1

COMPOSICIÓN, HERENCIA, POLIMORFISMO

COMPOSICIÓN

2

- La composición es la creación de una clase nueva agrupando objetos de clases preexistentes. Se encuentra la composición cuando en una clase se tienen atributos que corresponden con el tipo de una clase.
- Para reconocer la composición podemos preguntarnos si, por ejemplo, el billete "**tiene-un**" cliente.

Clase Cliente

3

```
public class Cliente
{
   private String nombre;
   private String dni;
   private String telefono;
}
```

Ejemplo Composición

4

```
public class Billete
{
   private long numBillete;
   private String fechaHora;
   private int numAsiento;
   private String locSalida;
   private String locLlegada;
   private Cliente cliente;
   private int precio;
.....}
```

DEFINICIÓN

5

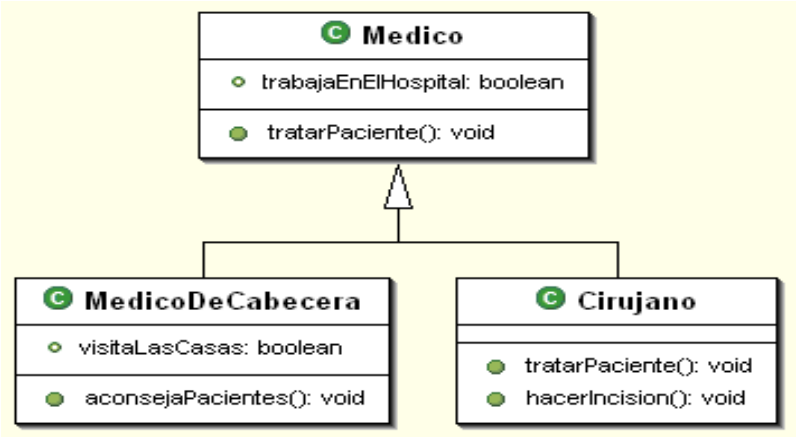
- La herencia es un mecanismo de la P.O.O. que permite construir una clase incorporando de manera implícita todas las características de una clase previamente existente.
- Es la capacidad de una clase de heredar tanto los métodos como los atributos de otra clase.

- La implementación de la herencia se realiza mediante la *keyword:* ***extends***

modificador_acceso class nom_clase extends nom_clase
{
}

Ejemplo

6



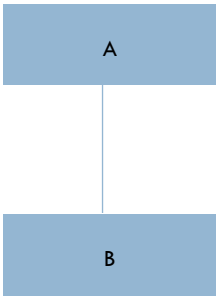
7

```
public class Medico
{
   public boolean
     trabajaEnHospital;
   public void tratarPaciente()
   {
     //Realizar un chequeo.
   }
}
```

```
public class MedicoDeCabecera extends
     Medico
{
     public boolean visitaLasCasas;
      public void aconsejaPacientes()
      {
          //Ofrecer remedios caseros.
       }
}
public class Cirujano extends Medico
{
    public void tratarPaciente()
   {
          //Realizar una operación.
    }
   public void hacerIncision()
  {
     //Realizar la incisión (¡ouch!).
   }
```

EJEMPLO

8



A es un ascendiente o **superclase** de B. Si la herencia entre A y B es directa decimos que A es la clase padre de B

B es un descendiente o **subclase** de A. Si la herencia entre A y B es directa decimos además que B es una clase hija de A

VENTAJAS

9

- Modelado de la realidad: las relaciones de especialización/generalización entre las entidades del mundo real.
- Evita redundancias
- Facilita la reutilización
- Sirve de soporte para el polimorfismo

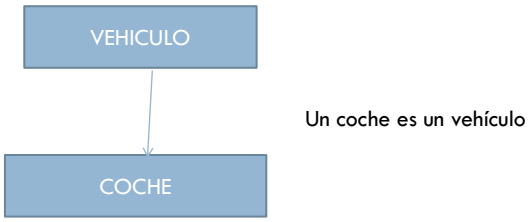
Situaciones en las que se aplica la herencia

10

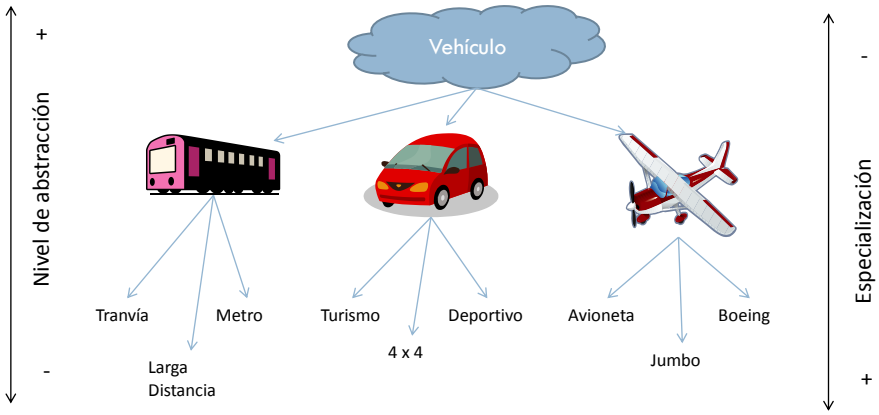
- Especialización
- Extensión
- Especificación
- Construcción ?????

Situaciones en las que se aplica la herencia

- Especialización: Dado un concepto B y otro concepto A que representa una especialización de B, entonces puede establecerse una relación de herencia entre las clases de objetos que representan a A y B.



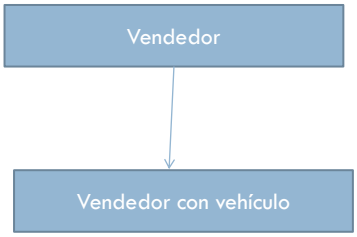
La herencia como especialización



Situaciones en las que se aplica la herencia

13

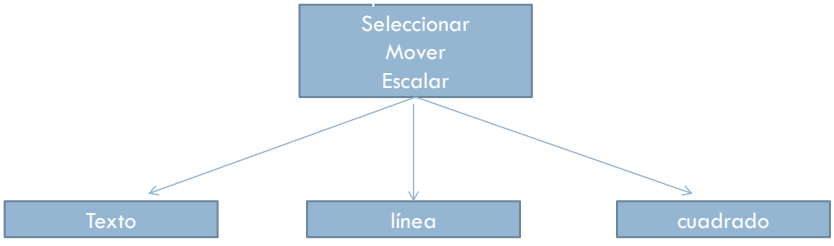
- Extensión: una clase puede servir para extender la funcionalidad de una superclase sin que represente necesariamente un concepto más específico.



Situaciones en las que se aplica la herencia

14

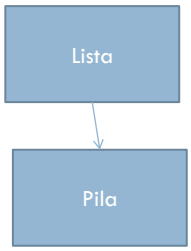
- Especificación: Una superclase puede servir para especificar la funcionalidad mínima común de un conjunto de descendientes.



Situaciones en las que se aplica la herencia

15

- Construcción: Una clase puede construirse a partir de otra, simplemente porque la hija puede aprovechar internamente parte o toda la funcionalidad del padre, aunque representen entidades sin conexión alguna.



Situaciones en las que se aplica la herencia

16

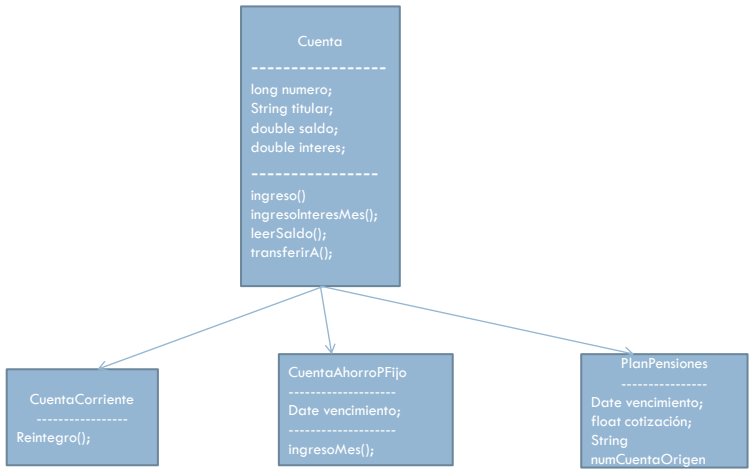
- Construcción: ¡¡MAL USO DE LA HERENCIA!!. No se debe emplear en estos casos. Si necesita la funcionalidad de una lista debe USAR una lista, no HEREDAR de una lista.
- No debemos usar la herencia cuando no se cumpla la regla **Es-un**

Ejemplo: Refresco es una bebida→Herencia

Bebida es un refresco ¿?→ NO

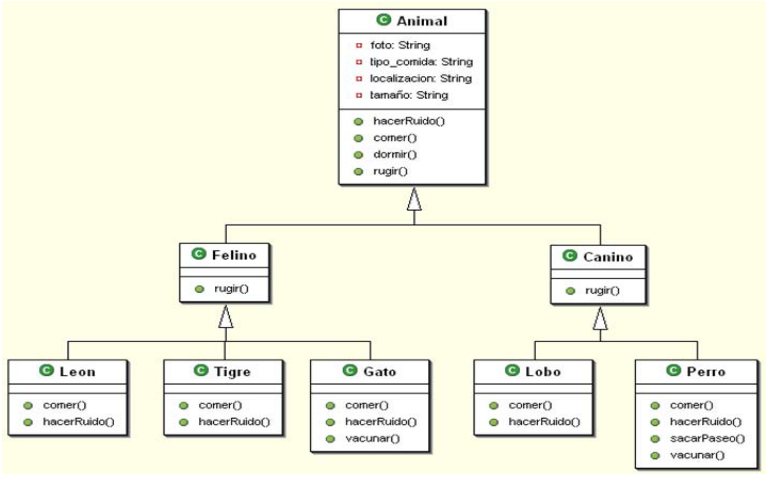
Situaciones en las que se aplica la herencia

17



Ejemplo

18



Reconocimiento Composición /Herencia

19

Herencia:

Clase hijo **<<es-un>>** Clasebase o padre

todos los objetos(miembros) de una clase hija (subconjunto) son objetos también de la clase padre (conjunto).

Composición :

Clase contenedora **<<tiene-un>>** Clase contenida

Tipos de Herencia

20

- Principalmente existen dos tipos de herencia.
  - **Herencia simple**: una clase solo puede tener un padre, por lo tanto la estructura de clases será en forma de árbol.
  - **Herencia múltiple**: Una clase puede tener uno o varios padres. La estructura de clases es un grafo. No soportada en Java

Tipos de Herencia

21

- Herencia simple:
  - Muy fácil de entender y manejar tanto por el lenguaje como por el programador.
  - Limitada puesto que en el mundo real un objeto puede pertenecer a varias clases y sin embargo aquí esta situación no se puede modelar

Herencia en Java

22

- Características
  - Herencia simple
  - Estructura jerárquica en árbol en donde en la raiz podemos encontrar la clase **Object,** de las que heredan todas las clases.
    - Todas las clases tienen un padre
    - Todos los objetos son "Object"
  - Java no permite que una subclase elimine un método o una variable de instancia aunque esta no la necesite
  - Una clase derivada (o subclase) puede acceder directamente a todos los atributos y métodos heredados que no sean ***private;*** a estos se accederá a través de métodos (get(), set())

Herencia en Java

23

- Se heredan todos los atributos y métodos, excepto los constructores.
- Una clase **final** no puede tener subclases.
- Si un método es static en la superclase, en la subclase también
- Si un método no es static en la superclase, en la subclase tampoco.

Consecuencias de la herencia

24

- Cuando una clase hereda de otra se dice que la clase de la que hereda es la ***clase padre*** o ***superclase*** y la clase que hereda es la ***clase hija*** o ***subclase***.
- La herencia permite ***reutilizar*** toda la definición de la superclase:
  - Se heredan los ***atributos*** y ***métodos***.
  - Los atributos forman parte de la subclase, aunque no los vea por la visibilidad privada.
  - Los métodos heredados están accesibles para la subclase como si fueran propios.
  - ***Los constructores no se heredan***, aunque es posible reutilizarlos.

La clase Object

25

- En Java todas las clases heredan de otra clase:
  - Si lo especificamos en el código con la *keyword extends,* nuestra clase heredará de la clase especificada.
  - Si no lo especificamos en el código, el compilador hace que nuestra clase herede de la clase Object (raíz de la jerarquía de clases en Java).
  - Ejemplo

```
public class MiClase extends Object
{
// Es redundante escribirlo puesto que el
// compilador lo hará por nosotros.
}
```

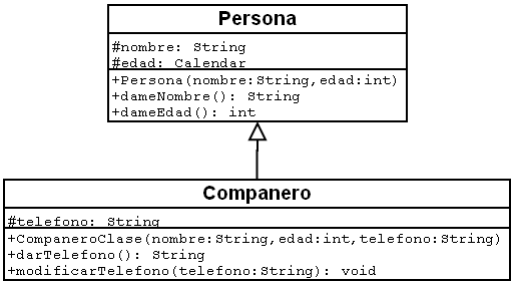
La clase Object

26

- Esto significa que nuestras clases siempre van a contar con los atributos y métodos de la clase Object
- Algunos de sus métodos
  - **public boolean equals(Object o);**
    Compara dos objetos y dice si son iguales.
  - **public String toString();**
    Devuelve la representación visual de un objeto.
  - **public Class getClass();**
    Devuelve la clase de la cual es instancia el objeto.

Herencia en Java

27



Herencia en Java

28

```
public class Persona{
    protected String nombre;
    protected int edad;

    public Persona(String nombre, int edad){
        this.nombre = nombre;
        this.edad = edad;
    }
    public String dameNombre(){
        return this.nombre;
    }
    public int dameEdad(){
        return this.edad;
    }
}
```

Herencia en Java

29

```
public class Companero extends Persona {
    protected String telefono;


    public Companero(String nombre, int edad, String telefono){
        this.nombre = nombre;
        this.edad = edad;
        this.telefono = telefono;
    }
    public String darTelefono(){
        return this.telefono;
    }
    public void modificarTelefono(String telefono){
        this.telefono = telefono;
    }
}
```

Herencia en Java

30

```
Companero compi= new Companero("Alberto",34,"953953953");

System.out.println(compi.darTelefono());
System.out.println(compi.dameNombre());
System.out.println(compi.dameEdad());
```

Castings

31

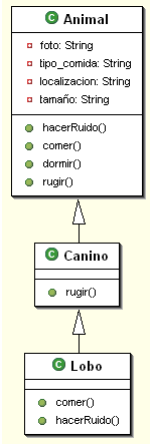
- El casting es una forma de realizar conversiones de tipos. Hay dos clases de casting:
  - **UpCasting**: conversión de un tipo en otro superior en la jerarquía de clases. No hace falta especificarlo.
  - **DownCasting**: conversión de un tipo en otro inferior en la jerarquía de clases.
- Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.

Ejemplo

32

```
public class Test Ejemplo
{
public static void main (String[] args)
{
      Lobo lobo = new Lobo();
      // UpCastings
      Canino canino = lobo;
      Object animal = new Lobo();
      animal.comer();
      // DownCastings
      lobo = (Lobo)animal;
      lobo.comer();
      Lobo otroLobo = (Lobo)canino;
      Lobo error = (Lobo) new Canino();
}
```

No compila. No puedes llamar al método comer() sobre un Object. No puedes convertir un Canino en un Lobo



Redefinición de métodos

33

- ¿Qué pasa si en la superclase hay un método que funciona distinto a como nos gustaría que funcionara en la subclase?
- ¿Son las subclases responsables de inicializar en sus constructores las variables heredadas de las superclases?
- ¿Qué pasa si un método de la superclase no debiera aparecer en la subclase?

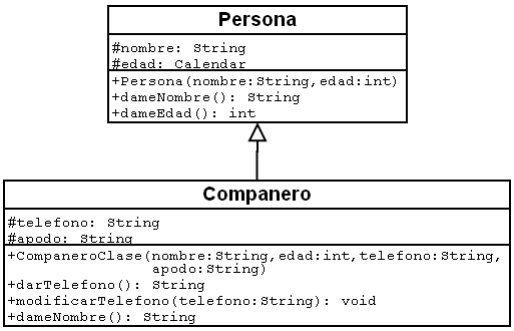
Redefinir o Sobreescribir un método

34

- Sobrescribir un método significa que una subclase reimplementa un método heredado.
- Para sobrescribir un método hay que respetar totalmente la declaración del método:
  - El nombre ha de ser el mismo.
  - Los parámetros y tipo de retorno han de ser los mismos.
  - El modificador de acceso no puede ser mas restrictivo.
- Al ejecutar un método, se busca su implementación de abajo hacia arriba en la jerarquía de clases.

Redefinición de métodos

35



Redefinición de métodos

36

```
public class Persona{
    protected String nombre;
    protected int edad;

    public Persona(String nombre, int edad){
        this.nombre = nombre;
        this.edad = edad;
    }
    public String dameNombre(){
        return this.nombre;
    }
    public int dameEdad(){
        return this.edad;
    }
}
```

Redefinición de métodos

37

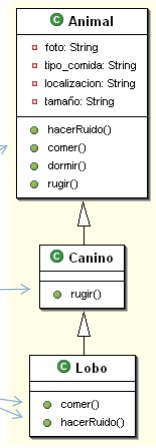
```
public class Companero extends Persona {
    protected String telefono;
    protected String apodo;


    public Companero(String nombre, int edad, String telefono, String apodo){
        this.nombre = nombre; this.edad = edad;
        this.telefono = telefono;
        this.apodo = apodo;
    }
    public String darTelefono(){
        return this.telefono;
    }
    public void modificarTelefono(String telefono){
        this.telefono = telefono;
    }
    public String dameNombre(){
        return this.apodo; // En la superclase era this.nombre
    }
}
```

Redefinición de métodos

38

```
public class Test
{
public static void main (String[] args)
{
    Lobo lobo = new Lobo();
    lobo.hacerRuido();
    lobo.rugir();
    lobo.comer();
    lobo.dormir();
}
```

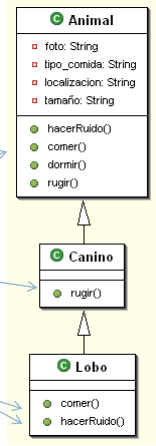


Redefinición de métodos

39

```
public class Test
{
public static void main (String[] args)
{
    Animal animal = new Lobo();
    animal.hacerRuido();
    animal.rugir();
    animal.comer();
    animal.dormir();
}
```

Los castings no modifican al objeto. Solo su tipo, por lo que se siguen ejecutando sobre el mismo objeto



Sobrescribir vs. Sobrecargar

40

- Sobrecargar un método es un concepto distinto a sobrescribir un método.
- La sobrecarga de un método significa tener varias implementaciones del mismo método con parámetros distintos:
  - Los parámetros tienen que ser distintos.
  - El modificador de acceso puede ser distinto.
  - El nombre ha de ser el mismo.
  - El tipo de retorno ha de ser el mismo

super y this

41

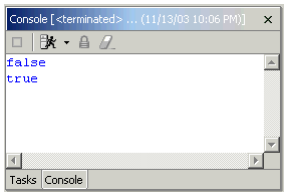
- **super** y **this** son dos *keywords de Java, son distintas*
- La palabra clave "***this***" es una ***referencia*** al objeto sobre el que se ejecuta el método, es decir, el ***objeto actual***.
- La palabra clave "***super***" es una ***facilidad del lenguaje*** para poder ejecutar constructores y métodos heredados que han sido redefinidos. **super** se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del padre.
- Cuando el atributo o método al que accedemos no ha sido sobrescrito en la subclase, el uso de super es redundante.
- Los constructores de las subclases incluyen una llamada a super() si no existe un super o un this.

super y this

42

Ejemplo de acceso a un atributo:

```
public class ClasePadre
{
   public boolean atributo = true;
}
public class ClaseHija extends ClasePadre
{
  public boolean atributo = false;
  public void imprimir()
 {
    System.out.println(atributo);
    System.out.println(super.atributo);
  }
}
```

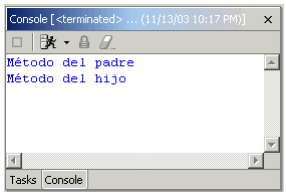


super y this

43

Ejemplo de acceso a un método:

```
public class ClasePadre
{
   public void imprimir()
  {
     System.out.println("Método del padre");
   }
}
public class ClaseHija extends ClasePadre
{
    public void imprimir()
   {
      super.imprimir();
      System.out.println("Método del hijo");
    }
}
```



Constructores

44

- Las subclases NO son responsables de inicializar las variables de instancia de las variables que hereda
- Para pasarle la responsabilidad de inicializar esas variables a las superclases puede llamar al constructor de estas mediante la sentencia **super()**
- Hay una especie de "herencia" de constructores

Constructores

45

```
public class Companero extends Persona {
    protected String telefono;


    public Companero(String nombre, int edad, String telefono){
        super(nombre,edad);
        this.telefono = telefono;
    }
    public String darTelefono(){
        return this.telefono;
    }
    public void modificarTelefono(String telefono){
        this.telefono = telefono;
    }
}
```

Métodos y clases abstractas

46

- Un método se dice que es abstracto si lo declaramos pero no lo implementamos.
- Utilidad:
  - podemos llamar a métodos que aún no hemos implementado o que se tienen que implementar en la subclases.
  - Podemos declarar métodos que tendrán todas las subclases pero que no se pueden implementar en la superclase
- Útil en polimorfismo

Operador instanceof

47

- Permite comprobar si un determinado objeto pertenece a una clase concreta. Se utiliza de esta forma:

  objeto* instanceof *clase

- Un ejemplo sería el siguiente:

```
String s = new String("No leas esto, sólo es un ejemplo");
if (s instanceof String)
System.out.prinln("Efectivamente s pertenece a la clase String");
```

La salida sería: *Efectivamente s pertenece a la clase String*

Operador instanceof

48

- Puede ocurrir que el objeto que estamos comprobando con *instanceof*, no sea una instancia directa de la clase que aparece a la derecha del operador, aun así *instanceof* devolverá *true* si el objeto es de un tipo compatible con el objeto que aparece a su derecha. Por ejemplo una instancia de una subclase

Operador instanceof

49

```
class Animal {}
class Perro extends Animal {
   public static void main (String[] args){
      Perro toby = new Perro();
      if (toby instanceof Animal)
   System.out.println("toby es un perro y también un
   animal");
    }
}
```

Métodos y clases abstractas

50

¿Qué pasa si tenemos tres clases que comparten un mismo mensaje (método) pero que su implementación es distinta en las tres?

Métodos y clases abstractas

51

Coche
#potencia: int #litros: int #diesel: boolean #gasolina: boolean
+moverse(posicionX:int,posicionY:int,velocidad:float): float

Helicoptero
#potencia: int #litros: int #diesel: boolean #gasolina: boolean #inclinacion: float
+moverse(posicionX:int,posicionY:int,velocidad:float): float

Lancha
#potenciaCadaMotor: int #litros: int #diesel: boolean #gasolina: boolean #numMotoresMax: int #numMotores: int
+moverse(posicionX:int,posicionY:int,velocidad:float): float

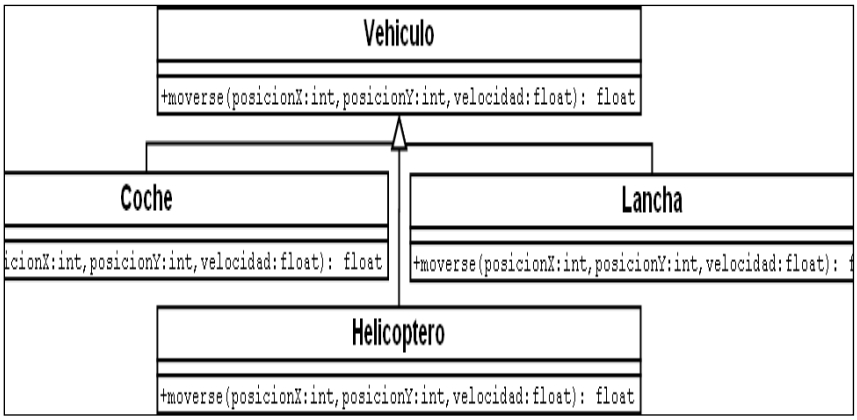
Métodos y clases abstractas

52

- Si creamos una superclase vehículo:
  - ¿Qué hacemos con el método moverse?
  - ¿Debe estar en la superclase? ¿es un método que tendrán todas subclases de vehículo?
  - ¿Qué implementación tendrá en la superclase?
  - ¿Me aporta alguna ventaja?

Métodos y clases abstractas

53



Métodos y clases abstractas

54

- Moverse en la superclase será un método abstracto:
  - Ventajas:
    - Las subclases tarde o temprano tienen que implementarlo.
    - En la superclases en donde este definido puede ser utilizado aunque no este implementado

```
public class Vehiculo{
    /* ... */

    Vehiculo(){
        /* ... */
    }

    /*...*/
    public abstract float moverse(int posX, int posY, float velocidad);

    public float mueveADestino(Destino destino){
        int i;
        Ruta ruta= new Ruta(destino);
        float recorrido=0;

        for(i=0;i<ruta.Length();i++){
            moverse(ruta[i].posicionX,ruta[i].posicionY);
        }

        return recorrido;
    }
}
```

55

Métodos y clases abstractas

56

- Una clase abstracta es una clase en la que al menos tiene un método que es abstracto.
  - Este(os) método(s) abstracto(s) puede que lo haya definido el o que los haya heredado.
  - Un clase abstracta NO puede instanciar objetos ¿Qué pasaría si se llamara a un método abstracto? ¿y si se llamara a un método no abstracto pero que usa un método abstracto?
- Para que una clase se declare como abstracta tenemos que usar la palabra clave **abstract**

```
public abstract class Vehiculo{
    /* ... */

    Vehiculo(){
        /* ... */
    }

    /*...*/
    public abstract float moverse(int posX, int posY, float velocidad);

    public float mueveADestino(Destino destino){
        int i;
        Ruta ruta= new Ruta(destino);
        float recorrido=0;

        for(i=0;i<ruta.Length();i++){
            moverse(ruta[i].posicionX,ruta[i].posicionY);
        }

        return recorrido;
    }
}
```

57

Métodos y clases abstractas

58

- IMPORTANTE: Para que un clase que hereda de una superclase abstracta pueda instanciar objetos debe de implementar todos los métodos abstractos que ha heredado, porque si no lo hiciera, también sería una clase abstracta
- **No es posible crear instancias** de una clase abstracta, pero si declarar entidades o variables de estas clases. Aunque la clase puede incluir la definición del constructor.

Clases abstractas

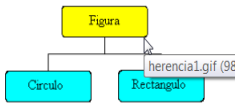
59

- Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta. Sin embargo, se pueden declarar variables de dichas clases.
- La definición de la clase abstracta *Figura, podría* contener la posición *x* e *y* de la figura particular, de su centro, y la función *area*, que se va a definir en las clases derivadas para calcular el área de cada figura en particular.

Clases abstractas

60

```
public abstract class Figura {
protected int x; protected int y;
public Figura(int x, int y) {
this.x=x; this.y=y; }
public abstract double area();
}
```



- Las clases derivadas heredan los miembros dato *x* e *y* de la clase base, y definen la función *area*, declarada **abstract** en la clase base *Figura*, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada *Rectangulo*, tiene como datos, aparte de su posición *(x, y)* en el plano, sus dimensiones, es decir, su anchura *ancho* y altura *alto*.

Enlace dinámico

61

- El lenguaje C++ y Java permiten decidir a que función llamar en tiempo de ejecución, esto se conoce como enlace tardío o dinámico.
- Podemos crear un array de la clase base *Figura* y guardar en sus elementos los valores devueltos por **new** al crear objetos de las clases derivadas

Enlace dinámico

62

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
```

Interfaces

63

- Definición informal:
  - Es una clase abstracta sin variables de instancia en donde todos sus métodos son abstractos
- Definición formal:
  - Es un contrato que establece una clase en el cual esta clase asegura que implementará un conjunto de métodos

Interfaces

64

```
public interface VehiculoCarrera {
    public float aceleracionDe0al100();
    public float usarNitro();
    public void usarMarchaEconomica();
    public void frenadaEmergencia();
}
```

Interfaces

- Para que una clase use un interfaz debe implementar todos sus métodos e indicar que está usando un interfaz.

```
public class Formula1 implements VehiculoCarrera{
    /* ... */
    public Formula1() {
        super();
    }
    public float aceleracionDe0a100(){
        /* ...*/
    }
    public float usarNitro(){
        /* ...*/
    }
    public void usarMarchaEconomica(){
        /* ... */
    }
    public void frenadaEmergencia(){
        /* ... */
    }
}
```

Interfaces

67

- Utilidad de los interfaces: “simular” herencia múltiple.
- Nota: los interfaces solo tienen sentido en lenguajes de programación que solo permiten usar la herencia simple

Modificadores de acceso

68

- Son los que permiten limitar o generalizar el acceso a los componentes de una clase (ya sea dato o método) o a la clase en si misma
- Los modificadores de acceso preceden a la declaración de un elemento de la clase:

[modificadores] tipo_variable nombre;

[modificadores] tipo_devuelto nombreMetodo (lista_Argumentos);

CONTROL DE ACCESO

69

- **public** => Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instancia públicos
- **protected** => Sólo las subclases de la clase y nadie más puede acceder a las variables y métodos de instancia protegidos.
- **private** => Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases
- **sin modificador** - Se puede acceder al elemento desde cualquier clase del package donde se define la clase.

Modificadores de acceso para clases

70

Las clases en si mismas pueden declararse:

- **public** - Todo el mundo puede usar la clase.
- **sin modificador** - La clase puede ser usada e instanciada por clases dentro del package donde se define.
- Las clases no pueden declararse ni **protected** , ni **private**

La cláusula `final`

71

- Para una clase, `final` significa que la clase no puede extenderse
- Para un método, `final` significa que no puede redefinirse en una clase derivada
- Para un dato miembro, `final` significa también que no puede ser redefinido en una clase derivada, como para los métodos, pero además significa que su valor no puede ser cambiado en ningún sitio; es decir el modificador `final` sirve también para definir valores constantes.

POLIMORFISMO

72

- [RAE 2001]: Cualidad de lo que tiene o puede tener distintas formas
- El polimorfismo en POO se da por el uso de la herencia
- Se produce por distintas implementaciones de los métodos definidos en la clase padre (**sobre escribir**):
  - Distinta implementación entre clase hija y padre
  - Distinta implementación entre clases hija
- Una misma llamada ejecuta distintas sentencias dependiendo de la clase a la que pertenezca el objeto
- El código a ejecutar se determina en tiempo de ejecución => **Enlace dinámico**

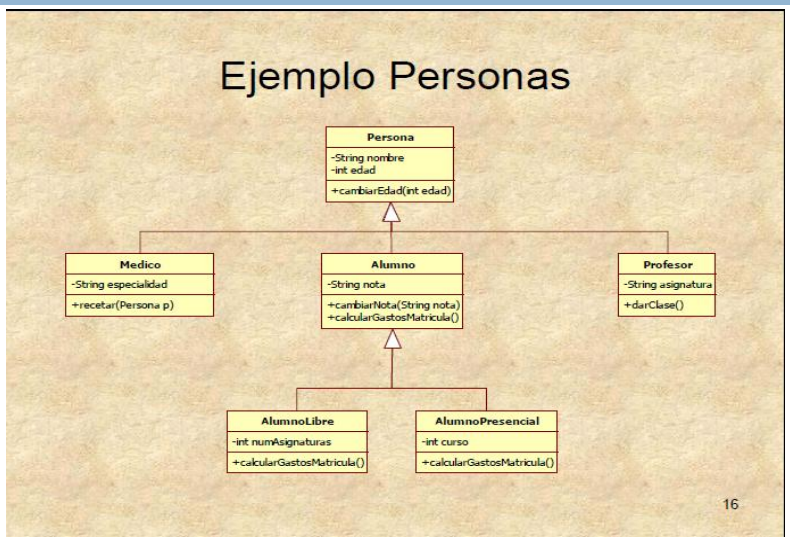
POLIMORFISMO

73

- Supongamos que declaramos: *Persona p;*
- Podría referenciar a un profesor o a un alumno en distintos momentos
- 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 implementación del método se ejecuta.
- **OJO!: la Sobrecarga de funciones no es lo mismo,** utiliza enlace estático, por tanto se decide en **tiempo de compilación.**

EJEMPLO

74



Ejemplo Polimorfismo

75

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