
TEMA 5:

Composición y Herencia

Programación II - 2017/2018

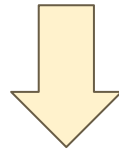
Pedro Cuesta Morales, Baltasar García Pérez-Schofield,
Encarnación González Rufino (Dpto. de Informática)

Índice

1. Introducción
2. Composición
3. Herencia
 - 3.1. Comportamiento de los constructores
 - 3.2. Reescritura de métodos
 - 3.3. Acceso a los métodos en la superclase
 - 3.4. Conversión a la superclase o upcasting
 - 3.5. Conversión a la subclase o downcasting
 - 3.6. Clases abstractas
 - 3.7. Evitando que una clase se convierta en superclase
4. Revisando la visibilidad de miembros
5. Revisando las excepciones
 - 5.1. Clases anidadas
6. Interfaces

1. Introducción

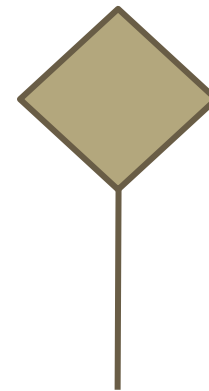
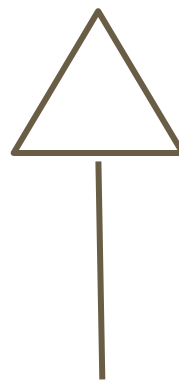
Programación Orientada a Objetos
Clases que se relacionan entre sí



Diseño - UML: *Diagrama de Clases*

Tres de las principales relaciones entre clases:

- Herencia
- Composición
- Asociación

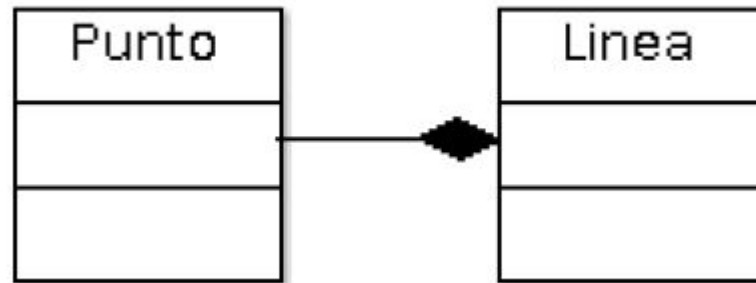


2. Composición

Relación entre clases → “es parte de”

Una de las clases (contenida) aparece como atributo de otra (contenedora)

Un objeto forma parte de otro objeto



p.e. la clase **Punto** es parte de la clase **Linea** (una línea está formada por dos puntos: comienzo y final)

```
class Linea {
    private Punto org;
    private Punto fin;

    public Linea(int x1, int y1, int x2, int y2)
    {
        this( new Punto( x1, y1 ), new Punto( x2, y2 ) );
    }

    public Linea(Punto i, Punto f)
    {
        org = i;
        fin = f;
    }
}
```

1: <http://ideone.com/vVQyFr>

```
public static void main (String[] args) throws java.lang.Exception
{
    Linea l1 = new Linea( 1, 1, 2, 2 );
    Linea l2 = new Linea( new Punto( 4, 5 ), new Punto( 6, 7 ) );

    System.out.println( l1 );
    System.out.println( l2 );
}
```

2. Composición

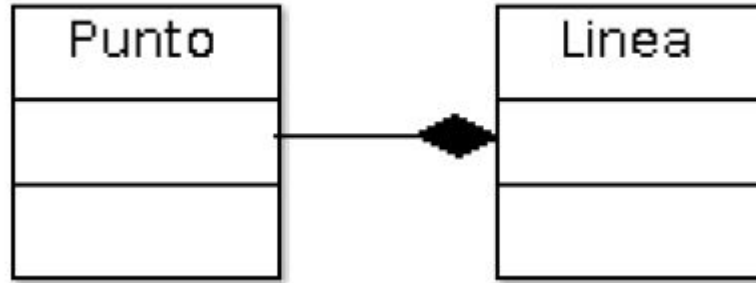
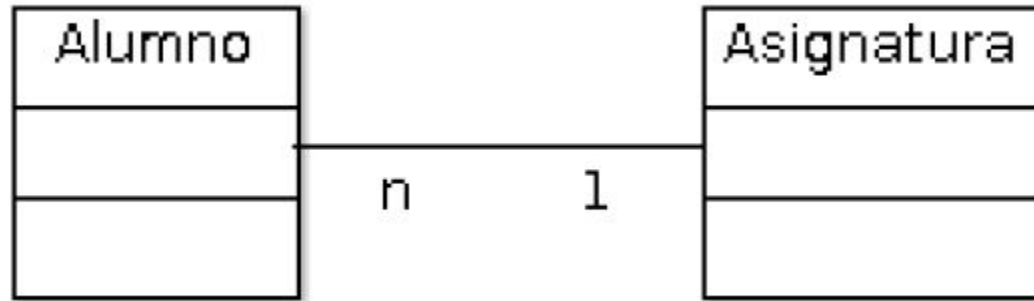


Diagrama de clases:

- ◆ Relación más fuerte de composición, el objeto contenido se crea cuando el objeto contenedor es creado, y se destruye cuando el objeto contenedor se destruye: la existencia del objeto contenido no tiene sentido sin el objeto contenedor

Otro ejemplo: **Persona** tiene **Fecha** (fechaNacimiento)

Composición débil o Asociación



Dos objetos colaboren entre ellos, pero que su existencia no sea dependiente el uno del otro

(desaparezca asignatura no supone desaparezca alumno o viceversa)

Representación:

- Línea une clases relacionadas por la asociación
- Cardinalidad: uno a uno (1,1), uno a muchos (1,n) o muchos a muchos (n,n)

2: <http://ideone.com/KHVjVz>

```
class Asignatura {
    private Alumno[] v;
    private String nombre;

    public Asignatura(String n)
    {
        nombre = n;
        v = new Alumno[ 0 ];
    }

    public void inserta(Alumno a)
    {
        final int numAlumnos = v.length;
        Alumno[] v2 = new Alumno[ numAlumnos + 1 ];
        for(int i = 0; i < numAlumnos; ++i) {
            v2[ i ] = v[ i ];
        }
        v2[ numAlumnos ] = a;
        v = v2;
    }
}
```

```
Alumno al1 = new Alumno( "Baltasar" );
Alumno al2 = new Alumno( "Pedro" );
Alumno al3 = new Alumno( "Nanny" );
Asignatura asg1 = new Asignatura( "PRO2" );
Asignatura asg2 = new Asignatura( "ALS" );
asg1.inserta( al1 );
asg1.inserta( al2 );
asg1.inserta( al3 );
asg2.inserta( al1 );
System.out.println( asg1 );
System.out.println( asg2 );
```

```
class Alumno {
    private String nombre;

    public Alumno(String n)
    {
        nombre = n;
    }
}
```


3. Herencia

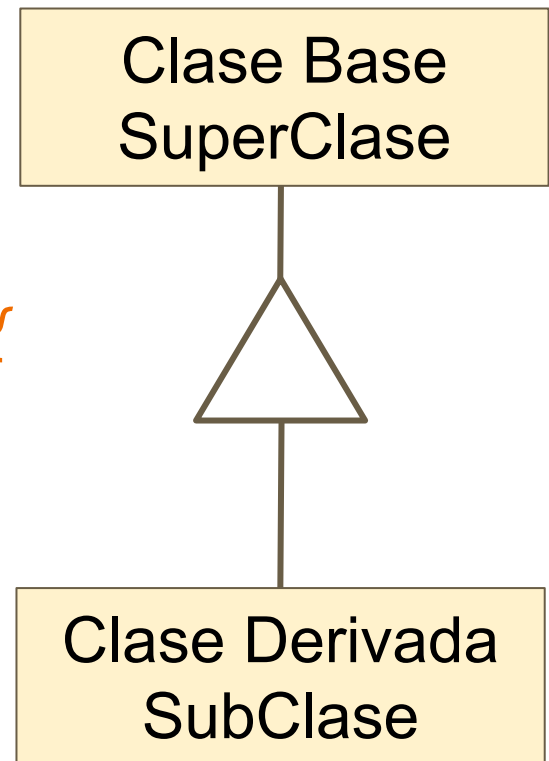
Reutilización de código: crear nuevas clases a partir de clases ya existentes

Relación entre clases: “es un tipo de” (clasificación)

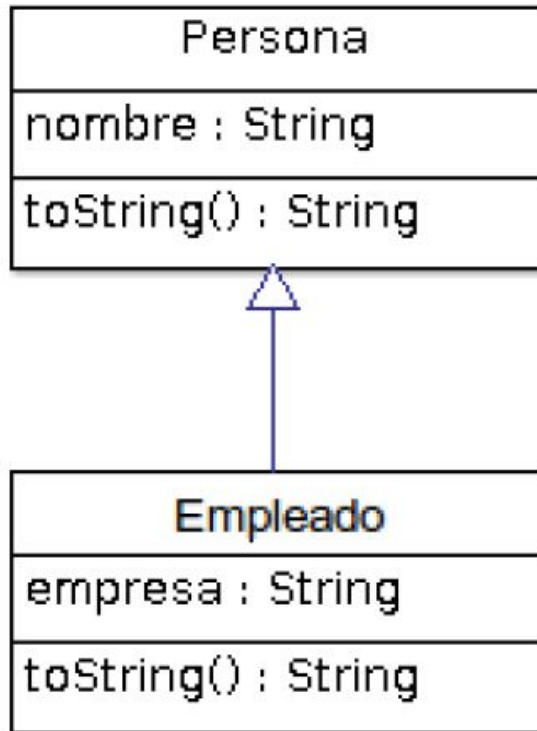
Sintaxis:

```
class SuperClase {  
    // ...  
}  
class SubClase extends SuperClase {  
    // ...  
}
```

JAVA sólo soporta herencia **simple**



3. Herencia



```
class Persona {
    private String nombre;
    public Persona(String n)
    {
        nombre = n;
    }
    public String getNombre()
    {
        return nombre;
    }
    public String toString()
    {
        return getNombre();
    }
}
```

3: <http://ideone.com/SvtCOW>

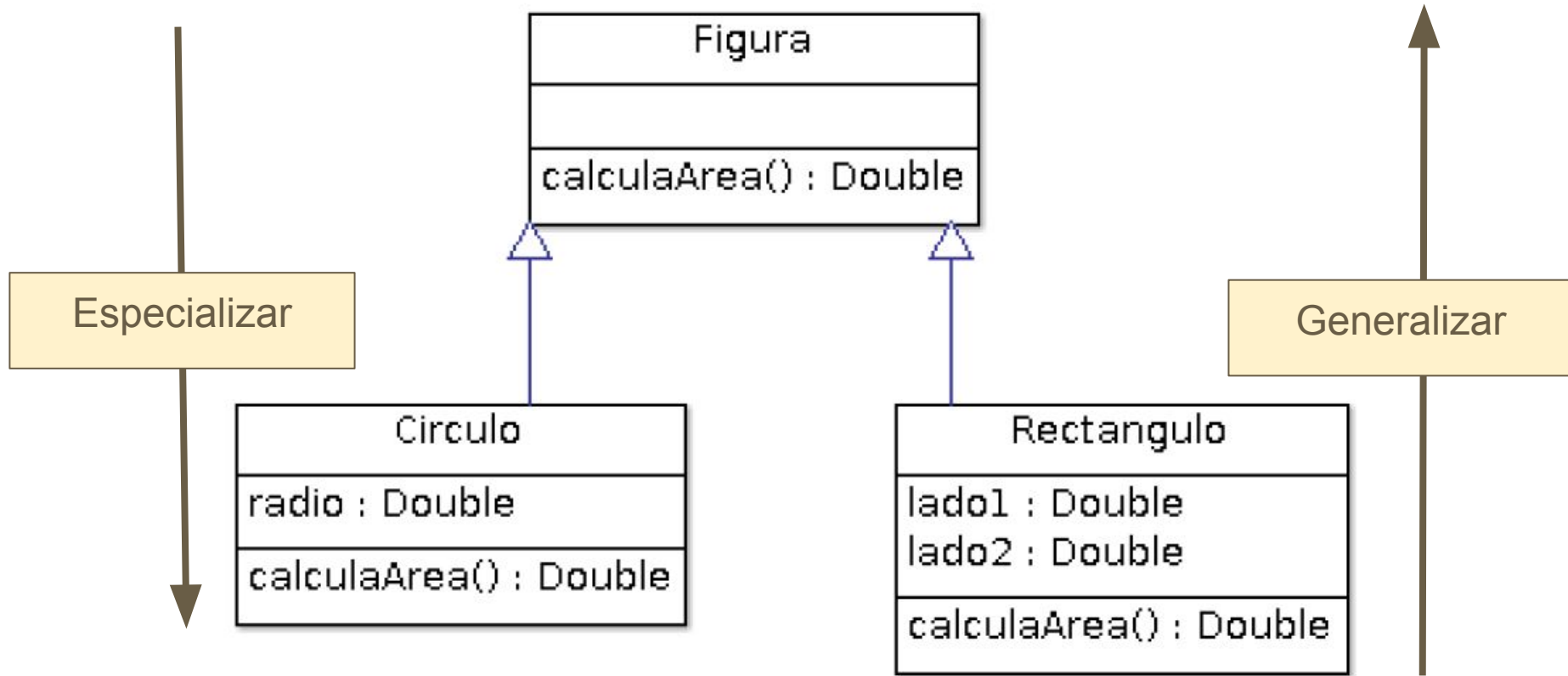
```
class Empleado extends Persona {
    private String empresa;
    public Empleado(String n, String e)
    {
        super( n );
        empresa = e;
    }
    public String getEmpresa()
    {
        return empresa;
    }
    public String toString()
    {
        return getNombre() + ": "
            + getEmpresa();
    }
}
```

Red arrows highlight the `super(n);` call in the `Empleado` constructor and the `return` statement in the `toString()` method.

```
public static void main (String[] args) throws java.lang.Exception
{
    Empleado e1 = new Empleado( "Baltasar", "Uvigo" );
    System.out.println( e1 );
}
```

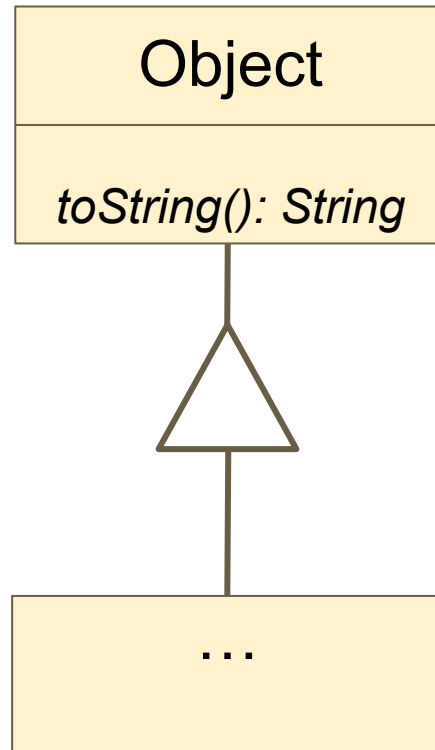
3. Herencia

4: <http://ideone.com/RXD5AC>



1. Clasificación: *Circulo* es un tipo de *Figura* y *Rectangulo* es un tipo de *Figura*
2. Definir la interfaz mínima de las clases derivadas: método *calculaArea()*

3. Herencia



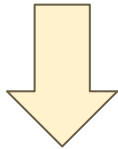
Object: superclase común a todas las clases en JAVA
(herencia implícita)

Cualquier clase es una extensión de la clase *object*

3. Herencia

Abuso de herencia - mal uso de la herencia

Herencia por conveniencia o herencia de implementación



Herencia para reutilizar código

No es posible establecer la relación *“es un tipo de”*

Ejemplos de mal uso:

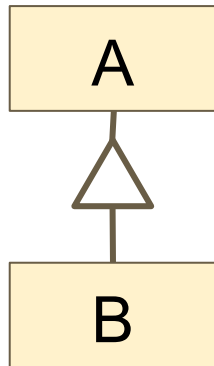
- Clase Alumno extiende la clase Materia
- Clase Empleado extiende la clase Empresa
- ...

3.1. Comportamiento de los constructores con herencia

Jerarquía de herencia:

Constructores se ejecutan en orden “descendente”

1. Crear un objeto de una subclase
2. Constructor de la subclase se detiene y transfiere el control al constructor de la superclase
3. Se repite el proceso si la clase tiene también superclase



Constructor de A
Constructor de B



```
class A {  
    public A(){  
        System.out.println("Constructor de A");  
    }  
}  
class B extends A {  
    public B(){  
        System.out.println( "Constructor de B");  
    }  
}  
public class Ppal {  
    public static void main(String[] args){  
        B objB = new B();  
    }  
}
```

3.1. Comportamiento de los constructores con herencia


1. Constructor de la clase base no tiene parámetros → al crear un objeto de la clase derivada el constructor de la clase base se invoca automáticamente (implícita)
2. Si el constructor de la clase base está parametrizado → el constructor de la clase derivada debe invocar (explícita) al constructor de la clase base:

super(argumentos);

primera sentencia

```
public Persona(String n)
{
    nombre = n;
}
```

```
public Empleado(String n, String e)
{
    super( n );
    empresa = e;
}
```



5: <http://ideone.com/xUml61>

3.2. Reescritura de métodos

Una clase base define un método para que se sobrescriba en sus clases derivadas

@Override → comprobar en tiempo de compilación si se está reescribiendo el método adecuadamente (tipo de retorno o parámetros)

```
class Circulo extends Figura {  
    private double radio;  
    public Circulo(double r) {  
        radio = r;  
    }  
    public double getRadio() {  
        return radio;  
    }  
    @Override  
    public double calculaArea() {  
        return radio * radio * Math.PI;  
    }  
    @Override  
    public String toString()  
    {  
        return String.format( "Circulo de radio %.2f", getRadio() );  
    }  
}
```

6: <http://ideone.com/zeWyZ1>

3.3. Acceso a métodos en la superclase

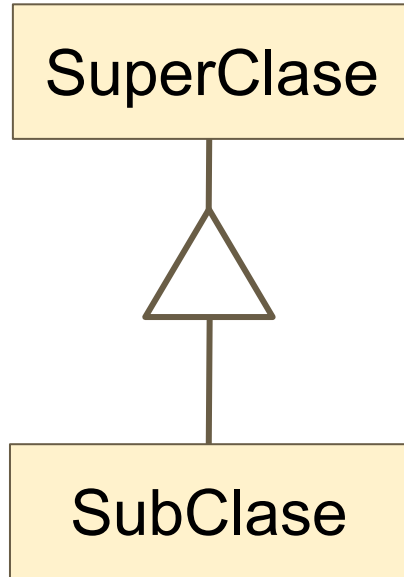
super → permite acceder a la superclase

Una referencia que apunta a una versión del objeto referenciado por *this*, que es del tipo de la superclase

```
class Empleado extends Persona {  
    private String empresa;  
    public Empleado(String n, String e)  
    {  
        super( n );  
        empresa = e;  
    }  
    public String getEmpresa()  
    {  
        return empresa;  
    }  
    public String toString()  
    {  
        return getNombre() + ": "  
            + getEmpresa();  
    }  
}
```

```
class Empleado extends Persona {  
    private String empresa;  
    public Empleado(String n, String e)  
    {  
        super( n );  
        empresa = e;  
    }  
    public String getEmpresa()  
    {  
        return empresa;  
    }  
    @Override  
    public String toString()  
    {  
        return super.toString() + ": "  
            + getEmpresa();  
    }  
}
```

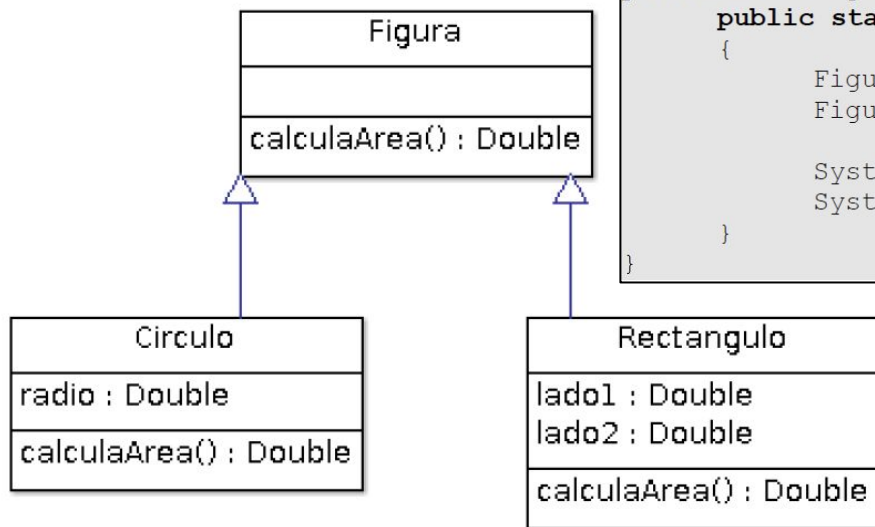
3.4. Conversión a la superclase o upcasting



Herencia → Relación de Tipos

Es posible “tratar” un objeto de la SubClase como si fuera de la SuperClase

Una referencia a un objeto de la SuperClase puede recibir una referencia a un objeto de la SubClase



```
public class Ppal {
    public static void main(String[] args)
    {
        Figura r1 = new Rectangulo( 5, 6 );
        Figura c1 = new Circulo( 1 );

        System.out.format( "%s, superficie: %.2f\n", r1, r1.calculaArea() );
        System.out.format( "%s, superficie: %.2f\n", c1, c1.calculaArea() );
    }
}
```

¿Qué código se ejecuta, el de Figura (devuelve 0.0) o el de las clases derivadas?

3.4. Conversión a la superclase o upcasting

Ventaja: puedan crearse vectores primitivos de la Superclase, con elementos que en realidad son objetos de sus subclases

```
public static void main (String[] args) throws java.lang.Exception
{
    Figura[] figuras = new Figura[ 2 ];

    figuras[ 0 ] = new Rectangulo( 5, 6 );
    figuras[ 1 ] = new Circulo( 1 );
    for(int i = 0; i < figuras.length; ++i) {
        Figura f = figuras[ i ];
        System.out.format( "%s, superficie: %.2f\n", f, f.calculaArea() );
    }
}
```

6: <http://ideone.com/zeWyZ1>

POLIMORFISMO

3.5. Conversión a la subclase o downcasting

(upcasting) Es seguro tratar de apuntar a una subclase con una referencia a la superclase → lo contrario no lo es

En ciertas ocasiones puede ser útil comprobar a qué clase pertenece el objeto al que se apunta con una referencia a la superclase → hacer esto continuamente es un error de diseño en el programa

Operador *instanceof*

boolean resultado = <objeto> instanceof <clase>;

Comprobación en tiempo de ejecución

Devuelve true si el objeto pertenece a dicha clase

```

public static double calculaArea(Figura f)
{
    double toret = 0;
    if ( f instanceof Rectangulo ) {
        Rectangulo r = (Rectangulo) f;
        toret = r.getLado1() * r.getLado2();
    }
    else
    if ( f instanceof Circulo ) {
        Circulo c = (Circulo) f;
        toret = c.getRadio() * c.getRadio() * Math.PI;
    }
    return toret;
}

public static void main (String[] args) throws java.lang.Exception
{
    Figura[] figuras = new Figura[ 2 ];

    figuras[ 0 ] = new Rectangulo( 5, 6 );
    figuras[ 1 ] = new Circulo( 1 );
    for(int i = 0; i < figuras.length; ++i) {
        Figura f = figuras[ i ];
        System.out.format( "%s, superficie: %.2f\n", f, calculaArea(f) );
    }
}

```

- CalcularArea() no definido en Circulo y Rectangulo
- Código poco mantenible: nueva subclase requiere modificar el método estático calculaArea()

8: <http://ideone.com/AwLPan>



```
public static void listaRectangulos(Figura[] figuras)
{
    for(int i = 0; i < figuras.length; ++i) {
        Figura f = figuras[ i ];
        if ( f instanceof Rectangulo ) {
            System.out.format( "%s, superficie: %.2f\n",f, f.calculaArea() );
        }
    }
}

public static void main (String[] args) throws java.lang.Exception
{
    Figura r1 = new Rectangulo( 5, 6 );
    Figura c1 = new Circulo( 1 );

    listaRectangulos( new Figura[]{ r1, c1 } );
}
```


3.6. Clases abstractas

Método abstracto: no tiene implementación (cuerpo o secuencia de instrucciones)

[acceso] abstract tipo nombreMetodo([parametros]);

Todas las subclases tienen que implementar el método

Clase abstracta: clase que contiene un método abstracto

[acceso] abstract class NombreClase { ... }

Imposible crear objetos de una clase abstracta

```
public class Figura {  
    public double calculaArea()  
    {  
        return 0.0;  
    }  
}
```

Clase abstracta pura: todos sus métodos son abstractos

```
public abstract class Figura {  
    public abstract double calculaArea();  
}
```

3.7. Evitando que una clase se convierta en superclase

Para indicar que no se desea que una clase se convierta en superclase → *final*

```
final class <nombre_de_la_clase> {  
    // ...  
}
```

No se podrán crear clases derivadas suyas

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

NOTA: calculaArea() en Cuadrado → clase Rectángulo (a pesar de que calculaArea() sea un método abstracto)

4. Revisitando la visibilidad de miembros

modificador *protected* (atributos o métodos):

- Privado a todos los efectos excepto desde las subclases de la clase donde se define
- No requiere crear un método público de acceso a un atributo cuando solo se desea acceder desde subclases

```
class Candado {  
    protected int clave;  
    private boolean abierto;  
  
    /** Crea un candado, asignando una clave.  
     * @param k La clave, como entero */  
    public Candado(int k)  
    {  
        clave = k;  
        abierto = false;  
    }  
}
```

```
class CandadoMinTresPosiciones extends Candado {  
    public CandadoMinTresPosiciones(int k)  
    {  
        super( k );  
  
        // Clave de menos de tres posiciones?  
        if ( clave < 100 ) {  
            clave += 100;  
        }  
  
        return;  
    }  
}
```

5. Revisando las excepciones

Error → *throw new Exception("...");*

Lanzando un objeto del mismo tipo para todos los errores

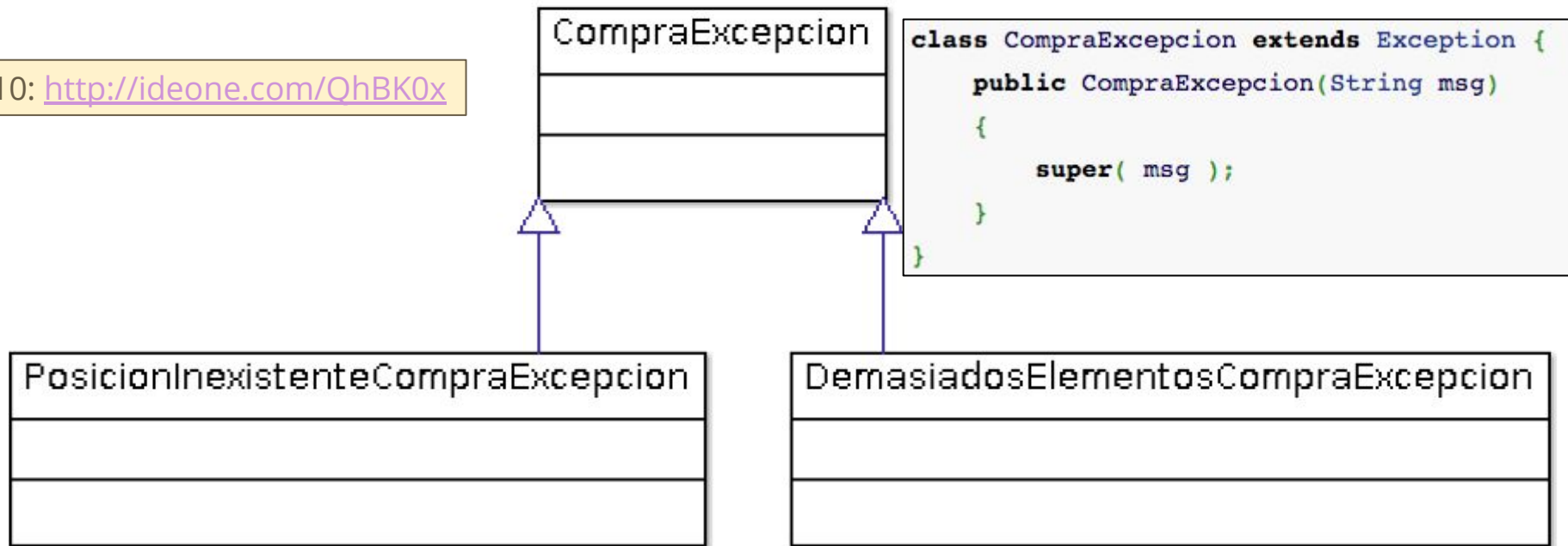
Distinguir diferentes situaciones de error → crear una jerarquía de herencia de excepciones

Herencia → clasificar las excepciones

Sintaxis:

```
class <Excepción_propia> extends Exception {  
    public <Excepción_propia>(String msg) {  
        super( msg );  
    }  
}
```

10: <http://ideone.com/QhBK0x>



```
class ComprExcepcion extends Exception {
    public ComprExcepcion(String msg)
    {
        super( msg );
    }
}
```

```
class DemasiadosElementosComprExcepcion extends ComprExcepcion {
    public DemasiadosElementosComprExcepcion(String msg)
    {
        super( msg );
    }
}

class PosicionInexistenteComprExcepcion extends ComprExcepcion {
    public PosicionInexistenteComprExcepcion(String msg)
    {
        super( msg );
    }
}
```

Lanzar excepciones:

```
/** Agrega un nuevo elemento en la lista  
 * @param elemento El nuevo elemento */  
public void inserta(String elemento) throws DemasiadosElementosCompraExcepcion  
{  
    if ( getNumElementos() >= getMaxElementos() ) {  
        throw new DemasiadosElementosCompraExcepcion( "max. alcanzado, con: " + elemento );  
    }  
  
    elementos[ num ] = elemento;  
    ++num;  
}  
  
/** Devuelve un elemento de la lista  
 * @param i La pos del elemento */  
public String get(int i) throws PosicionInexistenteCompraExcepcion  
{  
    if ( i >= getNumElementos() ) {  
        throw new PosicionInexistenteCompraExcepcion( "pos. no existe: " + i );  
    }  
  
    return elementos[ i ];  
}
```

Capturar excepciones:

```
try {  
    ...  
}  
catch (CompraExcepcion exc)  
{  
    System.err.println(...);  
}  
catch (Exception exc)  
{  
    System.err.println(...);  
}
```

5.1. Clases anidadas

JAVA: es posible crear una clase dentro de otra clase

1. Static: la clase no tiene acceso a los miembros de la clase que la engloba. Accesibles desde fuera (public) con *“NombreClaseContenedora.NombreClaseContenida”*
2. Miembro: creadas y manejadas sólo desde dentro de la clase que las engloba

Ejemplo: las clases para excepciones propias se lanzan desde métodos de la clase *ListaCompra* (*get()* e *inserta()*) → definirlas dentro de la clase como static

11: <http://ideone.com/GPI8S1>

6. Interfaces

Interfaces: clases abstractas puras que definen un conjunto de métodos que tienen que ser implementados obligatoriamente:

```
interface <nombre_de_interface> {  
    <cabecera de método1>;  
    <cabecera de método2>;  
    ...  
    <cabecera de método n>;  
}
```

Una clase implementa una interfaz o varias:

```
class <nombre_de_clase> extends superclase  
    implements interfaz1, interfaz2,...  
{ ...}
```

12: <http://ideone.com/Gd3dBH>

```
interface IGeneradorHTML {  
    public String toHTML();  
}  
  
interface IAsalariado {  
    public double getSalario();  
}
```

```
class Persona implements IGeneradorHTML {  
    private String nombre;  
  
    public Persona(String n)  
    {  
        nombre = n;  
    }  
}
```

```
@Override  
public String toHTML()  
{  
    return "<b>" + getNombre() + "</b>";  
}
```

```
class Empleado extends Persona implements IAsalariado {  
    private String empresa;  
    private double salario;  
  
    public Empleado(String n, String e, double s)  
    {  
        super( n );  
        empresa = e;  
        salario = s;  
    }  
}
```

```
@Override  
public String toHTML()  
{  
    return super.toHTML() + ": <i>" + getEmpresa()  
        + " (" + getSalario() + ")</i>";  
}
```

```
@Override  
public double getSalario()  
{  
    return salario;  
}
```