterable e Iterator

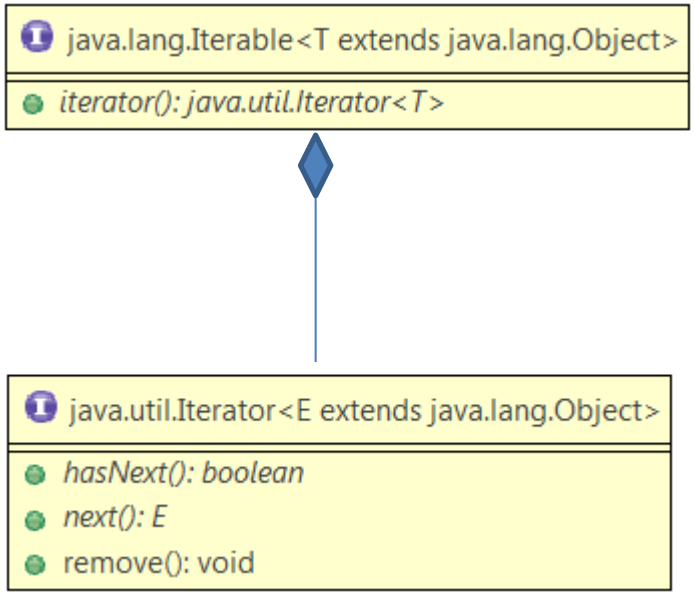
La interface **Iterator** es un *patrón de diseño clásico*[1] para recorrer una estructura de datos sin necesidad de detallar cómo los datos están almacenados en la estructura de datos.

La interface **Iterable** define el método **iterator** , el cual devuelve un iterador, una instancia de la interface **Iterator.**

La interface **Iterator** proporciona una manera fija de recorrer elementos en distintas estructuras de datos. El método **next()** proporciona acceso secuencial a los elementos. Podemos usar el método **hasNext()** para comprobar si hay más elementos en el iterador, y el método **remove()** para eliminar el último elemento devuelto por el iterador.

1. Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su **efectividad** resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser **reutilizable**, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Diagrama UML

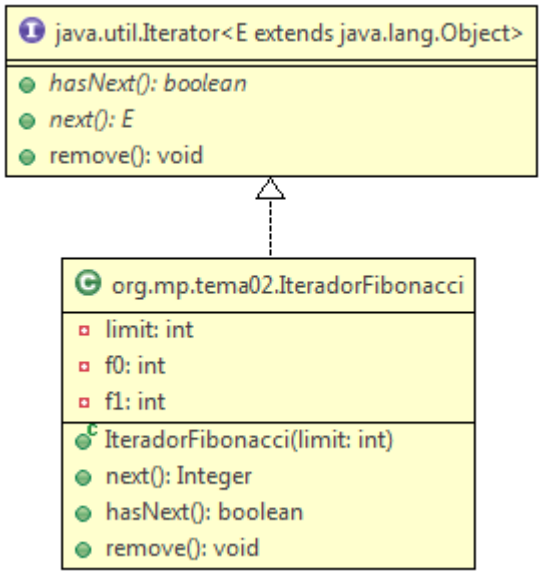


Devuelve un iterador para los elementos

Devuelve true si el iterador tiene más elementos que recorrer
Devuelve el siguiente elemento del iterador
Elimina el último elemento obtenido usando el método next()

Ejemplo de uso

Define una clase **IteradorFibonacci** para iterar números de Fibonacci. El constructor tiene un argumento que especifica el límite del máximo número de Fibonacci. Por ejemplo, new IteradorFibonacci(1000) crea un iterador que itera números de Fibonacci menores o iguales que 1000.



```java
package org.mp.tema02;

/*(Iterador números Fibonacci) Define una clase iterador llamada IteradorFibonacci
 para iterar números de Fibonacci. El constructor recibe un argumento que
 especifica el límite del máximo número de Fibonacci. Por ejemplo, new
 IteradorFibonacci(23302) crea un iterador que itera números de Fibonacci
 menores o iguales que 23302.
 *
 */

import java.util.Iterator;

public class IteradorFibonacci implements Iterator<Integer> {
    private int limit;
    private int f0 = 0;
    private int f1 = 1;

    public IteradorFibonacci(int limit) {
        this.limit = limit;
    }

    @Override
    public Integer next() {
        int temp = f0;
        f0 = f1;
        f1 = temp + f1;
        return temp;
    }

    @Override
    public boolean hasNext() {
        if (f0 > limit)
            return false;
        else
            return true;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException("Método no soportado");
    }
}
```

```java
package org.mp.tema02;

import java.util.Iterator;

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

        int contador = 1;
        int limite = 1000000;
        Iterator<Integer> iterator = new IteradorFibonacci(limite);

        System.out.println("SERIE DE FIBONACCI HASTA "+limite);
        System.out.println();
        while (iterator.hasNext()) {
            if (contador % 10 == 0) {
                System.out.printf("%8d\n",iterator.next());
            } else {
                System.out.printf("%8d",iterator.next());
            }
            contador++;
        }
    }
}
```

Salida

```
SERIE DE FIBONACCI HASTA 1000000

       0       1       1       2       3       5       8      13      21      34
      55      89     144     233     377     610     987    1597    2584    4181
    6765   10946   17711   28657   46368   75025  121393  196418  317811  514229
  832040
```

Interface Comparator

Se usa para comparar objetos de clases que no implementan la interface Comparable.

La interface **Comparable** define el método compareTo() el cual se usa para comparar dos elementos de una **misma clase** que implementa la interface.

Definiremos un *comparador* para comparar elementos de **clases distintas.** Para hacer esto, definiremos la clase que implemente la interface j**ava.util.Comparator<T>** y que sobre-escriba el método **compare.**

public int compare(T elemento1, T elemento2)

Este método devuelve:

un valor negativo si elemento1 < elemento2

un valor positivo si elemento1 > elemento2

y 0 si elemento1 = elemento2

Además, esta interface tiene el método equals

public boolean equals(Object elemento)

Este método devuelve:

verdadero si el objeto especificado es también un comparador e impone el mismo orden que este comparador.

falso en caso contrario

Ejemplo de uso

```java
package org.mp.tema02;

import java.util.Comparator;

public class ObjetoGeometricoComparador implements Comparator<ObjetoGeometrico>{
    public int compare(ObjetoGeometrico o1, ObjetoGeometrico o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();

        if (area1 < area2)
          return -1;
        else if (area1 == area2)
          return 0;
        else
          return 1;
      }
}
```

Salida

```
El área mayor de los objetos es: 78.53981633974483
```

```java
package org.mp.tema02;

import java.util.Comparator;

public class TestComparador {
  public static void main(String[] args) {
    ObjetoGeometrico g1 = new Rectangulo(5, 5);
    ObjetoGeometrico g2 = new Circulo(5);

    ObjetoGeometrico g =
      max(g1, g2, new ObjetoGeometricoComparador());

    System.out.println();
    System.out.println("La mayor área de los objetos es: " +
      g.getArea());
  }

  public static ObjetoGeometrico max(ObjetoGeometrico g1,
      ObjetoGeometrico g2, Comparator<ObjetoGeometrico> c) {
    if (c.compare(g1, g2) > 0)
      return g1;
    else
      return g2;
  }
}
```

Interfaces vs. Clases abstractas

En una interface, los datos deben ser constantes, una clase abstracta puede tener todo tipo de datos.

Cada método en una interface solo puede tener la cabecera sin implementación, una clase abstracta puede tener métodos concretos.

	Variables	Constructores	Métodos
Clases Abstractas	No hay restricciones	Los constructores se invocan en las subclases a través de encadenamiento de constructores. Una clase abstracta no puede ser instanciada usando el operador new	No hay restricciones
Interfaces	Todas las variables deben ser **public**, **static** y **final**	No tiene constructores. Una interface no puede ser instanciada usando el operador new.	Todos los métodos son de instancia y **public, abstract**

Java permite solo *herencia simple* para extensión de clases pero permite la implementación de *múltiples interfaces*. Por ejemplo:

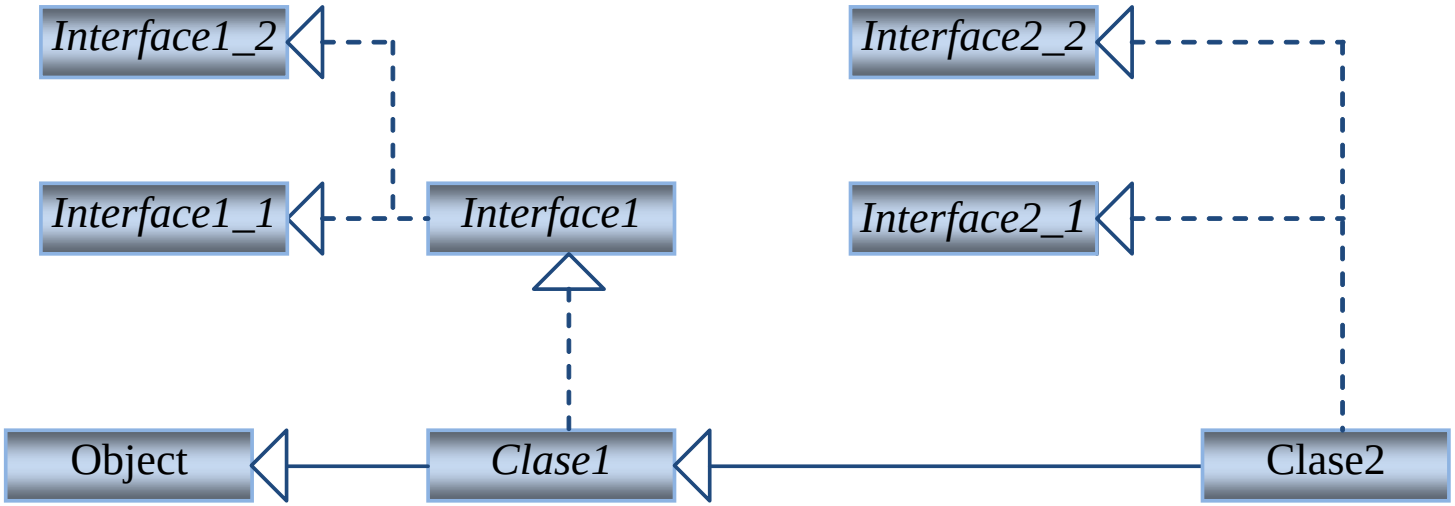
```
public class NuevaClase extends ClaseBase
        implements Interface1, ..., InterfaceN {
            ...
}
```

Una interface puede heredar de otras interfaces usando la palabra reservada **extends.** En este caso la interface se llama *subinterface*. Por ejemplo, **NuevaInterface** en el siguiente código es una subinterface de **Interface1**, **Interface2**,…, y **InterfaceN**.

```
public interface NuevaInterface extends Interface1, ... , InterfaceN {
        // constantes y métodos abstractos
}
```

Una clase que implemente **NuevaInterface** debe implementar los métodos abstractos definidos en **NuevaInterface**, **Interface1**, **Interface2**,…,**InterfaceN**. Una interface puede extender otras interfaces pero no clases. Una clase puede extender su superclase e implementar múltiples interfaces.

Todas las clases comparten una raíz única, la clase **Object**, pero no tienen una única raíz para las interfaces. Igual que una clase, una interface define también un tipo. Una variable de un tipo interface puede referenciar cualquier instancia de la clase que implementa la interface. Si una clase implementa una interface, la interface es como una superclase para la clase. Usaremos una **interface como un tipo de dato** y castear una variable de un tipo interface a sus subclases y viceversa. Por ejemplo, supongamos que **c** es una instancia de la **Clase2** tal y como se ve en la figura. **c** es también una instancia de **Object**, **Clase1**, **Interface1**, **Interface1_1**, **Interface1_2, Interface2_1**, e **Interface2_2**.



Convenciones de nombrado

Las clases se nombran con sustantivos. Los nombres de las interfaces pueden ser adjetivos o sustantivos.

¿Cómo decidir cuando usar una interface o una clase?

Las clases abstractas y las interfaces ambas se usan para especificar comportamientos comunes de objetos.

En general, una relación **fuerte es -un** que describe claramente una relación padre-hijo debe modelarse utilizando clases. Por ejemplo, el calendario Gregoriano es un calendario, por lo que la relación entre la clase **GregorianCalendar** y la clase **Calendar** se modela usando la herencia de clases. Una relación **débil es-un**, indica que un objeto posee ciertas propiedades. Esta puede ser modelada utilizando interfaces. Por ejemplo, todos los string son comparables, por tanto la clase **String** implementa la interface **Comparable**.

En general, las interfaces se prefieren sobre las clases abstractas porque una interface puede definir un supertipo común para clases no relacionadas. Las interfaces son más flexibles que las clases.

Guía en el diseño de clases

➢ Cohesión

Una clase debe describir una única entidad y todas las operaciones de la clase deben de encajar lógicamente para apoyar un propósito coherente. Se puede usar una clase para estudiantes, por ejemplo, pero no combinaríamos en la misma clase empleados y estudiantes porque son entidades diferentes. Una entidad única con muchas responsabilidades puede dividirse en distintas clases para separar responsabilidades. Las clases **String**, **StringBuffer**, tratan ambas con cadenas pero tienen diferentes responsabilidades. La clase **String** trata con strings inmutables y la clase **StringBuffer** crea strings mutables.

➢ Consistencia

Seguir el estilo de programación estándar de Java y las convenciones de nombrado. Elegir nombres representativos para las clases, propiedades y métodos. Un estilo habitual es colocar las propiedades antes que los constructores y estos antes que los métodos.

Poner nombres consistentes. No es una buena práctica, elegir diferentes nombres para una misma operación. Por ejemplo, el método **length()** devuelve el tamaño de un objeto de la clase **String**, **StringBuffer** y **StringBuilder**. Sería inconsistente si se usaran diferentes nombres para ese método en esas clases.

- **Encapsulación**

Una clase usa el modificador **private** para ocultar los datos y evitar un acceso directo por parte de los usuarios. Esto hace que la clase sea fácil de mantener.

Se proporciona el método getter si queremos que la propiedad se pueda leer y el método setter solo si queremos que la propiedad se pueda cambiar.

- **Claridad**

La cohesión, consistencia y encapsulación son buenas guías para lograr claridad en el diseño. Además, una clase debe tener una declaración clara que sea fácil de explicar y de entender. Los usuarios deben poder incorporar clases en diferentes estados y entorno. Por lo tanto, se deberá diseñar una clase que no imponga restricciones de cómo o cuando puede usarse, diseñar las propiedades de manera que permita al usuario ponerlas en cualquier orden y con cualquier combinación de valores y diseñar los métodos que funcionen independientemente del orden de aparición. Los métodos deben definirse intuitivamente sin causar confusión.

Por ejemplo, el método **substring(int beginIndex, int endIndex)** de la clase String es algo confuso.

- **Completitud**

Las clases se diseñan para que las utilicen distintos clientes. Con el propósito de que puedan usarse en un amplio abanico de aplicaciones, una clase debe proporcionar un diseño, de sus propiedades y métodos, que sea útil para diferentes clientes. Por ejemplo, la clase **String** contiene más de 40 métodos que son utilizables en una gran variedad de aplicaciones.

- **Variable de instancia vs. Variable estática**

Una variable o un método que son dependientes de una instancia específica de clase debe ser una variable o método de instancia. Una variable que es compartida por todas las instancias de una clase debe ser declarada estática.

Tanto las variables de instancia como las estáticas son partes que integran la POO. Un miembro dato o un método es de instancia o estático. Es un error común en el diseño definir un método de instancia que debería ser estático. Por ejemplo, el método **factorial (int n)** que calcula el factorial de n, debe definirse como estático porque es independiente de cualquier instancia específica.

Un constructor es siempre de instancia porque se usa para crear una instancia concreta. Una variable o método estático pueden invocarse desde un método de instancia, pero una variable o método de instancia no puede invocarse desde un método estático.

➢ Herencia vs. Agregación

La diferencia entre herencia y agregación es la diferencia entre una relación **es-un** y **tiene-un**.

Por ejemplo, una manzana *es una* fruta, por tanto utilizaremos la herencia para modelar la relación entre las clases **Manzana** y **Fruta**.

Una persona *tiene un* nombre, así usaremos la agregación para modelar la relación entre las clases **Persona** y **Nombre**.

➢ Interfaces vs. Clases abstractas

Ambas se utilizan para especificar características comunes de objetos.

¿Cuándo usar una u otra? En general, una relación **es-un fuerte** que claramente describe una relación padre-hijo debe modelarse usando **clases**. Por ejemplo, una naranja es una fruta, su relación debe modelarse usando herencia de clases. Una relación **es-un débil**, indica que un objeto posee ciertas propiedades. Esta relación debe modelarse usando **interfaces**. Por ejemplo, todos los strings son comparables, entonces la clase **String** implementa la interface **Comparable**. Un círculo o un rectángulo son objetos geométricos, entonces **Circulo** debe diseñarse como una subclase de **ObjetoGeométrico**. Los círculos son diferentes y comparables según su radio, por tanto **Circulo** debe implementar la interface **Comparable**.

Las interfaces son más flexibles que las clases abstractas porque una subclase puede extender solo una superclase pero puede implementar cualquier número de interfaces. Por otro lado, las interfaces no pueden contener métodos implementados. Lo ideal es combinar las interfaces y las clases abstractas para aprovechar las virtudes de ambas y usar una u otra según sea más conveniente. Esto será lo que hagamos en los temas siguientes en los que diseñaremos los tipos de datos Listas, Pila, Colas y Colas de Prioridad.

Resumen

- Los genéricos nos dan la capacidad de parametrizar tipos. Podemos definir una clase o un método con tipos genéricos los cuales sustituye el compilador por un tipo concreto.

- El beneficio de los genéricos es que permiten detectar errores en tiempo de compilación mejor que en tiempo de ejecución.

- Una clase o un método genérico nos permite especificar los tipos de objetos con los que la clase o método puede trabajar. Si intentamos usar clases o métodos con tipos de objetos incompatibles, el compilador detecta el error.

- Un tipo genérico definido en una clase, interface o método estático se llama *tipo genérico formal* el cual puede sustituirse después por un *tipo concreto actual.* Sustituir un tipo genérico se llama *instanciación genérica.*

- Una clase genérica como **ArrayList** usada sin un tipo parametrizado se llama *raw type*

- Los genéricos se implementan usando una aproximación llamada *type erasure*. El compilador usa la información del tipo genérico para compilar el código pero la borra después, por tanto, la información no está disponible en tiempo de ejecución. Esta aproximación permite que el código con genéricos sea compatible hacia atrás con el código heredado que usa tipos raw.

- No podemos crear una instancia usando tipos genéricos parametrizados.

- No podemos crear un array usando un tipo genérico parametrizado.

- No podemos usar un tipo genérico parametrizado en un contexto estático.

- Los tipos genéricos parametrizados no se pueden usar en clases de excepciones.

- Las clases abstractas son clases con datos y métodos, pero no se pueden crear instancias de clases abstractas usando el operador **new**.

- Un método abstracto no puede estar en una clase que no sea abstracta. Si una subclase de una superclase abstracta no implementa todos los métodos abstractos heredados de la superclase, la subclase debe ser definida como abstracta.

- Una clase que contiene métodos abstractos debe ser abstracta. Sin embargo, es posible definir una clase abstracta que no contenga ningún método abstracto.

- Una subclase puede ser abstracta incluso si su superclase es concreta.

- Una interface es como una clase que contiene solo constantes y métodos abstractos. En cierto modo, una interface es similar a una clase abstracta pero esta última puede contener variables y métodos implementados.

- Una interface es tratada como una clase especial en Java. Cada interface se compila en un archivo bytecode separado igual que una clase regular.

- La interface **java.lang.Comparable** define el método **compareTo**. Muchas clases de la librería de Java la implementan.

- La interface **java.lang.Cloneable** es una interface marcadora. Un objeto de una clase que implemente la interface **Cloneable** es clonable.

- La interface **java.util.Comparator** se usa para comparar objetos de clases que no implementan la interface **Comparable**.

- Utilizamos la interface **java.lang.Iterable** y **java.util.Iterator** para recorrer los elementos de una secuencia.

- Una clase puede heredar de una sola clase pero puede implementar una o más interfaces.

- Una interface puede heredar una o más interfaces.

¡MUCHAS GRACIAS!

UNIVERSIDAD DE ALMERÍA