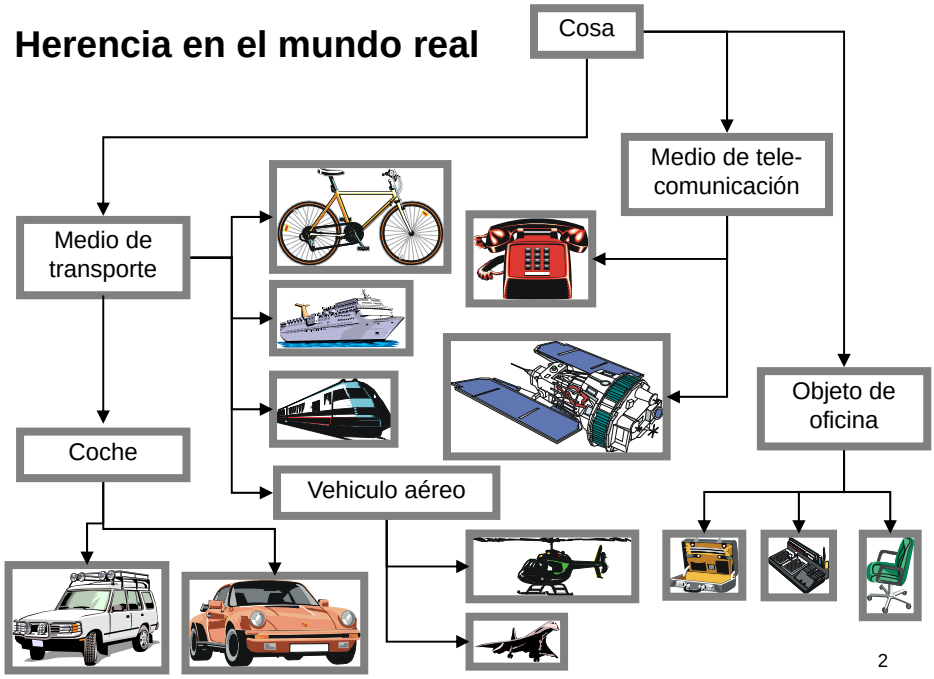
Herencia y Polimorfismo

Dra. Maricela Bravo
Universidad Autónoma Metropolitana
Unidad Azcapotzalco

1

Herencia en el mundo real

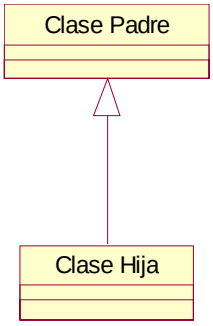


2

Herencia en Java

- Java permite definir una clase como subclase de una clase padre.

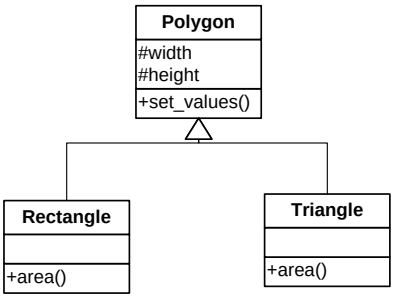
  class *clase_hija* ***extends*** *clase_padre*
  {
  .....
  }



3

Ejemplo de Herencia

```
class Polygon {
 protected int width, height;
 public void set_values (int a, int b) {
 width=a; height=b;} }
```



4

```
class Rectangle extends Polygon {
 public int area() { return (width * height); } }


class Triangle extends Polygon {
 public int area() { return (width * height /
2); }
 public static void main(String[] args) {
 Rectangle rect; Triangle trgl;
 rect = new Rectangle(); trgl = new
Triangle();
 rect.set_values (4,5); trgl.set_values (4,5);
 System.out.print("area" + rect.area() + '\n'
+
 trgl.area() + '\n');
}}
```

5

Constructores y Herencia

- Cuando se declara un obj de una clase derivada, se ejecutan los conts siguiendo el orden de derivación, es decir, primero el de la clase base, y después los conts de las clases derivadas de arriba a abajo.
- Para pasar parámetros al constructor de la clase padre:

  super (para1, para2, ..., paraN)

6

Ejemplo de super

```
class Persona {
    private String nombre;
    private int edad;
    public Persona() {}
    public Persona (String n, int e)
    { nombre = n; edad = e; }
}
class Alumno extends Persona {
    private int curso;
    private String nivelAcademico;
    public Alumno (String n, int e, int c, String nivel) {
        super(n, e);
        curso = c; nivel_academico = nivel;
    }
    public static void main(String[] args) {
        Alumno a = new Alumno("Pepe", 1, 2, "bueno");
    }
}
```

7

Redefinir f. miembros de la clase padre

```
class Persona {
    private String nombre;
    private int edad;
    ...................
    public String toString() { return nombre + edad; }
    public void setEdad(int e) { edad = e; }
}
class Alumno extends Persona {
    private int curso;
    private String nivelAcademico;
    ...................
    public String toString() {
        return super.toString() + curso + nivelAcademico;
    }
    public void setCurso(int c) { curso = c; }
}
```

8

Referencias a objetos de clases hijas

- Si tenemos *ClaseHijo hijo = new ClaseHijo(...);*
- entonces es posible *padre=hijo* donde *padre* es una variable de tipo *ClasePadre*.
  - pero no es posible!! *hijo=padre (*sí que es posible con casting *hijo= (ClaseHijo) padre*)
- Ahora bien:
  - Con *padre* sólo podemos acceder a atributos y métodos def. en la clase padre.

9

Referencias a objetos de clases hijas

```
public static void main(String[] args) {
    Persona p;
    Alumno a = new Alumno("pepe",23,1,"universitario");
    p=a; //ref padre señala al objeto hijo
    // acceso al objeto hijo mediante la ref padre
    p.setEdad(24);
    /* no es posible acceder a un miembro de la clase hija usando
    una ref a la clase padre*/
    p.setCurso(88); // ERROR
    }
```

10

Ejemplo

```
class Persona { ................... }
class Alumno extends Persona {
    ......................
    public String toString() {
        return super.toString() + curso + nivelAcademico;
    }
}
class Profesor extends Persona {
    private String asignatura;
    public Profesor (String n, int e, String asign) {
        super(n, e);
        asignatura = asign;
    }
    public String toString() {
        return super.toString() + asignatura;
    }
}
```

11

POLIMORFISMO

12

Polimorfismo

- Una misma llamada ejecuta diferentes sentencias dependiendo de la clase a la que pertenezca el objeto al que se aplica el método.
- Supongamos que declaramos: *Persona p;*
- Podría suceder que durante la ej. del programa, p referencie a un profesor o a un alumno en distintos momentos, y
- Entonces:
  - Si p referencia a un alumno, con p.toString(), se ejecuta el toString de la clase Alumno.
  - Si p referencia a un profesor, con p.toString(), se ejecuta el toString de la clase Profesor.
- **Enlace dinámico**: Se decide en **tiempo de ejecución** qué método se ejecuta.
- OJO!: Sobrecarga de fs => enlace estático (t. de compilación).

13

Ejemplo de Polimorfismo

```
public static void main(String[] args) {
  Persona v[]=new Persona[10];
// Se introducen alumnos, profesores y personas en v
  for (int i=0 ; i<10; i++)
    /* Se piden datos al usuario de profesor, alumno o persona */
    switch (tipo) {
          case /*  profesor */:  v[i] = new Profesor (….); break;
          case /* alumno */: v[i] = new Alumno(…); break;
          case /* persona */: v[i] = new Persona(…); break;
          default: /* ERROR */ }
    }
  for (int i=0 ; i<10; i++)
    System.out.println(v[i]);  // enlace dinámico con toString()
}
```

14

CLASES ABSTRACTAS

15

Clases Abstractas

- Una **clase abstracta** es una clase de la que no se pueden crear objetos, pero puede ser utilizada como clase padre para otras clases.
- Declaración:

```
abstract class NombreClase {
    ..............
}
```

16

Métodos abstractos

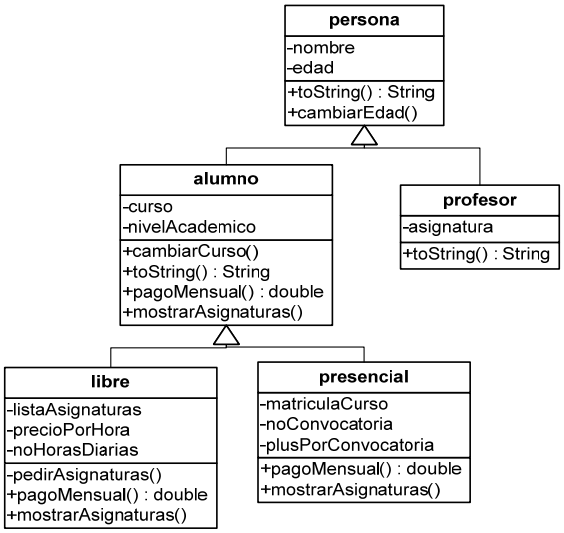
- Tenemos un método f() aplicable a todos los objetos de la clase A.
  - Área de un polígono.
- La implementación del método es completamente diferente en cada subclase de A.
  - Área de un triángulo.
  - Área de un rectángulo.

  .....
- Para declarar un método como abstracto, se pone delante la palabra reservada *abstract* y no define un cuerpo:

  abstract tipo nombreMétodo(....);
- Luego en cada subclase se define un método con la misma cabecera y distinto cuerpo.

17

Ejemplo de clase abstracta



18

Ejemplo de clase abstracta

```
abstract class Alumno extends Persona {
    protected int curso;
    private String nivelAcademico;
    public Alumno (String n, int e, int c, String nivel) {
        super(n, e);
        curso = c; nivelAcademico = nivel;
    }
    public String toString() {
        return super.toString() + curso + nivelAcademico;
    }
    abstract double pagoMensual();
    abstract String getAsignaturas();
}
```

19

Ejemplo de clase abstracta

```
class Libre extends Alumno {
    private String []listaDeAsignaturas;
    private static float precioPorHora=10;
    private int noHorasDiarias;
    private void pedirAsignaturas() {}// se inicializa listaDeAsignaturas
    public double pagoMensual() {
        return precioPorHora*noHorasDiarias*30; }
    public String getAsignaturas() {
        String asignaturas="";
        for (int i=0; i<listaDeAsignaturas.length; i++)
            asignaturas += listaDeAsignaturas[i] + ' ';
        return asignaturas;
    }
    public Libre(String n, int e, int c, String nivel, int horas)
      {super(n,e,c,nivel); noHorasDiarias = horas; pedirAsignaturas(); }
}
```

20

Ejemplo de clase abstracta

```
class Presencial extends Alumno {
    private double matriculaCurso;
    private double plusPorConvocatoria;
    private int noConvocatoria;
    public double pagoMensual()
    { return (matriculaCurso+plusPorConvocatoria*noConvocatoria)/12; }
    public String getAsignaturas(){
        return "todas las del curso " + curso;
    }
    public Presencial(String n, int e, int c, String nivel,
                double mc, double pc, int nc) {
        super(n,e,c,nivel);
        matriculaCurso=mc;
        plusPorConvocatoria=pc;
        noConvocatoria=nc;
    }
}
```

21

Ejemplo de clase abstracta

```
// FUNCIONES GLOBALES
void mostrarAsignaturas(Alumno v[]) {
 for (int i=0; i<v.length; i++)
  System.out.println(v[i].getAsignaturas());
  // enlace dinámico
}
double totalPagos(Alumno v[]) {
 double t=0;
 for (int i=0; i<v.length; i++)
  t += v[i].pagoMensual(); // enlace dinámico
 return t;
}
```

22

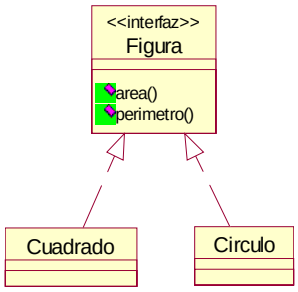
Interfaces

- Podría suceder que los objetos de varias clases compartan la capacidad de ejecutar un cierto conjunto de operaciones.
- Y dependiendo de la clase de objeto, cada operación se realice de diferente manera.
- Ejemplo:
  - Clases: Circulo, Elipse, Triangulo, ...
  - Todas esas clases incluyen los métodos: área, perimetro, cambiarEscala, etc.
- Podríamos definir una interfaz común que agrupe todos los métodos comunes (como métodos abstractos).
- Y luego definir varias clases de modo que implementen una misma interfaz.

23

Ejemplo de Interface

```
public interface Figura {
    abstract double area();
    abstract double perimetro();
}
public class Circulo implements Figura {
    private double radio;
    private static double PI=3.1416;
    .............
    public double area() { return PI*radio*radio; }
    public double perimetro() { return 2*PI*radio; }
}
public class Cuadrado implements Figura {
    private double lado;
    .............
    public double area() { return lado*lado; }
    public double perimetro() { return 4*lado; }
}
```



24

Ejemplo de Interface

- Una interface puede incluir también definiciones de constantes a parte de métodos abstractos.
- Una misma clase puede implementar más de una interface ⇒ Herencia múltiple de interfaces
- Se pueden crear jerarquías de interfaces (con extends!!).
- Se pueden declarar referencias a objetos que implementen una cierta interface.

```
double totalArea(Figura v[]) {
 double t=0;
 for (int i=0; i<v.length; i++)
  t += v[i].area(); // enlace dinámico
 return t;
}
```

25

Clases Abstractas VS Interfaces

- Las clases abstractas son similares a las interfaces en lo siguiente:
  - No se pueden instanciar objetos de ellas
  - Pueden contener una mezcla de métodos abstractos y métodos con implementación.

26

Clases Abstractas VS Interfaces

- Con las clases abstractas
  - Es posible declarar campos que no son static y final
  - Es posible declarar métodos public, protected, y private.
- Con las Interfaces
  - Todos los atributos son automáticamente public, static y final.
  - Todos los métodos que declaras o defines son public.
  - Solamente se puede extender una clase, sea o no abstracta.
  - Es posible implementar cualquier número de interfaces.

27

Cuando utilizar clases Abstractas o Interfaces

- Considera utilizar clases abstractas cuando:
  - Se desea compartir código entre varias clases estrechamente relacionadas.
  - Se espera que las clases que extienden de la clase abstracta posean muchos métodos comunes, o requieran del uso de modificadores de acceso diferentes a public (protected y private).
  - Se necesita declarar atributos non-static y non-final. Est permote definir métodos que pueden accesar y modificar el estado del objeto al cual pertenecen.

28

Cuando utilizar clases Abstractas o Interfaces

- Considerar el uso de Interfaces cuando:
  - Se espera que clases no relacionadas implementen la Interface.
  - Se necesita especificar el comportamiento de un tipo de dato particular, independientemente de quien implementa su comportamiento.

29

Modificadores de acceso

- DEFAULT:
  Si no elegimos ningún modificador, se usa el de por defecto, que sólo puede ser **accedido por clases que están en el mismo paquete**.
- PUBLIC:
  Este nivel de acceso permite a acceder al elemento **desde cualquier clase**, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.

30

Modificadores de aceso

- PRIVATE:
  Es el modificador más restrictivo y especifica que los elementos que lo utilizan **sólo pueden ser accedidos desde la misma clase en la que se encuentran**. Este modificador sólo puede utilizarse sobre los miembros de una clase y sobre interfaces y clases internas, no sobre clases o interfaces de primer nivel, dado que esto no tendría sentido.

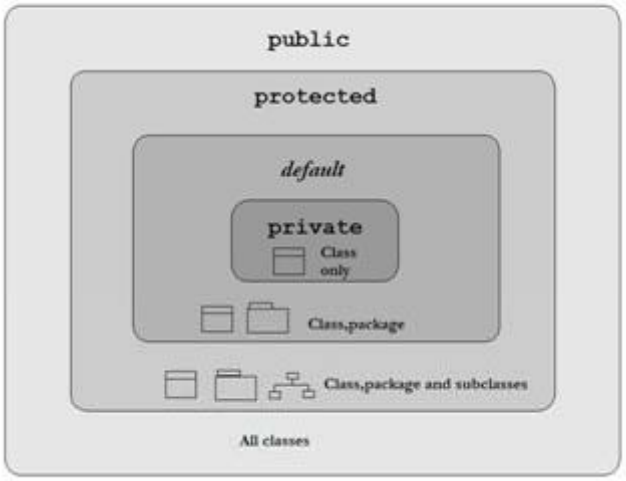
31

Modificadores de acceso

- PROTECTED:
  Indica que los elementos sólo pueden ser accedidos **desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra**, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como private, no tiene sentido a nivel de clases o interfaces no internas.

32

Modificadores de acceso



33

Modificadores de métodos

- STATIC:
  Sirve para crear miembros que pertenecen a la clase, y no a una instancia de la clase. Esto implica, entre otras cosas, que **no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos**.

- En ocasiones es necesario o conveniente generar elementos que tomen un mismo valor para cualquier número de instancias generadas o bien invocar/llamar métodos sin la necesidad de generar instancias, y es bajo estas dos circunstancias que es empleado el calificador *static*.

34

Modificadores de métodos

- Dos aspectos característicos de utilizar el calificador static en un elemento Java son los siguientes:
  - **- No puede ser generada ninguna instancia** (uso de *new*) de un elemento *static* puesto que solo existe una instancia.
  - **-** Todos los elementos definidos dentro de una estructura *static* deben ser *static* ellos mismos, o bien, poseer una instancia ya definida para poder ser invocados.
- *NOTA*: Lo anterior no implica que no puedan ser generadas instancias dentro de un elemento *static*; no es lo mismo llamar/invocar que crear/generar.

35

Modificadores de métodos

- FINAL:
  Indica que una variable, método o clase no se va a modificar, lo cuál puede ser útil para añadir más semántica, por cuestiones de rendimiento, y para detectar errores.
  - - Si una **variable** se marca como *final*, **no se podrá asignar un nuevo valor** a la variable.
    - Si una **clase** se marca como *final*, **no se podrá extender** la clase.
    - Si es un **método** el que se declara como *final*, **no se podrá sobreescribir**.
- *NOTA*: Una variable con modificadores *static* y *final* sería lo más cercano en Java a las constantes de otros lenguajes de programación.

36