Complementos de Programación

Tema 3. Herencia y polimorfismo

Andrés Cano Utrera
Departamento de Ciencias de la Computación e I.A.

DECSAI

ugr Universidad de Granada

ETSIIT Universidad de Granada

Curso 2015-16

Índice I

Índice II

Contenido del tema

Motivación

Objetivo del tema

Completar los conceptos básicos de orientación a objetos usando el lenguaje de programación Java: herencia, polimorfismo, ligadura dinámica.

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces
- Concepto de downcasting y upcasting

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces
- Concepto de downcasting y upcasting
- Completar el conocimiento del control de acceso a miembros de una clase

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces
- Concepto de downcasting y upcasting
- Completar el conocimiento del control de acceso a miembros de una clase
- Organizar clases en paquetes

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces
- Concepto de downcasting y upcasting
- Completar el conocimiento del control de acceso a miembros de una clase
- Organizar clases en paquetes
- Definición de clases internas

Motivación (II)

Los conceptos básicos a comprender son:

- Construcción y uso de jerarquías de clases
- Uso de super para llamar a constructores de la superclase
- Sobreescritura de métodos en subclases
- Construir algoritmos genéricos usando polimorfismo, ligadura dinámica, clases abstractas e interfaces
- Concepto de downcasting y upcasting
- Completar el conocimiento del control de acceso a miembros de una clase
- Organizar clases en paquetes
- Definición de clases internas
- Aprender la notación UML para definir jerarquías de clases

Contenido del tema

Introducción

Definición

La herencia permite la definición de nuevas clases a partir de otras clases

Introducción

Definición

La herencia permite la definición de nuevas clases a partir de otras clases

- La herencia es uno de los pilares de la PDO.
- Permite definir una clase general que define características comunes a un conjunto de elementos relacionados.
- Permite la creación de clasificaciones jerárquicas.
- Cada subclase puede añadir aquellas cosas particulares a ella.
- Las subclases heredan todas las variables de instancia y los métodos definidos por la superclase, y luego pueden añadir sus propios elementos.

Introducción

- Para definir que una subclase hereda de otra clase usamos **extends**.
- Java no permite la herencia múltiple.
- La subclase no puede acceder a aquellos miembros declarados como **private** en la superclase.

Contenido del tema

Ejemplo de herencia I

```
1  // Este programa utiliza herencia para extender la clase Box.
2  class Caja {
3    double ancho;
4    double alto;
5    double largo;
6    Caja(Caja ob) {
7      ancho = ob.ancho;
8      alto = ob.alto;
9      largo = ob.largo;
10   }
11   Caja(double w, double h, double d) {
12     ancho = w;
13     alto = h;
14     largo = d;
15   }
16   Caja() {
17     ancho = -1;
18     alto = -1;
19     largo = -1;
20   }
21   Caja(double len) {
22     ancho = alto = largo = len;
23   }
24   double volumen() {
25     return ancho * alto * largo;
26   }
27 }
28
```

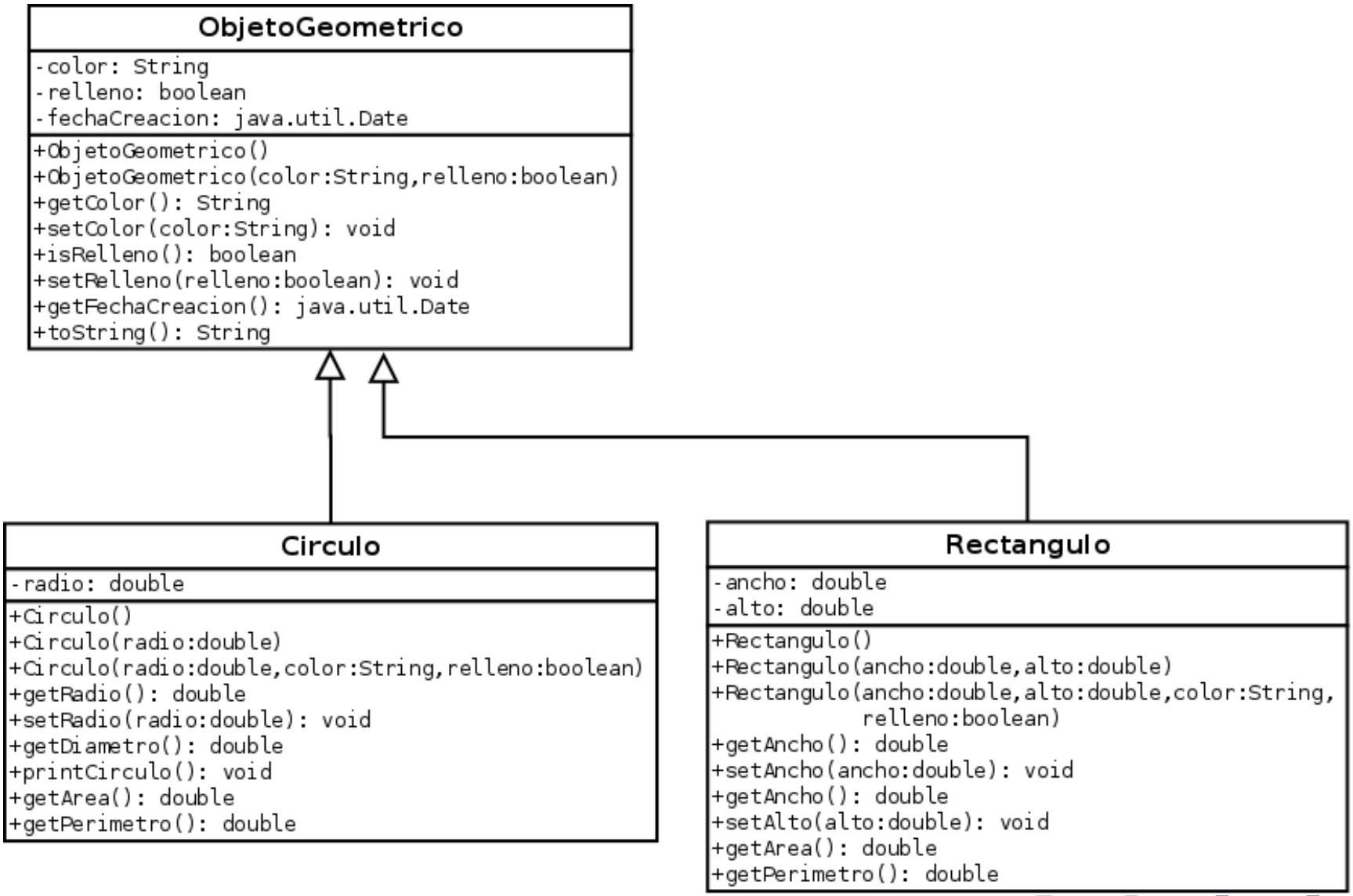
Ejemplo de herencia II

```
29 class CajaPeso extends Caja {
30   double peso; // peso de la caja
31   CajaPeso(double w, double h, double d, double m) {
32     ancho = w;
33     alto = h;
34     largo = d;
35     peso = m;
36   }
37 }
38
39
40 class DemoCajaPeso {
41   public static void main(String args[]) {
42     CajaPeso micaja1 = new CajaPeso(10, 20, 15, 34.3);
43     CajaPeso micaja2 = new CajaPeso(2, 3, 4, 0.076);
44     double vol;
45     vol = micaja1.volumen();
46     System.out.println("Volumen de micaja1 es " + vol);
47     System.out.println("Peso de micaja1 es " + micaja1.peso);
48     System.out.println();
49     vol = micaja2.volumen();
50     System.out.println("Volumen de micaja2 es " + vol);
51     System.out.println("Peso de micaja2 es " + micaja2.peso);
52   }
53 }
```

Ejemplo de herencia III

```
Volumen de micaja1 es 3000.0
Peso de micaja1 es 34.3

Volumen de micaja2 es 24.0
Peso de micaja2 es 0.076
```

Otro ejemplo de herencia



```
public class ObjetoGeometrico {
  private String color = "blanco";
  private boolean relleno;
  private java.util.Date fechaCreacion;

  public ObjetoGeometrico() {
    fechaCreacion = new java.util.Date();
  }
  public ObjetoGeometrico(String color, boolean relleno) {
    fechaCreacion = new java.util.Date();
    this.color = color;
    this.relleno = relleno;
  }
  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }
  public boolean isRelleno() {
    return relleno;
  }
  public void setRelleno(boolean relleno) {
    this.relleno = relleno;
  }
  public java.util.Date getFechaCreacion() {
    return fechaCreacion;
  }
  public String toString() {
    return "creado el " + fechaCreacion + "\ncolor: " + color +    " y relleno: " + relleno;
  }
}
```

```
public class Rectangulo extends ObjetoGeometrico {
  private double ancho;
  private double alto;
  public Rectangulo() {
  }
  public Rectangulo(double ancho, double alto) {
    this.ancho = ancho;
    this.alto = alto;
  }
  public Rectangulo(double ancho, double alto, String color, boolean relleno) {
    this.ancho = ancho;
    this.alto = alto;
    setColor(color);
    setRelleno(relleno);
  }
  public double getAncho() {
    return ancho;
  }
  public void setAncho(double ancho) {
    this.ancho = ancho;
  }
  public double getAlto() {
    return alto;
  }
  public void setAlto(double alto) {
    this.alto = alto;
  }
  public double getArea() {
    return ancho * alto;
  }
  public double getPerimetro() {
    return 2 * (ancho + alto);
  }
}
```

```java
public class Circulo extends ObjetoGeometrico {
  private double radio;

  public Circulo() {
  }
  public Circulo(double radio) {
    this.radio = radio;
  }
  public Circulo(double radio,  String color, boolean relleno) {
    this.radio = radio;
    setColor(color);
    setRelleno(relleno);
  }
  public double getRadio() {
    return radio;
  }
  public void setRadio(double radio) {
    this.radio = radio;
  }
  public double getArea() {
    return radio * radio * Math.PI;
  }
  public double getDiametro() {
    return 2 * radio;
  }
  public double getPerimetro() {
    return 2 * radio * Math.PI;
  }
  public void printCirculo() {
    System.out.println("El circulo fue creado el " + getFechaCreacion() +
      " y el radio es " + radio);
  }
}
```

```
public class TestCirculoRectangulo {
  public static void main(String[] args) {
    Circulo circle = new Circulo(1);
    System.out.println("Un circulo " + circle.toString());
    System.out.println("El color es " + circle.getColor());
    System.out.println("El radio es " + circle.getRadio());
    System.out.println("El area es " + circle.getArea());
    System.out.println("El diametro es " + circle.getDiametro());

    Rectangulo rectangle = new Rectangulo(2, 4);
    System.out.println("\nUn rectangulo " + rectangle.toString());
    System.out.println("El area es " + rectangle.getArea());
    System.out.println("El perimetro es " +
      rectangle.getPerimetro());
  }
}
```

```java
public class TestCirculoRectangulo {
  public static void main(String[] args) {
    Circulo circle = new Circulo(1);
    System.out.println("Un circulo " + circle.toString());
    System.out.println("El color es " + circle.getColor());
    System.out.println("El radio es " + circle.getRadio());
    System.out.println("El area es " + circle.getArea());
    System.out.println("El diametro es " + circle.getDiametro());

    Rectangulo rectangle = new Rectangulo(2, 4);
    System.out.println("\nUn rectangulo " + rectangle.toString());
    System.out.println("El area es " + rectangle.getArea());
    System.out.println("El perimetro es " +
      rectangle.getPerimetro());
  }
}
```

```
Un circulo creado el Tue Mar 26 13:48:38 CET 2013
color: blanco y relleno: false
El color es blanco
El radio es 1.0
El area es 3.141592653589793
El diametro es 2.0

Un rectangulo creado el Tue Mar 26 13:48:38 CET 2013
color: blanco y relleno: false
El area es 8.0
El perimetro es 12.0
```

Contenido del tema

Uso de super

super

La palabra reservada **super** permite a una subclase referenciar a su superclase inmediata. Es utilizada en las siguientes situaciones:

1. Para llamar al constructor de la superclase desde el constructor de la subclase.

   En este caso `super()` *debe ser la primera sentencia ejecutada dentro del constructor.*

2. Para acceder a un miembro (dato o método) de la superclase que ha sido ocultado por un miembro de la subclase.

Uso de super

Ejemplo 1

Ejemplo de super para llamar al constructor de la superclase

```
class CajaPeso extends Caja {
  double peso;

  CajaPeso(double w, double h, double d, double m) {
    super(w, h, d); // llama al constructor de la superclase
    peso = m;
  }
  CajaPeso(CajaPeso ob) {
    super(ob);
    peso = ob.peso;
  }
}
```

Uso de super

Ejemplo 2

Utilización de super para acceder a un miembro de la superclase

```
1  // Utilización de super para evitar la ocultación de nombres
2  class A {
3    int i;
4  }
5  class B extends A {
6    int i; // esta i oculta la i de A
7
8    B(int a, int b) {
9      super.i = a; // i in A
10     i = b; // i in B
11   }
12   void show() {
13     System.out.println("i en la superclase: " + super.i);
14     System.out.println("i en la subclase: " + i);
15   }
16 }
17 class UseSuper {
18   public static void main(String args[]) {
19     B subOb = new B(1, 2);
20     subOb.show();
21   }
22 }
```

Contenido del tema

Orden de ejecución de constructores

Ejecución de los constructores en la jerarquía de clases

Un constructor puede llamar explícitamente a un constructor de la superclase. Si no lo hace, el compilador coloca `super()` como primera sentencia del constructor.

- Esto hace que al llamar al constructor de una clase, se llamen a todos los constructores de las superclases, siguiendo el orden dado por la cadena de la herencia: *constructor chaining*

Ejemplo de orden de constructores

```
1  // Muestra cuando se ejecutan los constructores.
2  class A {
3    A() {
4      System.out.println("En el constructor de A.");
5    }
6  }
7  class B extends A {
8    B() {
9      System.out.println("En el constructor de B.");
10   }
11 }
12 class C extends B {
13   C() {
14     System.out.println("En el contructor de C.");
15   }
16 }
17 class CallingCons {
18   public static void main(String args[]) {
19     C c = new C();
20   }
21 }
```

Ejemplo de orden de constructores

```
1  // Muestra cuando se ejecutan los constructores.
2  class A {
3    A() {
4      System.out.println("En el constructor de A.");
5    }
6  }
7  class B extends A {
8    B() {
9      System.out.println("En el constructor de B.");
10   }
11 }
12 class C extends B {
13   C() {
14     System.out.println("En el contructor de C.");
15   }
16 }
17 class CallingCons {
18   public static void main(String args[]) {
19     C c = new C();
20   }
21 }
```

```
En el constructor de A.
En el constructor de B.
En el contructor de C.
```

Otro ejemplo de orden de constructores

```
1  public class Profesor extends Empleado {
2    public static void main(String[] args) {
3      new Profesor();
4    }
5
6    public Profesor() {
7      System.out.println("(4) Hacer tareas de Profesor");
8    }
9  }
10
11  class Empleado extends Persona {
12    public Empleado() {
13      this("(2) Llamar al constructor sobrecargado de Empleado");
14      System.out.println("(3) Hacer tareas de Empleado ");
15    }
16    public Empleado(String s) {
17      System.out.println(s);
18    }
19  }
20
21 class Persona {
22   public Persona() {
23     System.out.println("(1) Hacer tareas de Persona");
24   }
25 }
26
```

Otro ejemplo de orden de constructores

```
1  public class Profesor extends Empleado {
2    public static void main(String[] args) {
3      new Profesor();
4    }
5
6    public Profesor() {
7      System.out.println("(4) Hacer tareas de Profesor");
8    }
9  }
10
11  class Empleado extends Persona {
12    public Empleado() {
13      this("(2) Llamar al constructor sobrecargado de Empleado");
14      System.out.println("(3) Hacer tareas de Empleado ");
15    }
16    public Empleado(String s) {
17      System.out.println(s);
18    }
19  }
20
21 class Persona {
22    public Persona() {
23      System.out.println("(1) Hacer tareas de Persona");
24    }
25 }
26
```

```
(1) Hacer tareas de Persona
(2) Llamar al constructor sobrecargado de Empleado
(3) Hacer tareas de Empleado
(4) Hacer tareas de Profesor
```

Orden de ejecución de constructores

Cuidado

Si una clase se diseña para ser extendida, debe proporcionarse un constructor sin argumentos para evitar errores de compilación

```
public class Manzana extends Fruta{
}

class Fruta {
  public Fruta(String name) {
    System.out.println("Se llama al constructor de Fruta");
  }
}
```

Orden de ejecución de constructores

Cuidado

Si una clase se diseña para ser extendida, debe proporcionarse un constructor sin argumentos para evitar errores de compilación

```
public class Manzana extends Fruta{
}

class Fruta {
  public Fruta(String name) {
    System.out.println("Se llama al constructor de Fruta");
  }
}
```

```
Manzana.java:1: error: constructor Fruta in class Fruta cannot be applied to given types;
public class Manzana extends Fruta
       ^
  required: String
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

Contenido del tema

Sobreescritura de métodos (Overriding)

Sobreescritura de método

Consiste en construir un método en una subclase con el mismo nombre, parámetros y tipo devuelto, que otro método de una de sus superclases (inmediata o no).

- Usaremos este mecanismo cuando la subclase necesite modificar la implementación de un método definido en la superclase.

Sobreescritura de métodos (Overriding)

```
1  class A {
2    int i, j;
3    A(int a, int b) {
4      i = a;
5      j = b;
6    }
7    void show() {
8      System.out.println("i y j: " + i + " " + j);
9    }
10 }
11 class B extends A {
12   int k;
13   B(int a, int b, int c) {
14     super(a, b);
15     k = c;
16   }
17   // muestra k -- sobreescribe el metodo show() de A
18   void show() {
19     System.out.println("k: " + k);
20   }
21 }
22 class Override {
23   public static void main(String args[]) {
24     B subOb = new B(1, 2, 3);
25     subOb.show(); // llama al metodo show() de B
26   }
27 }
```

Sobreescritura de métodos (Overriding)

```
1  class A {
2    int i, j;
3    A(int a, int b) {
4      i = a;
5      j = b;
6    }
7    void show() {
8      System.out.println("i y j: " + i + " " + j);
9    }
10 }
11 class B extends A {
12   int k;
13   B(int a, int b, int c) {
14     super(a, b);
15     k = c;
16   }
17   // muestra k -- sobreescribe el metodo show() de A
18   void show() {
19     System.out.println("k: " + k);
20   }
21 }
22 class Override {
23   public static void main(String args[]) {
24     B subOb = new B(1, 2, 3);
25     subOb.show(); // llama al metodo show() de B
26   }
27 }
```

```
k: 3
```

Sobreescritura de métodos (Overriding)

Desde un método sobreescrito podemos llamar al de la superclase usando `super.metodo`.

```
public class Circulo extends ObjetoGeometrico {
  // ...
  // Sobreescritura del metodo toString definido en la superclase
  public String toString() {
    return super.toString() + "\nradio es " + radio;
  }
}
```

Es un error de compilación intentar usar `super.super.metodo` (para intentar ejecutar el método de la clase abuela).

Algunos puntos a tener en cuenta en la sobreescritura de métodos

- Un método de instancia sólo puede sobreescribirse si es accesible.
  - De esta forma, un método **private** de la superclase no puede sobreescribirse.
  - Si se redefine en la subclase, entonces el método de la superclase y el de la subclase, serán métodos *no relacionados*.

Algunos puntos a tener en cuenta en la sobreescritura de métodos

- Un método de instancia sólo puede sobreescribirse si es accesible.
  - De esta forma, un método **private** de la superclase no puede sobreescribirse.
  - Si se redefine en la subclase, entonces el método de la superclase y el de la subclase, serán métodos *no relacionados*.
- Un método **static** no puede sobreescribirse.
  - Si se redefine un método **static** de la superclase, éste queda oculto.
  - El método oculto así, se podría utilizar con `NombreSuperclase.metodoEstatico`

Contenido del tema

Clase Object

Todas las clases de java, heredan directa o indirectamente de la clase **Object**

Esta clase tiene 11 métodos que son heredados (o sobreescritos) por las subclases:

- `Object boolean equals()`
- `int hashCode()`
- `String toString()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`
- `void notify()`
- `void notifyAll()`
- `Class<?> getClass()`
- `protected void finalize()`
- `protected Object clone()`

Método toString()

La llamada a este método sobre un objeto devuelve a string que describe el objeto.

- Por defecto, devuelve un string con el nombre de la clase a la que pertenece el objeto, un signo `@` y la dirección de memoria del objeto en hexadecimal.

```
Circulo circulo = new Circulo();
System.out.println(circulo.toString());
System.out.println(circulo); // equivalente al anterior
```

Método toString()

La llamada a este método sobre un objeto devuelve a string que describe el objeto.

- Por defecto, devuelve un string con el nombre de la clase a la que pertenece el objeto, un signo `@` y la dirección de memoria del objeto en hexadecimal.

```
Circulo circulo = new Circulo();
System.out.println(circulo.toString());
System.out.println(circulo); // equivalente al anterior
```

```
Circulo@15027e5
Circulo@15027e5
```

Método toString()

Debemos sobreescribir este método en nuestras clases si queremos una descripción más informativa.

```
public class Circulo extends ObjetoGeometrico {
  // ...
  // Sobreescritura del metodo toString definido en la superclase
  public String toString() {
    return "Radio:  " + radio + "; Centro:  " + centro.toString();
  }
}
```

Método toString()

Debemos sobreescribir este método en nuestras clases si queremos una descripción más informativa.

```
public class Circulo extends ObjetoGeometrico {
  // ...
  // Sobreescritura del metodo toString definido en la superclase
  public String toString() {
    return "Radio:   " + radio + "; Centro:   " + centro.toString();
  }
}
```

```
Radio: 7; Centro: 5, 8
```

Contenido del tema

Polimorfismo

Una variable de la superclase puede referenciar a un objeto de una subclase

```
public class ObjetoGeometrico {
  private String color = "blanco";
  ...
  public String getColor() {
    return color;
  }
  ...
}
public class Circulo extends ObjetoGeometrico {
  private double radio;
  ...
  public double getRadio() {
    return radio;
  }
  ...
}
public class TestCirculo {
  public static void main(String[] args) {
    Circulo c = new Circulo(10);
    ObjetoGeometrico figura;

    figura=c;
    System.out.println("Color del circulo: "+figura.getColor());
    //System.out.println("Radio del circulo: "+figura.getRadio()); Error de compilacion
  }
}
```

Polimorfismo

- La herencia permite definir una clase con características adicionales a las que hereda de la superclase.
- Una subclase es una *especialización* de su superclase: cada instancia de la subclase es también una instancia de la superclase, pero no al revés.

  Por ejemplo, un `Circulo` ***es un*** `ObjetoGeometrico`
  pero no todo `ObjetoGeometrico` ***es un*** `Circulo`*.*

- Por tanto, podremos usar una referencia de la superclase, para apuntar a un objeto de la subclase.
- El tipo de la variable referencia, (y no el del objeto al que apunta) es el que determina los miembros que son accesibles.

Ejemplo

```
class RefDemo {
  public static void main(String args[]) {
    CajaPeso cajaconpeso = new CajaPeso(3, 5, 7, 8.37);
    Caja cajanormal = new Caja();
    double vol;

    vol = cajaconpeso.volumen();
    System.out.println("El volumen de cajaconpeso es " + vol);
    System.out.println("El peso de cajaconpeso es " +
                       cajaconpeso.peso);
    System.out.println();

    // asigna una referencia de CajaPeso a una referencia de Caja
    cajanormal = cajaconpeso;
    vol = cajanormal.volumen(); // OK,  volume() definido en Caja
    System.out.println("Volumen de cajanormal es " + vol);

    /* La siguiente sentencia no es valida porque cajanormal
       no define un miembro llamado peso. */
    //  System.out.println("El peso de cajanormal es " +
                         // cajanormal.peso);
  }
}
```

Ejemplo

```
class RefDemo {
  public static void main(String args[]) {
    CajaPeso cajaconpeso = new CajaPeso(3, 5, 7, 8.37);
    Caja cajanormal = new Caja();
    double vol;

    vol = cajaconpeso.volumen();
    System.out.println("El volumen de cajaconpeso es " + vol);
    System.out.println("El peso de cajaconpeso es " +
                       cajaconpeso.peso);
    System.out.println();

    // asigna una referencia de CajaPeso a una referencia de Caja
    cajanormal = cajaconpeso;
    vol = cajanormal.volumen(); // OK,  volume() definido en Caja
    System.out.println("Volumen de cajanormal es " + vol);

    /* La siguiente sentencia no es valida porque cajanormal
       no define un miembro llamado peso. */
    //  System.out.println("El peso de cajanormal es " +
                         // cajanormal.peso);
  }
}
```

```
El volumen de cajaconpeso es 105.0
El peso de cajaconpeso es 8.37

Volumen de cajanormal es 105.0
```

Contenido del tema

Ligadura dinámica

Ligadura dinámica

Es el mecanismo mediante el cual una llamada a un método sobreescrito se resuelve en **tiempo de ejecución** en lugar de en tiempo de compilación: **polimorfismo en tiempo de ejecución**

- Cuando un método sobreescrito se llama a través de una referencia de la superclase, Java determina la versión del método que debe ejecutar en función del objeto que está siendo referenciado.

Ejemplo

```
public class ObjetoGeometrico {
   ...
   public String toString() {
     return "creado el " + fechaCreacion + "\ncolor: " + color +
            " y relleno: " + relleno;
   }
}
public class Circulo extends ObjetoGeometrico {
   ...
   public String toString() {
     return super.toString() + "\n    radio es " + radio;
   }
   ...
}
public class Rectangulo {
   ...
   public String toString() {
     return super.toString() + "\n    anchura es " + ancho
            + "\n    altura es " + alto;
   }
   ...
}
```

Ejemplo

```
public class TestCirculo {
  public static void main(String[] args) {
    Circulo c = new Circulo(10);
    ObjetoGeometrico figura;

    figura=c;
    System.out.println(figura.toString());
  }
}
```

```
creado el Thu Mar 31 08:58:36 CEST 2016
color: blanco y relleno: false
    radio es 10.0
```

Otro ejemplo

```
public class TestCirculoRectangulo {
  public static void main(String[] args) {
    ObjetoGeometrico[] arrayFiguras=new ObjetoGeometrico[3];
    arrayFiguras[0]=new Circulo(10);
    arrayFiguras[1]=new Rectangulo(10,20);
    arrayFiguras[2]=new Circulo(40);
    for(ObjetoGeometrico figura:arrayFiguras)
       System.out.println(figura.getClass().getName()+" "+
                          figura.toString());
  }
}
```

Otro ejemplo

```
public class TestCirculoRectangulo {
  public static void main(String[] args) {
    ObjetoGeometrico[] arrayFiguras=new ObjetoGeometrico[3];
    arrayFiguras[0]=new Circulo(10);
    arrayFiguras[1]=new Rectangulo(10,20);
    arrayFiguras[2]=new Circulo(40);
    for(ObjetoGeometrico figura:arrayFiguras)
       System.out.println(figura.getClass().getName()+" "+
                          figura.toString());
  }
}
```

```
Circulo creado el Thu Mar 31 08:50:48 CEST 2016
color: blanco y relleno: false
    radio es 10.0
Rectangulo creado el Thu Mar 31 08:50:48 CEST 2016
color: blanco y relleno: false
    anchura es 10.0
    altura es 20.0
Circulo creado el Thu Mar 31 08:50:48 CEST 2016
color: blanco y relleno: false
    radio es 40.0
```

Otro ejemplo usando el parámetro de un método

```
public class LigaduraDinamicaDemo {
  public static void main(String[] args) {
    imprimir(new EstudiantedeGrado());
    imprimir(new Estudiante());
    imprimir(new Persona());
    imprimir(new Object());
  }
  public static void imprimir(Object x) {
    System.out.println(x.toString());
  }
}
class EstudiantedeGrado extends Estudiante {
}
class Estudiante extends Persona {
  public String toString() {
    return "Estudiante";
  }
}
class Persona extends Object {
  public String toString() {
    return "Persona";
  }
}
```

Otro ejemplo usando el parámetro de un método

```
public class LigaduraDinamicaDemo {
  public static void main(String[] args) {
    imprimir(new EstudiantedeGrado());
    imprimir(new Estudiante());
    imprimir(new Persona());
    imprimir(new Object());
  }
  public static void imprimir(Object x) {
    System.out.println(x.toString());
  }
}
class EstudiantedeGrado extends Estudiante {
}
class Estudiante extends Persona {
  public String toString() {
    return "Estudiante";
  }
}
class Persona extends Object {
  public String toString() {
    return "Persona";
  }
}
```

```
Estudiante
Estudiante
Persona
java.lang.Object@536091de
```

Contenido del tema

Casting de objetos

Casting de objetos

Significa que la referencia a un objeto puede ser convertida al tipo de otra referencia.

Casting de objetos

Casting de objetos

Significa que la referencia a un objeto puede ser convertida al tipo de otra referencia.

Upcasting

Es un *casting implícito* que se hace al asignar una referencia a objeto de una subclase, a una referencia de la superclase.

Casting de objetos

Casting de objetos

Significa que la referencia a un objeto puede ser convertida al tipo de otra referencia.

Upcasting

Es un *casting implícito* que se hace al asignar una referencia a objeto de una subclase, a una referencia de la superclase.

Ejemplo de upcasting

```
Object o = new Estudiante(); // casting implicito
imprimir(o);
```

o directamente:

```
imprimir(new Estudiante());
```

Downcasting

- El siguiente código da **error de compilación**: no se puede asignar directamente una referencia de la superclase a una de la subclase.

```
Object o = new Estudiante();
Estudiante b = o; // Error de compilacion
```

Downcasting

- El siguiente código da **error de compilación**: no se puede asignar directamente una referencia de la superclase a una de la subclase.

```
Object o = new Estudiante();
Estudiante b = o; // Error de compilacion
```

- La razón es que un objeto `Estudiante` es siempre una instancia de un `Object` pero un `Object` no es necesariamente una instancia de `Estudiante`.

Downcasting

- El siguiente código da **error de compilación**: no se puede asignar directamente una referencia de la superclase a una de la subclase.

```
Object o = new Estudiante();
Estudiante b = o; // Error de compilacion
```

- La razón es que un objeto `Estudiante` es siempre una instancia de un `Object` pero un `Object` no es necesariamente una instancia de `Estudiante`.
- El compilador no es suficientemente inteligente en el caso anterior, para saber que `o` es efectivamente un `Estudiante`.

Downcasting

- El siguiente código da **error de compilación**: no se puede asignar directamente una referencia de la superclase a una de la subclase.

```
Object o = new Estudiante();
Estudiante b = o; // Error de compilacion
```

- La razón es que un objeto `Estudiante` es siempre una instancia de un `Object` pero un `Object` no es necesariamente una instancia de `Estudiante`.
- El compilador no es suficientemente inteligente en el caso anterior, para saber que `o` es efectivamente un `Estudiante`.
- Necesitamos un *casting explícito* (**downcasting**) para indicárselo al compilador.

```
Object o = new Estudiante();
Estudiante b = (Estudiante)o; // casting explicito
```

Operador instanceof

- Siempre es posible convertir el tipo de una instancia de la subclase al tipo de una variable de la superclase (*upcasting*).

  *Una instancia de la subclase es **siempre** una instancia de su superclase.*

Operador instanceof

- Siempre es posible convertir el tipo de una instancia de la subclase al tipo de una variable de la superclase (*upcasting*).

  *Una instancia de la subclase es **siempre** una instancia de su superclase.*

- Para convertir una instancia de una superclase a una variable de su subclase (*downcasting*) se debe usar un casting explícito para confirmar tu intención de hacerlo al compilador.

Operador instanceof

- Siempre es posible convertir el tipo de una instancia de la subclase al tipo de una variable de la superclase (*upcasting*).

  *Una instancia de la subclase es **siempre** una instancia de su superclase.*

- Para convertir una instancia de una superclase a una variable de su subclase (*downcasting*) se debe usar un casting explícito para confirmar tu intención de hacerlo al compilador.
- Para que este casting pueda hacerse, deberíamos comprobar que el objeto es efectivamente una instancia de la subclase.

  Si no lo es, se producirá una excepción `ClassCastException` *en tiempo de ejecución.*

Operador instanceof

- Siempre es posible convertir el tipo de una instancia de la subclase al tipo de una variable de la superclase (*upcasting*).

  *Una instancia de la subclase es **siempre** una instancia de su superclase.*

- Para convertir una instancia de una superclase a una variable de su subclase (*downcasting*) se debe usar un casting explícito para confirmar tu intención de hacerlo al compilador.
- Para que este casting pueda hacerse, deberíamos comprobar que el objeto es efectivamente una instancia de la subclase.

  Si no lo es, se producirá una excepción `ClassCastException` *en tiempo de ejecución.*

- Para hacer esta comprobación, podemos utilizar el operador `instanceof`.

Operador instanceof

Ejemplo de uso de instanceof

```
ObjetoGeometrico figura = new Circulo();
...
if (figura instanceof Circulo) {
    System.out.println("El diametro del circulo es: " +
        ((Circulo)figura).getDiametro());
}
```

Operador instanceof

Ejemplo de uso de instanceof

```
ObjetoGeometrico figura = new Circulo();
...
if (figura instanceof Circulo) {
    System.out.println("El diametro del circulo es: " +
        ((Circulo)figura).getDiametro());
}
```

- Si usásemos `figura.getDiametro()` (sin el casting) se produciría un error de compilación porque la clase `ObjetoGeometrico` no tiene un método `getDiametro()`.
- Por tanto es obligatorio el casting.

Operador instanceof

Ejemplo de uso de instanceof

```
ObjetoGeometrico figura = new Circulo();
...
if (figura instanceof Circulo) {
    System.out.println("El diametro del circulo es: " +
        ((Circulo)figura).getDiametro());
}
```

- Si usásemos `figura.getDiametro()` (sin el casting) se produciría un error de compilación porque la clase `ObjetoGeometrico` no tiene un método `getDiametro()`.
- Por tanto es obligatorio el casting.
- Pero, ¿por qué no declaramos `figura` de tipo `Circulo`?

Operador instanceof

Ejemplo de uso de instanceof

```
ObjetoGeometrico figura = new Circulo();
...
if (figura instanceof Circulo) {
    System.out.println("El diametro del circulo es: " +
        ((Circulo)figura).getDiametro());
}
```

- Si usásemos `figura.getDiametro()` (sin el casting) se produciría un error de compilación porque la clase `ObjetoGeometrico` no tiene un método `getDiametro()`.
- Por tanto es obligatorio el casting.
- Pero, ¿por qué no declaramos `figura` de tipo `Circulo`?

*Esto permite **programación genérica**: código que sirve para varios tipos de objetos (no se necesita hacer una versión del código para cada tipo).*

Otro ejemplo de uso de instanceof

```
public class CastingDemo {
  public static void main(String[] args) {
    Object objeto1 = new Circulo(1);
    Object objeto2 = new Rectangulo(1, 1);
    displayObjecto(objeto1);
    displayObjecto(objeto2);
  }
  public static void displayObjecto(Object objeto) {
    if (objeto instanceof Circulo) {
      System.out.println("El area del circulo es " +
        ((Circulo)objeto).getArea());
      System.out.println("El diametro del circulo es " +
        ((Circulo)objeto).getDiametro());
    }
    else if (objeto instanceof  Rectangulo) {
      System.out.println("El area del rectangulo es " +
        ((Rectangulo)objeto).getArea());
    }
  }
}
```

Otro ejemplo de uso de instanceof

```
public class CastingDemo {
  public static void main(String[] args) {
    Object objeto1 = new Circulo(1);
    Object objeto2 = new Rectangulo(1, 1);
    displayObjecto(objeto1);
    displayObjecto(objeto2);
  }
  public static void displayObjecto(Object objeto) {
    if (objeto instanceof Circulo) {
      System.out.println("El area del circulo es " +
        ((Circulo)objeto).getArea());
      System.out.println("El diametro del circulo es " +
        ((Circulo)objeto).getDiametro());
    }
    else if (objeto instanceof  Rectangulo) {
      System.out.println("El area del rectangulo es " +
        ((Rectangulo)objeto).getArea());
    }
  }
}
```

```
El area del circulo es 3.141592653589793
El diametro del circulo es 2.0
El area del rectangulo es 1.0
```

Nota sobre los casting explícitos

El casting de un valor de tipo primitivo es diferente al de una referencia a objeto.

- El casting de un valor de tipo primitivo devuelve un nuevo valor

```
int edad = 45;
byte nuevaEdad = (byte)edad; // Nuevo valor para nuevaEdad
```

Nota sobre los casting explícitos

El casting de un valor de tipo primitivo es diferente al de una referencia a objeto.

- El casting de un valor de tipo primitivo devuelve un nuevo valor

```
int edad = 45;
byte nuevaEdad = (byte)edad; // Nuevo valor para nuevaEdad
```

- Sin embargo, un casting de una referencia a objeto no crea un nuevo objeto.

```
Object o = new Circulo();
Circulo c = (Circulo)o; // No crea ningun objeto nuevo
```

Nota sobre los casting explícitos

El casting de un valor de tipo primitivo es diferente al de una referencia a objeto.

- El casting de un valor de tipo primitivo devuelve un nuevo valor

```
int edad = 45;
byte nuevaEdad = (byte)edad; // Nuevo valor para nuevaEdad
```

- Sin embargo, un casting de una referencia a objeto no crea un nuevo objeto.

```
Object o = new Circulo();
Circulo c = (Circulo)o; // No crea ningun objeto nuevo
```

Las referencias `o` y `c` apuntan ahora al mismo objeto.

Contenido del tema

El método equals

El método equals

Es un método definido en la clase `Object` que se usa para ver si dos objetos son iguales.

```
objeto1.equals(objeto2);
```

El método equals

El método equals

Es un método definido en la clase `Object` que se usa para ver si dos objetos son iguales.

```
objeto1.equals(objeto2);
```

- El prototipo de `equals` es:

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

El método equals

El método equals

Es un método definido en la clase `Object` que se usa para ver si dos objetos son iguales.

```
objeto1.equals(objeto2);
```

- El prototipo de `equals` es:

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

- La implementación por defecto del método `equals` en la clase `Object` es:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

El método equals

El método equals

Es un método definido en la clase `Object` que se usa para ver si dos objetos son iguales.

```
objeto1.equals(objeto2);
```

- El prototipo de `equals` es:
```
public boolean equals(Object o)
```
- La implementación por defecto del método `equals` en la clase `Object` es:
```
public boolean equals(Object obj) {
    return (this == obj);
}
```
- Esta implementación comprueba si dos referencias apuntan al mismo objeto mediante el operador ==.

El método equals

El método equals

Es un método definido en la clase `Object` que se usa para ver si dos objetos son iguales.

```
objeto1.equals(objeto2);
```

- El prototipo de `equals` es:
```
public boolean equals(Object o)
```
- La implementación por defecto del método `equals` en la clase `Object` es:
```
public boolean equals(Object obj) {
    return (this == obj);
}
```
- Esta implementación comprueba si dos referencias apuntan al mismo objeto mediante el operador ==.
- Se debería sobreescribir este método en nuestras subclases para comprobar que dos objetos tienen el mismo contenido.

Sobreescribiendo el método equals

El método `equals` está sobreescrito en muchas clases en el API de Java, tal como `java.lang.String` y `java.util.Date`.

```
public boolean equals(Object o) {
  if (o instanceof Circulo) {
    return radio == ((Circulo)o).radio;
  }
  else
    return false;
}
```

Sobreescribiendo el método equals

El método `equals` está sobreescrito en muchas clases en el API de Java, tal como `java.lang.String` y `java.util.Date`.

```
public boolean equals(Object o) {
  if (o instanceof Circulo) {
    return radio == ((Circulo)o).radio;
  }
  else
    return false;
}
```

- El operador == se usa para comparar dos datos de tipo primitivo o para determinar si dos referencias apuntan al mismo objeto.

Sobreescribiendo el método equals

El método `equals` está sobreescrito en muchas clases en el API de Java, tal como `java.lang.String` y `java.util.Date`.

```
public boolean equals(Object o) {
  if (o instanceof Circulo) {
    return radio == ((Circulo)o).radio;
  }
  else
    return false;
}
```

- El operador == se usa para comparar dos datos de tipo primitivo o para determinar si dos referencias apuntan al mismo objeto.
- El objetivo del método `equals` es comprobar si dos objetos tienen el mismo contenido.

Sobreescribiendo el método equals

El método `equals` está sobreescrito en muchas clases en el API de Java, tal como `java.lang.String` y `java.util.Date`.

```
public boolean equals(Object o) {
  if (o instanceof Circulo) {
    return radio == ((Circulo)o).radio;
  }
  else
    return false;
}
```

- El operador == se usa para comparar dos datos de tipo primitivo o para determinar si dos referencias apuntan al mismo objeto.
- El objetivo del método `equals` es comprobar si dos objetos tienen el mismo contenido.
- El operador == es más fuerte que el método `equals`, ya que == comprueba si dos variables referencian al mismo objeto.

Contenido del tema

Datos y métodos protected

Miembros protected

Un dato o método `protected` de una clase, es accesible desde los métodos de las subclases.

```
public class ObjetoGeometrico {
   ...
   protected ObjetoGeometrico(String color, boolean relleno) {
     fechaCreacion = new java.util.Date();
     this.color = color;
     this.relleno = relleno;
   }
}
public class Circulo extends ObjetoGeometrico {
   ...
   public Circulo(double radio, String color, boolean relleno) {
     super(color,relleno); // Llama a constructor superclase
     this.radio = radio;
   }
}
```

Contenido del tema

Uso de **final**

1. Para creación de constantes con nombre.

Uso de **final**

1. Para creación de constantes con nombre.
2. Para evitar sobreescritura de métodos

   *Los métodos declarados como **final** no pueden ser sobreescritos*

Uso de **final**

1. Para creación de constantes con nombre.
2. Para evitar sobreescritura de métodos

   *Los métodos declarados como **final** no pueden ser sobreescritos*

3. Para evitar la extensión de una clase

   *Se usa **final** en la declaración de la clase para evitar que la clase sea extendida: O sea, todos sus métodos serán **final** implícitamente.*

Ejemplo de **final** para evitar sobreescritura de un método

```
class A {
  final void metodo() {
    System.out.println("Este es un metodo final.");
  }
}

class B extends A {
  void metodo() { // ERROR: No se puede sobreescribir.
    System.out.println("No es correcto!");
  }
}
```

Ejemplo de **final** para evitar extensión de una clase

```
final class A {
  // ...
}

// La clase siguiente no es valida.
class B extends A { // ERROR: No se puede crear una subclase de A
  // ...
}
```

Contenido del tema

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).
- Sin embargo, estos métodos no pueden implementarse en la clase `ObjetoGeometrico`, porque su implementación depende del tipo específico de la figura.

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).
- Sin embargo, estos métodos no pueden implementarse en la clase `ObjetoGeometrico`, porque su implementación depende del tipo específico de la figura.
- Tales métodos se conocen como métodos abstractos.

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).
- Sin embargo, estos métodos no pueden implementarse en la clase `ObjetoGeometrico`, porque su implementación depende del tipo específico de la figura.
- Tales métodos se conocen como métodos abstractos.
- Se usa el modificador `abstract` en la cabecera del método.

Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).
- Sin embargo, estos métodos no pueden implementarse en la clase `ObjetoGeometrico`, porque su implementación depende del tipo específico de la figura.
- Tales métodos se conocen como métodos abstractos.
- Se usa el modificador `abstract` en la cabecera del método.
- Al hacerlo, la clase debe declararse también como `abstract`.

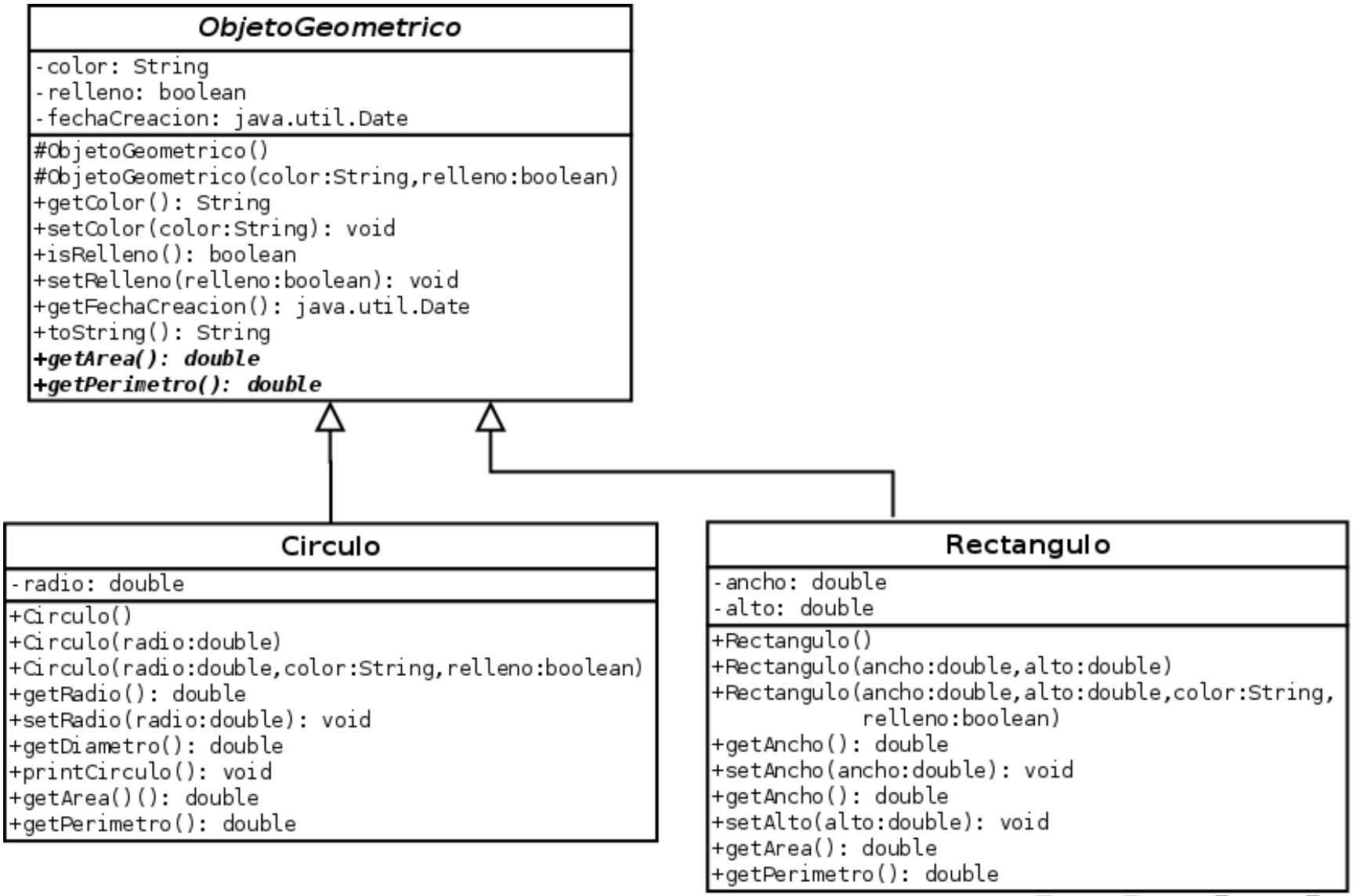
Clases abstractas

Clase abstracta

Permite definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de todos sus métodos. Se usa cuando la clase es tan abstracta que no se pueden construir instancias concretas de esta clase.

- Por ejemplo, en la clase `ObjetoGeometrico`, se podrían incluir los métodos `getArea()` y `getPerimetro()`, ya que son aplicables a todas las subclases (`Circulo`, `Rectangulo`).
- Sin embargo, estos métodos no pueden implementarse en la clase `ObjetoGeometrico`, porque su implementación depende del tipo específico de la figura.
- Tales métodos se conocen como métodos abstractos.
- Se usa el modificador `abstract` en la cabecera del método.
- Al hacerlo, la clase debe declararse también como `abstract`.
- En la notación UML, los nombres de clases y métodos `abstract` se ponen en itálica.

Clases abstractas



Clases abstractas: Ejemplo

```
public abstract class ObjetoGeometrico {
  private String color = "blanco";
  private boolean relleno;
  private java.util.Date fechaCreacion;

  protected ObjetoGeometrico() {
    fechaCreacion = new java.util.Date();
  }
  protected ObjetoGeometrico(String color, boolean relleno) {
    fechaCreacion = new java.util.Date();
    this.color = color;
    this.relleno = relleno;
  }
  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }
```

Clases abstractas: Ejemplo

```
  public boolean isRelleno() {
    return relleno;
  }
  public void setRelleno(boolean relleno) {
    this.relleno = relleno;
  }
  public java.util.Date getFechaCreacion() {
    return fechaCreacion;
  }
  public String toString() {
    return "creado el " + fechaCreacion + "\ncolor: " + color +
      " y relleno: " + relleno;
  }
  public abstract double getArea();
  public abstract double getPerimetro();
}
```

Clases abstractas: Ejemplo

```
public class Circulo extends ObjetoGeometrico {
  private double radio;
  ...
  public double getArea() {
    return radio * radio * Math.PI;
  }
  public double getPerimetro() {
    return 2 * radio * Math.PI;
  }
  ...
}
public class Rectangulo extends ObjetoGeometrico {
  private double ancho;
  private double alto;
  ...
  public double getArea() {
    return ancho * alto;
  }
  public double getPerimetro() {
    return 2 * (ancho + alto);
  }
  ...
}
```

Clases abstractas: Ejemplo

```
public class TestObjetoGeometrico {
  public static void main(String[] args) {
    //ObjetoGeometrico objeto = new ObjetoGeometrico(); ERROR: ObjetoGeometrico es abstract
    ObjetoGeometrico objeto1 = new Circulo(5);
    ObjetoGeometrico objeto2 = new Rectangulo(5, 3);
    System.out.println("Tienen los dos objetos igual area? " +
      equalArea(objeto1, objeto2));
    displayObjetoGeometrico(objeto1);
    displayObjetoGeometrico(objeto2);
  }
  public static boolean equalArea(ObjetoGeometrico objeto1,
      ObjetoGeometrico objeto2) {
    return objeto1.getArea() == objeto2.getArea();
  }
  public static void displayObjetoGeometrico(ObjetoGeometrico objeto) {
    System.out.println("\nEl area es " + objeto.getArea());
    System.out.println("El perimetro es " + objeto.getPerimetro());
  }
}
```

Clases abstractas: Ejemplo

```
public class TestObjetoGeometrico {
  public static void main(String[] args) {
    //ObjetoGeometrico objeto = new ObjetoGeometrico(); ERROR: ObjetoGeometrico es abstract
    ObjetoGeometrico objeto1 = new Circulo(5);
    ObjetoGeometrico objeto2 = new Rectangulo(5, 3);
    System.out.println("Tienen los dos objetos igual area? " +
      equalArea(objeto1, objeto2));
    displayObjetoGeometrico(objeto1);
    displayObjetoGeometrico(objeto2);
  }
  public static boolean equalArea(ObjetoGeometrico objeto1,
      ObjetoGeometrico objeto2) {
    return objeto1.getArea() == objeto2.getArea();
  }
  public static void displayObjetoGeometrico(ObjetoGeometrico objeto) {
    System.out.println("\nEl area es " + objeto.getArea());
    System.out.println("El perimetro es " + objeto.getPerimetro());
  }
}
```

```
¿Tienen los dos objetos igual area? false

El area es 78.53981633974483
El perimetro es 31.41592653589793

El area es 15.0
El perimetro es 16.0
```

Clases abstractas: Ejemplo

- En el ejemplo anterior, los métodos abstractos `getArea()` y `getPerimetro()` de la clase `ObjetoGeometrico` están sobreescritos en las subclases `Circulo` y `Rectangulo`.

Clases abstractas: Ejemplo

- En el ejemplo anterior, los métodos abstractos `getArea()` y `getPerimetro()` de la clase `ObjetoGeometrico` están sobreescritos en las subclases `Circulo` y `Rectangulo`.
- La JVM determina dinámicamente a qué versión de `getArea()` y `getPerimetro` debe llamarse cuando se usan en `TestObjetoGeometrico`.

Clases abstractas: Ejemplo

- En el ejemplo anterior, los métodos abstractos `getArea()` y `getPerimetro()` de la clase `ObjetoGeometrico` están sobreescritos en las subclases `Circulo` y `Rectangulo`.
- La JVM determina dinámicamente a qué versión de `getArea()` y `getPerimetro` debe llamarse cuando se usan en `TestObjetoGeometrico`.
- Si no hubiésemos definido el método `getArea()` en la clase `ObjetoGeometrico`, no se podría haber definido el método `equalArea` para comparar si dos objetos tienen igual área.

Clases abstractas: otro ejemplo de uso

Sí es posible declarar variables cuyo tipo sea una clase `abstract`

```
public class TestObjetoGeometrico2 {
  public static void main(String[] args) {
    //ObjetoGeometrico objeto = new ObjetoGeometrico(); ERROR: ObjetoGeometrico es abstract
    Circulo circulo = new Circulo(5);
    Rectangulo rectangulo = new Rectangulo(5, 3);
    ObjetoGeometrico figura1,figura2; // Esto si es correcto, no se crea ningun objeto
    figura1 = circulo;
    figura2 = rectangulo;

    System.out.println("?'Tienen los dos objetos igual area? " +
      equalArea(figura1, figura1));
    displayObjetoGeometrico(figura1);
    displayObjetoGeometrico(figura2);
  }
  public static boolean equalArea(ObjetoGeometrico objeto1,
      ObjetoGeometrico objeto2) {
    return objeto1.getArea() == objeto2.getArea();
  }
  public static void displayObjetoGeometrico(ObjetoGeometrico objeto) {
    //System.out.println();
    System.out.println("\nEl area es " + objeto.getArea());
    System.out.println("El perimetro es " + objeto.getPerimetro());
  }
}
```

Clases abstractas: otro ejemplo de uso

```
¿Tienen los dos objetos igual area? true

El area es 78.53981633974483
El perimetro es 31.41592653589793

El area es 15.0
El perimetro es 16.0
```

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

```
abstract tipo nombre(parámetros);
```

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

```
abstract tipo nombre(parámetros);
```

- No se pueden crear objetos de clases abstractas (usando **new**).

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

  ```
  abstract tipo nombre(parámetros);
  ```

- No se pueden crear objetos de clases abstractas (usando **new**).
- Aunque las clases abstractas sí pueden tener constructores, que pueden ser llamados desde los constructores de las subclases.

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

  ```
  abstract tipo nombre(parámetros);
  ```

- No se pueden crear objetos de clases abstractas (usando **new**).
- Aunque las clases abstractas sí pueden tener constructores, que pueden ser llamados desde los constructores de las subclases.
- Es posible definir una clase como abstracta aunque no contenga ningún método abstracto. Esta clase se usa como clase base para otras subclases.

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

  ```
  abstract tipo nombre(parámetros);
  ```

- No se pueden crear objetos de clases abstractas (usando **new**).
- Aunque las clases abstractas sí pueden tener constructores, que pueden ser llamados desde los constructores de las subclases.
- Es posible definir una clase como abstracta aunque no contenga ningún método abstracto. Esta clase se usa como clase base para otras subclases.
- Una clase puede ser abstracta, aunque la superclase no lo sea.

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

  ```
  abstract tipo nombre(parámetros);
  ```

- No se pueden crear objetos de clases abstractas (usando **new**).
- Aunque las clases abstractas sí pueden tener constructores, que pueden ser llamados desde los constructores de las subclases.
- Es posible definir una clase como abstracta aunque no contenga ningún método abstracto. Esta clase se usa como clase base para otras subclases.
- Una clase puede ser abstracta, aunque la superclase no lo sea.
- No se pueden crear constructores **abstract** o métodos **static abstract**.

Algunas notas sobre las clases abstractas

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por la subclase o bien ésta se declarará también como abstracta.
- Los métodos abstractos tienen la forma:

  ```
  abstract tipo nombre(parámetros);
  ```

- No se pueden crear objetos de clases abstractas (usando **new**).
- Aunque las clases abstractas sí pueden tener constructores, que pueden ser llamados desde los constructores de las subclases.
- Es posible definir una clase como abstracta aunque no contenga ningún método abstracto. Esta clase se usa como clase base para otras subclases.
- Una clase puede ser abstracta, aunque la superclase no lo sea.
- No se pueden crear constructores **abstract** o métodos **static abstract**.
- Sí que podemos declarar variables referencia de una clase abstracta.

Contenido del tema

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo**: Las clases del paquete **paquete** (ficheros* `.class` *y* `.java`*) se almacenarán en directorio **paquete

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo**: Las clases del paquete **paquete** (ficheros* `.class` *y* `.java`*) se almacenarán en directorio **paquete

- Para usar las clases de un paquete, debe importarse explícitamente con la sentencia

```
import nombre-paquete;
```

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo**: Las clases del paquete **paquete** (ficheros* `.class` *y* `.java`*) se almacenarán en directorio **paquete

- Para usar las clases de un paquete, debe importarse explícitamente con la sentencia

```
import nombre-paquete;
```

- Permiten restringir la visibilidad de las clases que contiene:

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo**: Las clases del paquete **paquete** (ficheros* `.class` *y* `.java`*) se almacenarán en directorio **paquete

- Para usar las clases de un paquete, debe importarse explícitamente con la sentencia

```
import nombre-paquete;
```

- Permiten restringir la visibilidad de las clases que contiene:

  Se pueden definir clases en un paquete sin que el mundo exterior sepa que están allí.

Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos (carpetas).

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo**: Las clases del paquete **paquete** (ficheros* `.class` *y* `.java`*) se almacenarán en directorio **paquete

- Para usar las clases de un paquete, debe importarse explícitamente con la sentencia

```
import nombre-paquete;
```

- Permiten restringir la visibilidad de las clases que contiene:

  Se pueden definir clases en un paquete sin que el mundo exterior sepa que están allí.

- Se pueden definir miembros de una clase, que sean sólo accesibles por miembros del mismo paquete

Definición de paquetes

Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

```
package nombre_paquete;
```

- Todas las clases de ese archivo serán de ese paquete.

Definición de paquetes

Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

```
package nombre_paquete;
```

- Todas las clases de ese archivo serán de ese paquete.
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*.

Definición de paquetes

Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

```
package nombre-paquete;
```

- Todas las clases de ese archivo serán de ese paquete.
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*.
- Una misma sentencia `package` puede incluirse en varios archivos fuente (en varias clases).

Definición de paquetes

Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

```
package nombre-paquete;
```

- Todas las clases de ese archivo serán de ese paquete.
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*.
- Una misma sentencia `package` puede incluirse en varios archivos fuente (en varias clases).
- Se puede crear una jerarquía de paquetes:
  `package paq1[.paq2[.paq3]];`
  Ejemplo: `package java.awt.image`

Definición de paquetes

Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

```
package nombre-paquete;
```

- Todas las clases de ese archivo serán de ese paquete.
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*.
- Una misma sentencia `package` puede incluirse en varios archivos fuente (en varias clases).
- Se puede crear una jerarquía de paquetes:
  `package paq1[.paq2[.paq3]];`
  Ejemplo: `package java.awt.image`
- El anterior paquete supone que existe un directorio `java/awt/image`, dónde se colocan las clases de este paquete.

Importar un paquete

Importar un paquete

Al importar las clases de un paquete, podremos referirnos a las clases que contiene, directamente con su nombre. Usamos la sintaxis

```
import paquete[.paquete2].(nombre_clase|*);
```

- Si no importamos el paquete, tendremos que especificar el nombre completo del paquete antes del nombre de la clase cada vez que nos refiramos a esa clase.
  Por ejemplo para la clase **Date** usaríamos **java.útil.Date**.

Importar un paquete

Importar un paquete

Al importar las clases de un paquete, podremos referirnos a las clases que contiene, directamente con su nombre. Usamos la sintaxis

```
import paquete[.paquete2].(nombre_clase|*);
```

- Si no importamos el paquete, tendremos que especificar el nombre completo del paquete antes del nombre de la clase cada vez que nos refiramos a esa clase.
  Por ejemplo para la clase **Date** usaríamos **java.útil.Date**.
- La sentencia `import` debe ir tras la sentencia **package**.

Importar un paquete

Importar un paquete

Al importar las clases de un paquete, podremos referirnos a las clases que contiene, directamente con su nombre. Usamos la sintaxis

```
import paquete[.paquete2].(nombre_clase|*);
```

- Si no importamos el paquete, tendremos que especificar el nombre completo del paquete antes del nombre de la clase cada vez que nos refiramos a esa clase.
  Por ejemplo para la clase **Date** usaríamos **java.útil.Date**.
- La sentencia `import` debe ir tras la sentencia **package**.
- Al usar `*` especificamos que se importa el paquete completo. Esto incrementa el tiempo de compilación, pero no el de ejecución.

Importar un paquete

Importar un paquete

Al importar las clases de un paquete, podremos referirnos a las clases que contiene, directamente con su nombre. Usamos la sintaxis

```
import paquete[.paquete2].(nombre_clase|*);
```

- Si no importamos el paquete, tendremos que especificar el nombre completo del paquete antes del nombre de la clase cada vez que nos refiramos a esa clase.
  Por ejemplo para la clase **Date** usaríamos **java.útil.Date**.
- La sentencia `import` debe ir tras la sentencia **package**.
- Al usar `*` especificamos que se importa el paquete completo. Esto incrementa el tiempo de compilación, pero no el de ejecución.
- Las clases estándar de Java están dentro del paquete **java**.

Importar un paquete

Importar un paquete

Al importar las clases de un paquete, podremos referirnos a las clases que contiene, directamente con su nombre. Usamos la sintaxis

```
import paquete[.paquete2].(nombre_clase|*);
```

- Si no importamos el paquete, tendremos que especificar el nombre completo del paquete antes del nombre de la clase cada vez que nos refiramos a esa clase.
  Por ejemplo para la clase **Date** usaríamos **java.útil.Date**.
- La sentencia `import` debe ir tras la sentencia **package**.
- Al usar `*` especificamos que se importa el paquete completo. Esto incrementa el tiempo de compilación, pero no el de ejecución.
- Las clases estándar de Java están dentro del paquete **java**.
- Las funciones básicas del lenguaje se almacenan en el paquete. **java.lang**, el cual es importado por defecto.

Ejemplo

Ejemplo de uso de package e import

```
package mipaquete;
/* La clase Balance, su constructor, y su metodo mostrar()
   deben ser publicos para poder ser utilizados por
   codigo que no sea una subclase y este fuera de su paquete.
*/
public class Balance {
  String nombre;
  double bal;
  public Balance(String n, double b) {
    nombre = n;
    bal = b;
  }
  public void mostrar() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(nombre + ": $" + bal);
  }
}
```

Ejemplo

```
import mipaquete.*;
class TestBalance {
  public static void main(String args[]) {
    /* Como Balance es publica, se puede utilizar la
       clase Balance y llamar a su constructor. */
    Balance test = new Balance("Antonio Campos", 99.88);
    test.mostrar(); // tambien se puede llamar al metodo mostrar()
  }
}
```

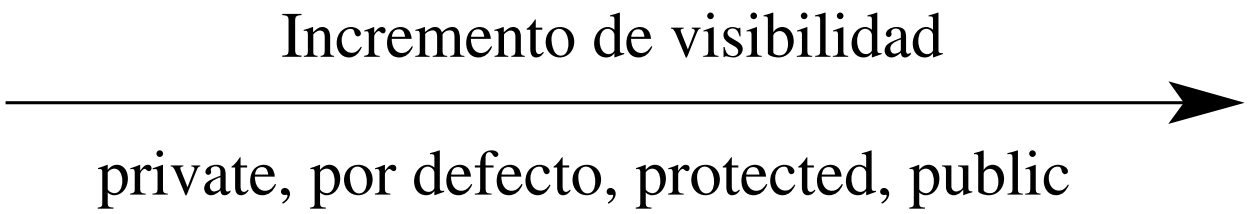
Contenido del tema

Modificadores de visibilidad

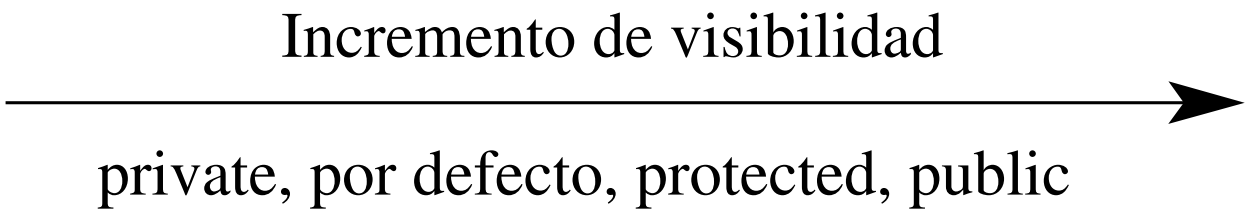
Las clases y los paquetes son dos medios de encapsular y contener el espacio de nombres y el ámbito de las variables y métodos.

- **Paquetes**: Actúan como recipientes de clases y otros paquetes subordinados.
- **Clases**: Actúan como recipientes de datos y código.

Modificadores de visibilidad para miembros de una clase



Modificadores de visibilidad para miembros de una clase



Desde método en ...	**private**	**sin modif.**	**protected**	**public**
misma clase	sí	sí	sí	sí
clase (subclase o no) del mismo paquete	no	sí	sí	sí
subclase de diferente paquete	no	no	sí	sí
no subclase de diferente paquete	no	no	no	sí

Modificadores de visibilidad para miembros de una clase

- Usamos `private` para ocultar los miembros de la clase al resto de clases.

Modificadores de visibilidad para miembros de una clase

- Usamos `private` para ocultar los miembros de la clase al resto de clases.
- No usamos ningún modificador (acceso por defecto) para permitir que los miembros de una clase sean accesibles desde cualquier clase del mismo paquete, pero no desde otros paquetes.

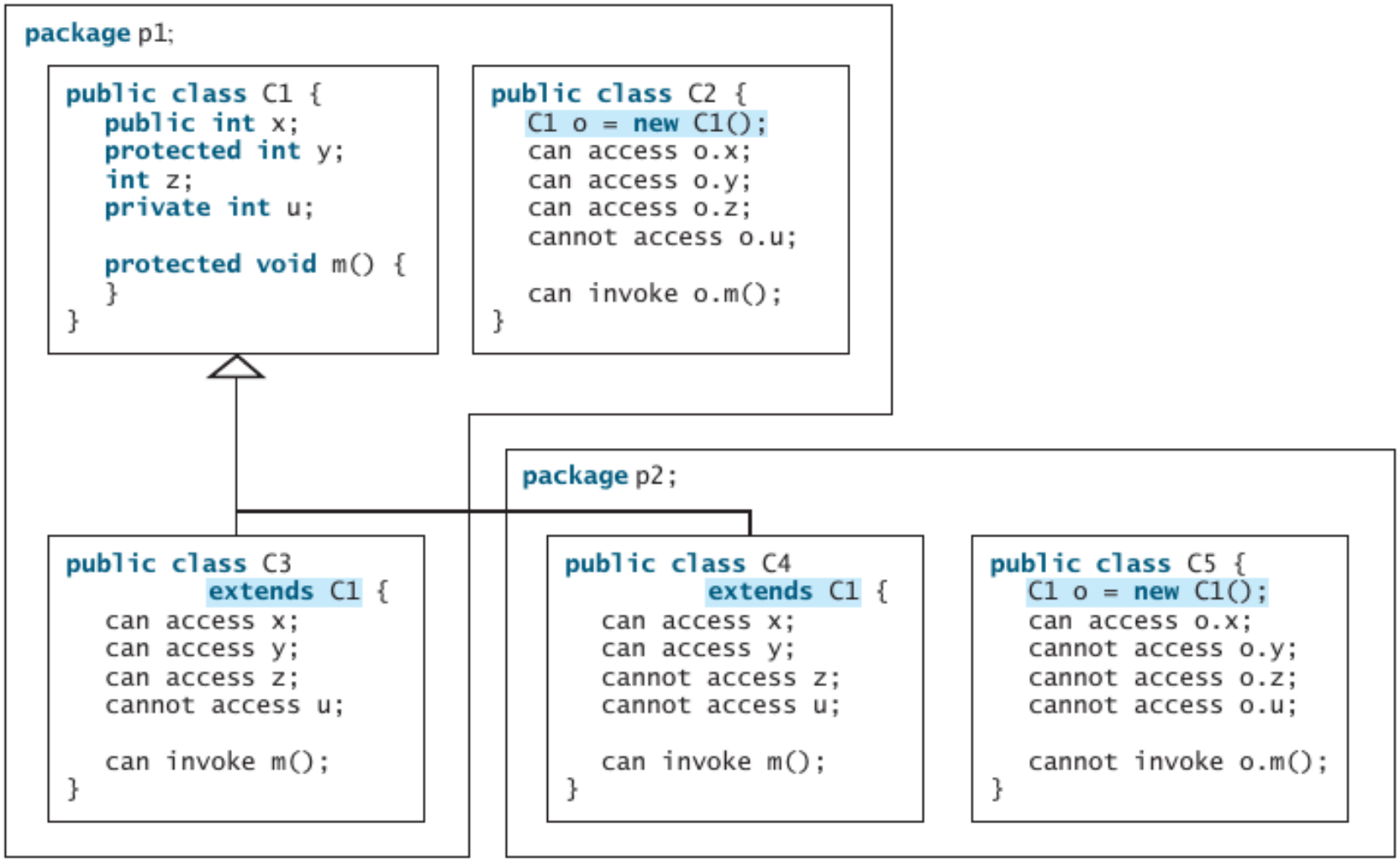
Modificadores de visibilidad para miembros de una clase

- Usamos `private` para ocultar los miembros de la clase al resto de clases.
- No usamos ningún modificador (acceso por defecto) para permitir que los miembros de una clase sean accesibles desde cualquier clase del mismo paquete, pero no desde otros paquetes.
- Usamos `protected` para permitir que los miembros de la clase sean accesibles desde subclases de cualquier paquete o clases del mismo paquete.

Modificadores de visibilidad para miembros de una clase

- Usamos `private` para ocultar los miembros de la clase al resto de clases.
- No usamos ningún modificador (acceso por defecto) para permitir que los miembros de una clase sean accesibles desde cualquier clase del mismo paquete, pero no desde otros paquetes.
- Usamos `protected` para permitir que los miembros de la clase sean accesibles desde subclases de cualquier paquete o clases del mismo paquete.
- Usamos `public` para permitir que los miembros de una clase sean accesibles desde cualquier clase.

Modificadores de visibilidad para miembros de una clase



Modificación de la visibilidad en una subclase

Modificación de la visibilidad en una subclase

- Una subclase puede sobreescribir un método `protected` de su superclase, y cambiar su visibilidad a `public`.
- Sin embargo una subclase no puede reducir la visibilidad de un método definido en la superclase.

Ejemplo de modificadores de visibilidad

```
package p1;
public class Proteccion {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;
  public Proteccion() {
    System.out.println("constructor base ");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class Derivada extends Proteccion {
  Derivada() {
    System.out.println("constructor de Derivada");
    System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class MismoPaquete {
  MismoPaquete() {
    Proteccion p = new Proteccion();
    System.out.println("constructor de MismoPaquete");
    System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

```java
package p2;

class Proteccion2 extends p1.Proteccion {
  Proteccion2() {
    System.out.println("constructor de Proteccion2");
//  System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class OtroPaquete {
  OtroPaquete() {
    p1.Proteccion p = new p1.Proteccion();
    System.out.println("constructor de otro paquete");
//  System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
//  System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

Modificadores de visibilidad para clases

- **Acceso por defecto**: Accesible sólo por código del mismo paquete
- **Acceso public**: Accesible por cualquier código

Contenido del tema

Interfaces

Interfaz

Es sintácticamente como una clase y contiene sólo constantes y métodos abstractos.

```
acceso interface NombreInterfaz {
    /** Declaraciones de constantes */
    /** Lista de metodos abstractos */
}
```

Interfaces

Interfaz

Es sintácticamente como una clase y contiene sólo constantes y métodos abstractos.

```
acceso interface NombreInterfaz {
    /** Declaraciones de constantes */
    /** Lista de metodos abstractos */
}
```

- Es similar en muchos aspectos a una clase abstracta.

Interfaces

Interfaz

Es sintácticamente como una clase y contiene sólo constantes y métodos abstractos.

```
acceso interface NombreInterfaz {
    /** Declaraciones de constantes */
    /** Lista de metodos abstractos */
}
```

- Es similar en muchos aspectos a una clase abstracta.
- Se utilizan para especificar un comportamiento (lo que debe hacer una clase pero no cómo lo hace) común para los objetos de las clases que lo implementen.

Interfaces

Interfaz

Es sintácticamente como una clase y contiene sólo constantes y métodos abstractos.

```
acceso interface NombreInterfaz {
    /** Declaraciones de constantes */
    /** Lista de metodos abstractos */
}
```

- Es similar en muchos aspectos a una clase abstracta.
- Se utilizan para especificar un comportamiento (lo que debe hacer una clase pero no cómo lo hace) común para los objetos de las clases que lo implementen.
- Por ejemplo, podría usarse para especificar que los objetos son *comparables*, *comestibles* o *clonables*.

Interfaces

Interfaz

Es sintácticamente como una clase y contiene sólo constantes y métodos abstractos.

```
acceso interface NombreInterfaz {
   /** Declaraciones de constantes */
   /** Lista de metodos abstractos */
}
```

- Es similar en muchos aspectos a una clase abstracta.
- Se utilizan para especificar un comportamiento (lo que debe hacer una clase pero no cómo lo hace) común para los objetos de las clases que lo implementen.
- Por ejemplo, podría usarse para especificar que los objetos son *comparables*, *comestibles* o *clonables*.
- *acceso* puede ser **public** o vacío (acceso por defecto) al igual que en clases.

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.
- Se pueden usar interfaces más o menos de la misma forma que clases abstractas. Por ejemplo:

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.
- Se pueden usar interfaces más o menos de la misma forma que clases abstractas. Por ejemplo:
  - Se puede usar una interfaz para el tipo de una variable referencia.

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.
- Se pueden usar interfaces más o menos de la misma forma que clases abstractas. Por ejemplo:
  - Se puede usar una interfaz para el tipo de una variable referencia.
  - No se pueden crear objetos de una interfaz con `new`.

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.
- Se pueden usar interfaces más o menos de la misma forma que clases abstractas. Por ejemplo:
  - Se puede usar una interfaz para el tipo de una variable referencia.
  - No se pueden crear objetos de una interfaz con `new`.
- Podemos usar la interfaz `Comestible` para especificar si un objeto es o no comestible.

Interfaces

Ejemplo de interfaz

```
public interface Comestible {
  /** Describe como se come */
  public abstract String comoComer();
}
```

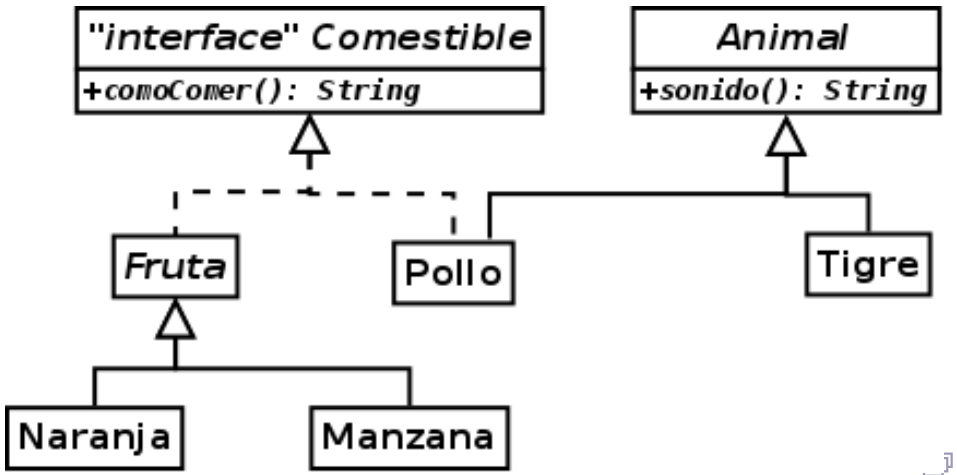
- Al compilar una interfaz se obtiene un fichero bytecode al igual que al compilar una clase.
- Se pueden usar interfaces más o menos de la misma forma que clases abstractas. Por ejemplo:
  - Se puede usar una interfaz para el tipo de una variable referencia.
  - No se pueden crear objetos de una interfaz con `new`.
- Podemos usar la interfaz `Comestible` para especificar si un objeto es o no comestible.
- Para ello, la clase del objeto debe implementar esta interfaz, usando la palabra reservada `implements`.

Interfaces

Ejemplo de uso de la interfaz Comestible

Podemos usar la interfaz `Comestible` para especificar si un objeto es comestible.

- Por ejemplo, en el siguiente código, las clases `Pollo` y `Fruta` implementan esta interfaz.
- La clase `Pollo` también extiende la clase abstracta `Animal`.
- La clase `Fruta` es una clase abstracta ya que no implementa el método `comoComer()`.



Interfaces

```
abstract class Animal {
  public abstract String sonido();
}
class Pollo extends Animal implements Comestible {
  public String comoComer() {
    return "Pollo: Freirlo";
  }
  public String sonido() {
    return "Pollo: pio, pio";
  }
}
class Tigre extends Animal {
  public String sonido() {
    return "Tigre: RROOAARR";
  }
}
abstract class Fruta implements Comestible {
  // Se omiten atributos, constructores y metodos
}
class Manzana extends Fruta {
  public String comoComer() {
    return "Manzana: Hacer sidra de manzana";
  }
}
class Naranja extends Fruta {
  public String comoComer() {
    return "Orange: Hacer zumo de naranja";
  }
}
```

Interfaces

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

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

Interfaces

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

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

```
Tigre: RROOAARR
Pollo: Freirlo
Pollo: pio, pio
Manzana: Hacer sidra de manzana
```

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Si una interfaz tiene variables, éstas serán implícitamente **final** y **static**.

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Si una interfaz tiene variables, éstas serán implícitamente **final** y **static**.
- Puesto que en una interfaz todos los atributos son `public static final` y todos los métodos son `public abstract`, Java permite omitir estos especificadores.

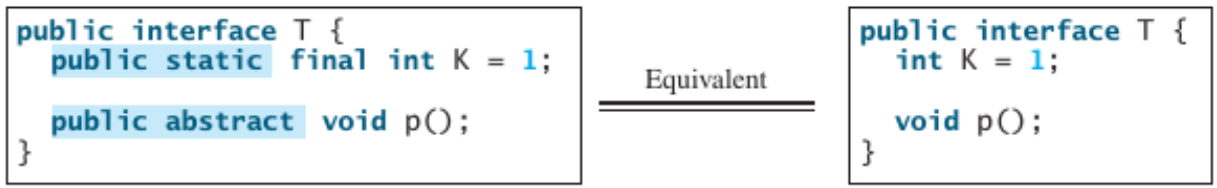
```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Si una interfaz tiene variables, éstas serán implícitamente **final** y **static**.
- Puesto que en una interfaz todos los atributos son `public static final` y todos los métodos son `public abstract`, Java permite omitir estos especificadores.



- Al implementar un método de una interfaz en una clase, éste tiene que declararse como **public**.

Algunas notas sobre interfaces

- Una clase puede implementar cualquier número de interfaces.
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Si una interfaz tiene variables, éstas serán implícitamente **final** y **static**.
- Puesto que en una interfaz todos los atributos son `public static final` y todos los métodos son `public abstract`, Java permite omitir estos especificadores.

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

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

- Al implementar un método de una interfaz en una clase, éste tiene que declararse como **public**.
- Si una clase implementa una interfaz, pero no implementa todos sus métodos, entonces debe ser declarada como **abstract**.

Ligadura dinámica usando variables de una interfaz

Se pueden declarar variables cuyo tipo es una interfaz para referenciar objetos de clases que implementan esa interfaz.

- Esto permite usarlas para la *ligadura dinámica*: determinar en tiempo de ejecución el método al que se llamará según la clase del objeto al que apunte.

Ligadura dinámica usando variables de una interfaz

Se pueden declarar variables cuyo tipo es una interfaz para referenciar objetos de clases que implementan esa interfaz.

- Esto permite usarlas para la *ligadura dinámica*: determinar en tiempo de ejecución el método al que se llamará según la clase del objeto al que apunte.

```
public class TestComestible {
 public static void main(String[] args) {
   Comestible[] objetos = {new Naranja(), new Pollo(), new Manzana()};
   for (Comestible alimento:objetos) {
       System.out.println(alimento.comoComer());
   }
  }
}
```

Ligadura dinámica usando variables de una interfaz

Se pueden declarar variables cuyo tipo es una interfaz para referenciar objetos de clases que implementan esa interfaz.

- Esto permite usarlas para la *ligadura dinámica*: determinar en tiempo de ejecución el método al que se llamará según la clase del objeto al que apunte.

```
public class TestComestible {
 public static void main(String[] args) {
   Comestible[] objetos = {new Naranja(), new Pollo(), new Manzana()};
   for (Comestible alimento:objetos) {
       System.out.println(alimento.comoComer());
   }
  }
}
```

```
Naranja: Hacer zumo de naranja
Pollo: Freirlo
Manzana: Hacer sidra de manzana
```

Ligadura dinámica usando variables de una interfaz

Sería un error de compilación, usar una variable de una interfaz, para apuntar a un objeto de una clase que no lo implemente.

Ligadura dinámica usando variables de una interfaz

Sería un error de compilación, usar una variable de una interfaz, para apuntar a un objeto de una clase que no lo implemente.

```
public class TestComestible {
 public static void main(String[] args) {
   Comestible[] objetos = {new Tigre(), // Error de compilacion
                           new Pollo(),
                           new Manzana()};
   for (Comestible alimento:objetos) {
       System.out.println(alimento.comoComer());
   }
  }
}
```

Ligadura dinámica usando variables de una interfaz

Sería un error de compilación, usar una variable de una interfaz, para apuntar a un objeto de una clase que no lo implemente.

```
public class TestComestible {
 public static void main(String[] args) {
   Comestible[] objetos = {new Tigre(), // Error de compilacion
                           new Pollo(),
                           new Manzana()};
   for (Comestible alimento:objetos) {
       System.out.println(alimento.comoComer());
   }
  }
}
```

```
TestComestible.java:3: error: incompatible types
    Comestible[] objetos = {new Tigre(),
                            ^
  required: Comestible
  found:    Tigre
1 error
```

Variables en interfaces

Variables en interfaces

Las variables se usan para definir constantes compartidas en múltiples clases.

Variables en interfaces

Variables en interfaces

Las variables se usan para definir constantes compartidas en múltiples clases.

```
import java.util.Random;
interface ConstantesCompartidas {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
class Pregunta implements ConstantesCompartidas {
  Random rand = new Random();
  int preguntar() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30) return NO;          // 30%
    else if (prob < 60) return YES;    // 30%
    else if (prob < 75) return MAYBE; // 15%
    else if (prob < 85) return LATER; // 10%
    else if (prob < 98) return SOON;   // 13%
    else return NEVER;                 // 2%
  }
}
```

Variables en interfaces

```
class Prueba implements ConstantesCompartidas {
  static void answer(int resultado) {
    switch(resultado) {
      case NO:
        System.out.println("No"); break;
      case YES:
        System.out.println("Si"); break;
      case MAYBE:
        System.out.println("Puede ser"); break;
      case LATER:
        System.out.println("Mas tarde"); break;
      case SOON:
        System.outVariables en.println("Pronto"); break;
      case NEVER:
        System.out.println("Nunca"); break;
    }
  }
  public static void main(String args[]) {
    Pregunta q = new Pregunta();
    answer(q.preguntar());
    answer(q.preguntar());
    answer(q.preguntar());
    answer(q.preguntar());
  }
}
```

Variables en interfaces

```
class Prueba implements ConstantesCompartidas {
  static void answer(int resultado) {
    switch(resultado) {
      case NO:
        System.out.println("No"); break;
      case YES:
        System.out.println("Si"); break;
      case MAYBE:
        System.out.println("Puede ser"); break;
      case LATER:
        System.out.println("Mas tarde"); break;
      case SOON:
        System.outVariables en.println("Pronto"); break;
      case NEVER:
        System.out.println("Nunca"); break;
    }
  }
  public static void main(String args[]) {
    Pregunta q = new Pregunta();
    answer(q.preguntar());
    answer(q.preguntar());
    answer(q.preguntar());
    answer(q.preguntar());
  }
}
```

```
No
Puede ser
Puede ser
Si
```

Extensión de interfaces

Extensión de interfaces

Una interfaz puede extender a otra utilizando la palabra reservada **extends**. Una clase que implemente una interfaz que herede de otra, debe implementar todos los métodos de la cadena de herencia.

```
interface A {
  void metodo1();
  void metodo2();
}
interface B extends A {
  void metodo3();
}
class MiClase implements B {
  public void metodo1() {
    System.out.println("Implemento metodo1().");
  }
  public void metodo2() {
    System.out.println("Implemento metodo2().");
  }
  public void metodo3() {
    System.out.println("Implemento metodo3().");
  }
}
```

```
class Prueba {
  public static void main(String arg[]) {
    MiClase ob = new MiClase();
    ob.metodo1();
    ob.metodo2();
    ob.metodo3();
  }
}
```

Extensión de interfaces

Extensión de interfaces

Una interfaz puede extender a otra utilizando la palabra reservada **extends**. Una clase que implemente una interfaz que herede de otra, debe implementar todos los métodos de la cadena de herencia.

```
interface A {
  void metodo1();
  void metodo2();
}
interface B extends A {
  void metodo3();
}
class MiClase implements B {
  public void metodo1() {
    System.out.println("Implemento metodo1().");
  }
  public void metodo2() {
    System.out.println("Implemento metodo2().");
  }
  public void metodo3() {
    System.out.println("Implemento metodo3().");
  }
}
```

```
class Prueba {
  public static void main(String arg[]) {
    MiClase ob = new MiClase();
    ob.metodo1();
    ob.metodo2();
    ob.metodo3();
  }
}
```

```
Implemento metodo1().
Implemento metodo2().
Implemento metodo3().
```

Contenido del tema

Interfaz Cloneable

Interfaz Cloneable

Especifica que un objeto puede ser clonado.

- Para hacer una copia de un objeto podemos usar el método `clone()` (clase `Object`).
- Solo los objetos de clases que implementan `Cloneable` pueden ser clonados.
- El interfaz `Cloneable` no contiene ningún método ni constante, y está definido así:

```
package java.lang;
public interface Cloneable {
}
```

- Muchas clases en la biblioteca de Java implementan `Cloneable`: `Date`, `Calendar`, `ArrayList`.

Interfaz Cloneable

Ejemplo de uso con Calendar:

```
Calendar calendar = new GregorianCalendar(2013, 2, 1);
Calendar calendar1 = calendar;
Calendar calendar2 = (Calendar)calendar.clone();
System.out.println("calendar == calendar1 es "
                   + (calendar == calendar1));
System.out.println("calendar == calendar2 es "
                   + (calendar == calendar2));
System.out.println("calendar.equals(calendar2) es "
                   + calendar.equals(calendar2));
```

Interfaz Cloneable

Ejemplo de uso con Calendar:

```
Calendar calendar = new GregorianCalendar(2013, 2, 1);
Calendar calendar1 = calendar;
Calendar calendar2 = (Calendar)calendar.clone();
System.out.println("calendar == calendar1 es "
                   + (calendar == calendar1));
System.out.println("calendar == calendar2 es "
                   + (calendar == calendar2));
System.out.println("calendar.equals(calendar2) es "
                   + calendar.equals(calendar2));
```

```
calendar == calendar1 es true
calendar == calendar2 es false
calendar.equals(calendar2) es true
```

Interfaz Cloneable

Ejemplo de uso con ArrayList:

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 es " + list1);
System.out.println("list2 es " + list2);
System.out.println("list3 es " + list3);
```

Interfaz Cloneable

Ejemplo de uso con ArrayList:

```
ArrayList<Double> list1 = new ArrayList<>();
list1.add(1.5);
list1.add(2.5);
list1.add(3.5);
ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
ArrayList<Double> list3 = list1;
list2.add(4.5);
list3.remove(1.5);
System.out.println("list1 es " + list1);
System.out.println("list2 es " + list2);
System.out.println("list3 es " + list3);
```

```
list1 es [2.5, 3.5]
list2 es [1.5, 2.5, 3.5, 4.5]
list3 es [2.5, 3.5]
```

Interfaz Cloneable

Ejemplo de uso con arrays:

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 es " + list1[0] + ", " + list1[1]);
System.out.println("list2 es " + list2[0] + ", " + list2[1]);
```

Interfaz Cloneable

Ejemplo de uso con arrays:

```
int[] list1 = {1, 2};
int[] list2 = list1.clone();
list1[0] = 7;
list2[1] = 8;
System.out.println("list1 es " + list1[0] + ", " + list1[1]);
System.out.println("list2 es " + list2[0] + ", " + list2[1]);
```

```
list1 es 7, 2
list2 es 1, 8
```

Interfaz Cloneable

Definiendo clases clonables

Para que una de nuestras clases sea clonable debe:

- Implementar el interfaz `Cloneable`.
- Sobreescribir el método `clone()`.
- La llamada a `clone()` puede lanzar excepciones `CloneNotSupportedException` que debemos capturar obligatoriamente.
  Esta excepción se lanzaría si la clase del objeto con el que usamos `clone()` no implementa `Cloneable`.

Interfaz Cloneable

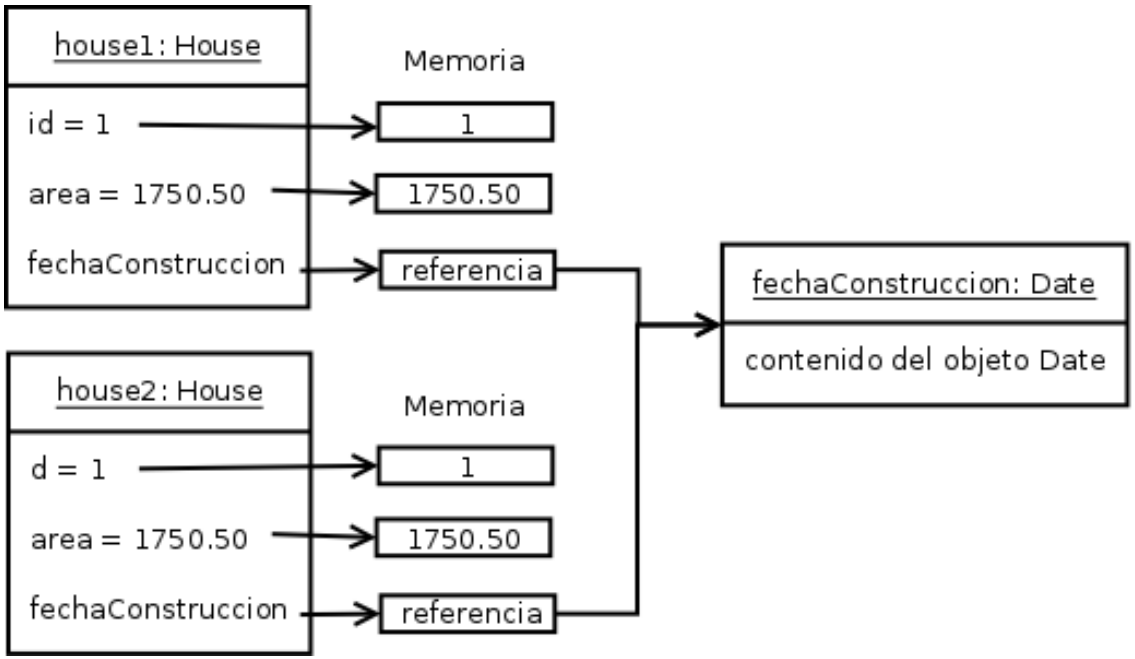
```
public class House implements Cloneable, Comparable {
  private int id;
  private double area;
  private java.util.Date fechaConstruccion;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    fechaConstruccion = new java.util.Date();
  }

  public double getId() {
    return id;
  }
  public double getArea() {
    return area;
  }
  public java.util.Date getFechaConstruccion() {
    return fechaConstruccion;
  }
```

Interfaz Cloneable

```
  public Object clone() {
    try{
      return super.clone();
    }catch(CloneNotSupportedException ex){
      System.out.println(
            "CloneNotSupportedException: no se puede clonar");
      return null;
    }
  }
  public int compareTo(Object o) {
    if (area > ((House)o).area)
      return 1;
    else if (area < ((House)o).area)
      return -1;
    else
      return 0;
  }
}
```

Interfaz Cloneable

```
public class TestHouse {
  public static void main(String[] args) {
    House house1 = new House(1, 1750.0);
    House house2 = (House) (house1.clone());
  }
}
```



Interfaz Cloneable

Funcionamiento por defecto del método clone() de la clase Object

- Si un dato de instancia es de tipo primitivo, su valor es copiado. (Ejemplo: `area` de tipo `double`)
- Si un dato de instancia es de una clase, su referencia es copiada. (Ejemplo: `fechaConstruccion` de tipo `Date`)
- Este comportamiento es conocido como *shallow copy* (copia superficial).

Interfaz Cloneable

Deep copy: copias profundas

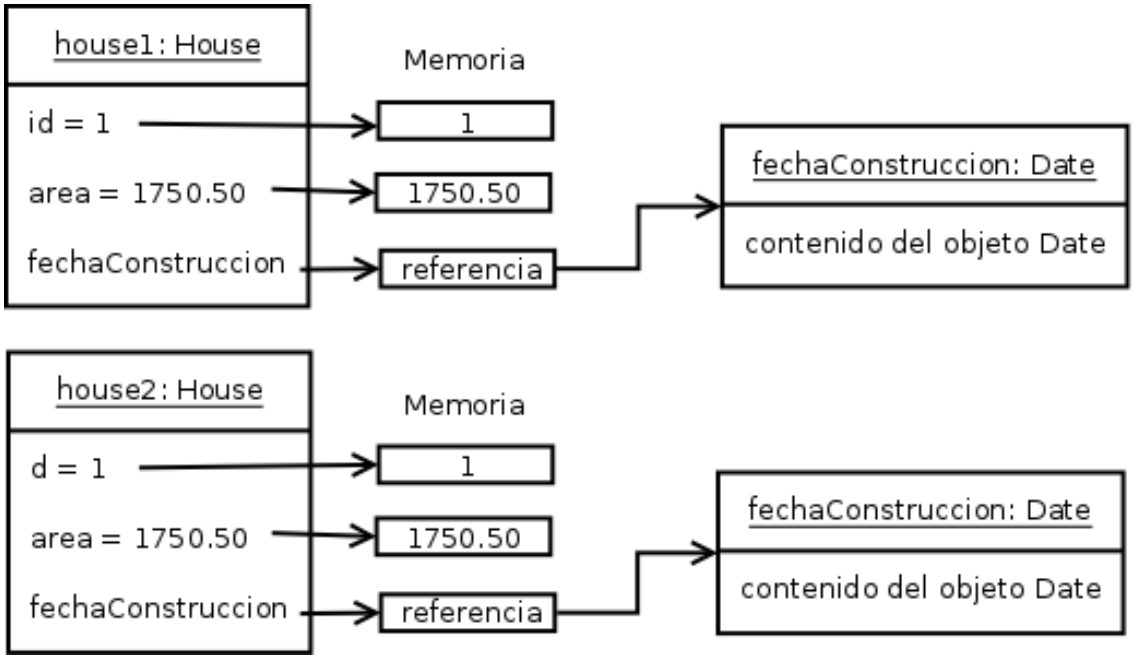
Para que `clone()` haga una copia profunda, además de llamar a `super.clone()`, debe usar `clone()` con los objetos contenidos en la clase.

```
public Object clone() throws CloneNotSupportedException {
  House houseClone = (House)super.clone();
  houseClone.fechaConstruccion = (Date)(fechaConstruccion.clone());
  return houseClone;
}
```

o bien

```
public Object clone(){
  try{
    House houseClone = (House)super.clone();
    houseClone.fechaConstruccion = (Date)(fechaConstruccion.clone());
    return houseClone;
  }catch(CloneNotSupportedException ex){
    System.out.println(" no se puede clonar");
    return null;
  }
```

Interfaz Cloneable



Contenido del tema

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**
    - Pueden acceder directamente a todos los miembros de la clase externa (incluso los privados).

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**
    - Pueden acceder directamente a todos los miembros de la clase externa (incluso los privados).
    - Cada objeto de la clase interna tiene asociado un objeto de la clase externa (que no puede cambiarse una vez creado el objeto de la clase interna).

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**
    - Pueden acceder directamente a todos los miembros de la clase externa (incluso los privados).
    - Cada objeto de la clase interna tiene asociado un objeto de la clase externa (que no puede cambiarse una vez creado el objeto de la clase interna).
- Pueden declararse públicas, protegidas, de paquete o privadas.

Clases internas

Clase interna

Clase (o interfaz) definida dentro del ámbito de otra clase o interfaz.

- Hay dos tipos de clases internas:
  - Clases internas **estáticas**.
    - No pueden acceder a los miembros de la clase externa directamente. Necesitan un objeto de la clase externa para acceder a los miembros no estáticos (datos y métodos) de la clase externa.
  - Clases internas **no estáticas**
    - Pueden acceder directamente a todos los miembros de la clase externa (incluso los privados).
    - Cada objeto de la clase interna tiene asociado un objeto de la clase externa (que no puede cambiarse una vez creado el objeto de la clase interna).
- Pueden declararse públicas, protegidas, de paquete o privadas.
- El nombre completo de una clase interna es: *paquete.claseExterna.claseInterna*

Clases internas no estáticas

Clase interna no estática

Los métodos de una clase interna no estática tienen acceso a todas las variables y métodos de la clase externa de la misma forma que cualquier otro método no estático de la clase externa.

```
class ClaseExterna {
  int dato = 100;
  void test() {
    ClaseInterna objetoClaseInterna =
                 new ClaseInterna();
    objetoClaseInterna.display();
  }
  class ClaseInterna {
    void display() {
      System.out.println("display: dato = " + dato);
    }
  }
}
```

```
class DemoClaseInterna {
  public static void main(String args[]) {
    ClaseExterna objeto = new ClaseExterna();
    objeto.test();
  }
}
```

Clases internas no estáticas

Clase interna no estática

Los métodos de una clase interna no estática tienen acceso a todas las variables y métodos de la clase externa de la misma forma que cualquier otro método no estático de la clase externa.

```
class ClaseExterna {
  int dato = 100;
  void test() {
    ClaseInterna objetoClaseInterna =
                  new ClaseInterna();
    objetoClaseInterna.display();
  }
  class ClaseInterna {
    void display() {
      System.out.println("display: dato = " + dato);
    }
  }
}
```

```
class DemoClaseInterna {
  public static void main(String args[]) {
    ClaseExterna objeto = new ClaseExterna();
    objeto.test();
  }
}
```

```
display: dato = 100
```

Otro ejemplo de clase interna no estática

El siguiente ejemplo implementa el patrón iterador.

```
interface Selector {
  boolean fin();
  Object actual();
  void siguiente();
}
public class Secuencia {
  private Object[] array;
  private int siguiente = 0;
  public Secuencia(int tamano) {
    array = new Object[tamano];
  }
  public void aniadir(Object x) {
    if(siguiente < array.length)
      array[siguiente++] = x;
  }
  private class SelectorSecuencia implements Selector {
    private int i = 0;
    public boolean fin() { return i == array.length; }
    public Object actual() { return array[i]; }
    public void siguiente() { if(i < array.length) i++; }
  }
  public Selector selector() {
    return new SelectorSecuencia();
  }
}
```

```
public class DemoClaseInterna{
  public static void main(String[] args) {
    Secuencia secuencia = new Secuencia(10);
    for(int i = 0; i < 10; i++)
      secuencia.aniadir(Integer.toString(i));
    Selector selector = secuencia.selector();
    while(!selector.fin()) {
      System.out.print(selector.actual() + " ");
      selector.siguiente();
    }
  }
}
```

Otro ejemplo de clase interna no estática

El siguiente ejemplo implementa el patrón iterador.

```
interface Selector {
  boolean fin();
  Object actual();
  void siguiente();
}
public class Secuencia {
  private Object[] array;
  private int siguiente = 0;
  public Secuencia(int tamano) {
    array = new Object[tamano];
  }
  public void aniadir(Object x) {
    if(siguiente < array.length)
      array[siguiente++] = x;
  }
  private class SelectorSecuencia implements Selector {
    private int i = 0;
    public boolean fin() { return i == array.length; }
    public Object actual() { return array[i]; }
    public void siguiente() { if(i < array.length) i++; }
  }
  public Selector selector() {
    return new SelectorSecuencia();
  }
}
```

```
public class DemoClaseInterna{
  public static void main(String[] args) {
    Secuencia secuencia = new Secuencia(10);
    for(int i = 0; i < 10; i++)
      secuencia.aniadir(Integer.toString(i));
    Selector selector = secuencia.selector();
    while(!selector.fin()) {
      System.out.print(selector.actual() + " ");
      selector.siguiente();
    }
  }
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Notas sobre clases internas no estáticas

- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.

Notas sobre clases internas no estáticas

- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.
- Un objeto de una clase interna no puede crearse sin un objeto de la clase externa.

Notas sobre clases internas no estáticas

- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.
- Un objeto de una clase interna no puede crearse sin un objeto de la clase externa.
- Para crear un objeto de la clase interna usaremos:

```
objetoClaseExterna.new claseInterna(argumentos)
```

```
public class ClaseExterna {
  public class ClaseInterna {}
  public static void main(String[] args) {
    ClaseExterna objetoCE = new ClaseExterna();
    ClaseExterna.ClaseInterna objetoCI = objetoCE.new ClaseInterna();
  }
}
```

Notas sobre clases internas no estáticas

- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.
- Un objeto de una clase interna no puede crearse sin un objeto de la clase externa.
- Para crear un objeto de la clase interna usaremos:

```
objetoClaseExterna.new claseInterna(argumentos)
```

```
public class ClaseExterna {
  public class ClaseInterna {}
  public static void main(String[] args) {
    ClaseExterna objetoCE = new ClaseExterna();
    ClaseExterna.ClaseInterna objetoCI = objetoCE.new ClaseInterna();
  }
}
```

- En métodos no estáticos de la clase externa, no es necesario poner `objetoClaseExterna`.

Notas sobre clases internas no estáticas

- El ***estado*** (*data state*) de un objeto de una clase interna no estática lo forman sus datos miembro, y los del objeto usado para crearlo.
- Un objeto de una clase interna no puede crearse sin un objeto de la clase externa.
- Para crear un objeto de la clase interna usaremos:

```
objetoClaseExterna.new claseInterna(argumentos)
```

```
public class ClaseExterna {
  public class ClaseInterna {}
  public static void main(String[] args) {
    ClaseExterna objetoCE = new ClaseExterna();
    ClaseExterna.ClaseInterna objetoCI = objetoCE.new ClaseInterna();
  }
}
```

- En métodos no estáticos de la clase externa, no es necesario poner `objetoClaseExterna`.
- Las clases internas no estáticas no pueden contener miembros **static**.

Clases internas locales

Clase interna local

Clase interna definida dentro de cualquier bloque de código.

- No pueden declararse como **public**, **private** o **protected**.
- Sólo pueden usarse en el método en que se definen.

```
class ClaseExterna {
  int dato = 100;

  void metodo() {
    for(int i=0; i<10; i++) {
      class ClaseInterna {
        void display() {
          System.out.println("display: dato = " + dato);
        }
      }
      ClaseInterna objetoCI = new ClaseInterna();
      objetoCI.display();
    }
  }
}
```

```
class DemoClaseInternaLocal{
  public static void main(String args[]) {
    ClaseExterna objetoCE = new ClaseExterna();
    objetoCE.metodo();
  }
}
```

Clases internas locales anónimas

- Se crean usando la siguiente sintaxis:

  `new [clase_o_interfaz()]{cuerpo de la clase}`

- No pueden tener constructores.
- El nombre opcional `clase_o_interfaz` es el nombre de una clase que se extiende o una interfaz que se implementa. Si se omite, la clase anónima extiende **Object**.
- Si incluimos parámetros en la sentencia `new`, éstos serán pasados al constructor de la superclase.

```
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="DemoClaseInternaAnonima" width=200 height=100>
  </applet>
*/
public class DemoClaseInternaAnonima extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Boton de raton pulsado");
      }
    });
  }
}
```

Clases internas estáticas

Clase interna estática

Los métodos de estas clases no pueden acceder directamente a los miembros no estáticos de la clase externa. No necesitan un objeto de la clase externa para crear un objeto de la clase interna.

- Se definen anteponiendo el modificador **static** en la definición de la clase.

Clases internas estáticas

Clase interna estática

Los métodos de estas clases no pueden acceder directamente a los miembros no estáticos de la clase externa. No necesitan un objeto de la clase externa para crear un objeto de la clase interna.

- Se definen anteponiendo el modificador **static** en la definición de la clase.
- Pueden definirse también dentro de una interfaz, en cuyo caso son automáticamente **public** y **static**.

Clases internas estáticas

Clase interna estática

Los métodos de estas clases no pueden acceder directamente a los miembros no estáticos de la clase externa. No necesitan un objeto de la clase externa para crear un objeto de la clase interna.

- Se definen anteponiendo el modificador **static** en la definición de la clase.
- Pueden definirse también dentro de una interfaz, en cuyo caso son automáticamente **public** y **static**.
- Pueden contener variables y métodos **static**, y otras clases internas **static** (a diferencia de las clases internas no estáticas).

Ejemplo de clase interna estática

El siguiente ejemplo define una clase interna estática `Prueba` que se utiliza para probar el funcionamiento de la clase externa.

```
public class ClaseExterna {
  public void metodo() {
    System.out.println("metodo()");
  }
  public static class Prueba {
    public static void main(String[] args) {
      ClaseExterna objetoCE = new ClaseExterna();
      objetoCE.metodo();
    }
  }
}
```

Otro ejemplo de clase interna estática

```
import java.util.Hashtable;
import java.util.Enumeration;
public class Grafo {
   private Hashtable<String,Nodo>  listaNodos =
           new Hashtable<String,Nodo>();
   public void aniadirNodo( int x, int y ) {
      Nodo n = new Nodo( x, y );
      if ( ! listaNodos.containsKey( n.key() ) )
         listaNodos.put( n.key(), n );
   }
   public String toString() {
      StringBuffer sb = new StringBuffer( "[ " );
      Enumeration e = listaNodos.elements();
      while ( e.hasMoreElements() )
         sb.append( e.nextElement().toString() +
                    " " );
      sb.append( "]" );
      return sb.toString();
   }
   public static void main( String[] args ) {
      System.out.println( "Creando el grafo" );
      Grafo g = new Grafo();
      System.out.println( "Insertando nodos" );
      g.aniadirNodo( 4, 5 );
      g.aniadirNodo( -6, 11 );
      System.out.println( g );
   }
   private static class Nodo {
      private int x, y;
      public Nodo( int x, int y ) {
         this.x = x;
         this.y = y;
       }
      public String key() {
         return x + "," + y;
      }
      public String toString() {
         return "(" + x + "," + y + ")";
      }
   } // fin de clase Nodo
} // fin de clase Grafo
```

Otro ejemplo de clase interna estática

```
import java.util.Hashtable;
import java.util.Enumeration;
public class Grafo {
   private Hashtable<String,Nodo>  listaNodos =
           new Hashtable<String,Nodo>();
   public void aniadirNodo( int x, int y ) {
      Nodo n = new Nodo( x, y );
      if ( ! listaNodos.containsKey( n.key() ) )
         listaNodos.put( n.key(), n );
   }
   public String toString() {
      StringBuffer sb = new StringBuffer( "[ " );
      Enumeration e = listaNodos.elements();
      while ( e.hasMoreElements() )
         sb.append( e.nextElement().toString() +
                    " " );
      sb.append( "]" );
      return sb.toString();
   }
   public static void main( String[] args ) {
      System.out.println( "Creando el grafo" );
      Grafo g = new Grafo();
      System.out.println( "Insertando nodos" );
      g.aniadirNodo( 4, 5 );
      g.aniadirNodo( -6, 11 );
      System.out.println( g );
   }
```

```
   private static class Nodo {
      private int x, y;
      public Nodo( int x, int y ) {
         this.x = x;
         this.y = y;
       }
      public String key() {
         return x + "," + y;
      }
      public String toString() {
         return "(" + x + "," + y + ")";
      }
   } // fin de clase Nodo
} // fin de clase Grafo
```

```
Creando el grafo
Insertando nodos
[ (-6,11) (4,5) ]
```