Tema 3
Herencia

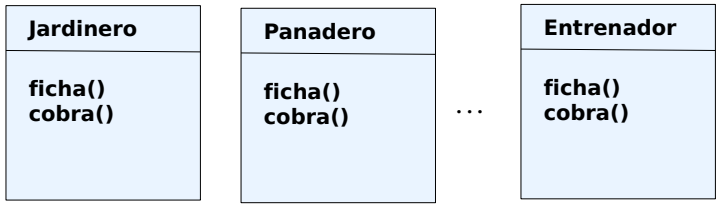
Programación II

Alicia Garrido Alenda

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

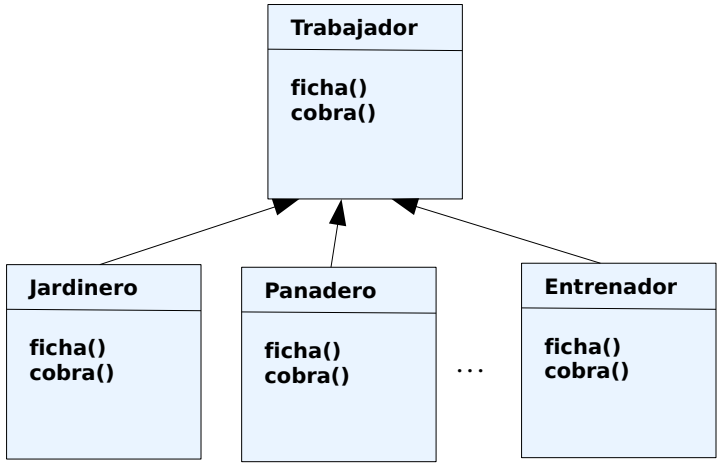
Introducción

- Supongamos que tenemos:



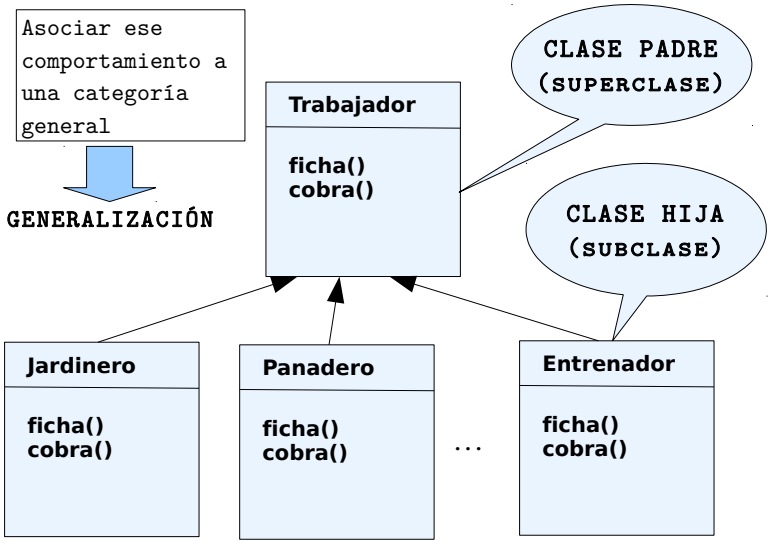
Introducción

- Todos tienen algo en común.



Introducción

- Los hijos heredan características de los padres.



Clasificación y generalización

- La generalización genera jerarquías de generalización/especialización.
- La implementación de estas jerarquías en un lenguaje de programación produce jerarquías de herencia.
- La herencia permite clasificar los tipos de datos por variedad ⇒ aproximación al mundo real.
- Ahora podemos clasificar los conceptos de acuerdo a:
  - Pertenencia (TIENE-UN) → Agregación/Composición
  - Variedad (ES-UN) → Herencia

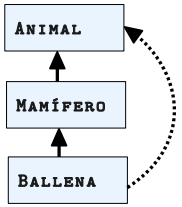
¿Qué es la herencia?

HERENCIA

Es el mecanismo de implementación por el cual elementos más específicos incorporan la estructura y comportamiento de elementos más generales[a].

[a] Rumbaugh 99

- Con la herencia se puede especializar o extender la funcionalidad de una clase, creando nuevas clases a partir de ella.
- La herencia es transitiva:



Tipos de herencia

- Según número de superclases:
  - Simple: única superclase.
  - Múltiple: más de una superclase.
- Según implementación:
  - Implementación: se hereda la implementación de los métodos.
  - Interfaz: sólo se hereda la interfaz.

Herencia de implementación simple

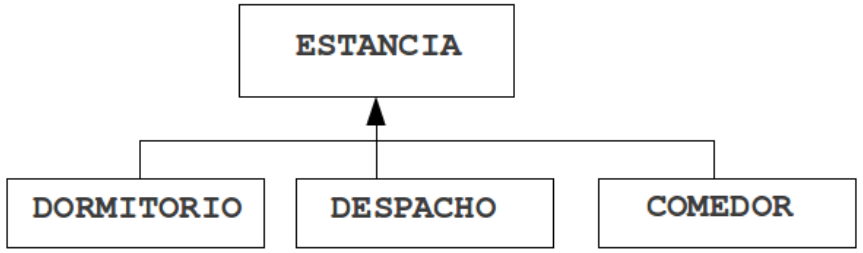
- Una subclase hereda **todas** las variables y métodos de instancia de la superclase.
- La implementación de los métodos es heredada pero puede sobreescribirse en las subclases.
- En Java:
  - La herencia de implementación sólo puede ser *simple*.
  - Existe una superclase común a todas las clases que se denomina `Object` (raíz única de la estructura arbórea de clases).

Herencia de implementación en Java

- **`[public|private][abstract|final]`** class NombreClase **`extends`** NombreSuperClase
- `public, private`: Modificador opcional que indica la visibilidad de la clase.
- `abstract`: Modificador que indica que la clase tiene algún método sin especificar y por tanto no se pueden crear instancias de esta clase.
- `final`: Modificador opcional que indica que la clase es *final*, es decir, que no se va a heredar de ella.
- `extends`: Palabra reservada que indica que esta clase es una subclase de otra.

Herencia simple en Java

- Ejemplo:



Herencia simple en Java

```
package principal;
public class Estancia{
 private String color;
 private boolean lampara;
 public Estancia(String c){
  color=c;
 }
 public void pinta(String nc){ color=nc; }
 public void enciende(){lampara=true;}
 public void apaga(){lampara=false;}
 public boolean getLampara(){return lampara;}
}
```

```
package secundario;
import java.util.*;
import principal.*;
public class Dormitorio extends Estancia{
 private ArrayList<Boolean> ventanas;
 private boolean camahecha;
 public Dormitorio(String c,int numv){...}
 public boolean airea(int i){...}
 ...
}
```


Herencia simple en Java

```
package secundario;
import java.util.*;
import principal.*;
public class Despacho extends Estancia{
 private ArrayList<Estanterias> estantes;
 public Despacho(String c,int maxestantes){...}
 public boolean agregaLibro(Libro ejemplar){...}
 ...
}
```

```
package secundario;
import java.util.*;
import principal.*;
public class Comedor extends Estancia{
 private boolean televisor;
 private ArrayList<String> mesa;
 public Comedor(String c,int maxnume){...}
 public int poneMesa(String[] elementos){...}
 ...
}
```

Herencia y derechos de acceso

- Los miembros definidos como públicos en la superclase o en la subclase serán accesibles desde el exterior.
- Los miembros definidos como privados serán inaccesibles:
  - Una subclase no puede acceder a los miembros privados de su superclase.
  - Una subclase puede acceder a cualquier miembro público de su superclase como si fuera propio (no se necesita ninguna variable).
  - Modificador de acceso **protected**: una subclase puede acceder a cualquier miembro protegido de su superclase como si fuera propio.

Herencia y derechos de acceso

```
package principal;
public class Estancia{
 private String color;
 private boolean lampara;
 ...
}
```

```
package secundario;
import java.util.*;
import principal.*;
public class Dormitorio extends Estancia{
 private ArrayList<Boolean> ventanas;
 private boolean camahecha;
 ...
 // ERROR: lampara es privada de Estancia; es necesario
 // invocar a getLampara() o usar el modificador protected
 public void onOffLuz(){
  if (lampara)
    apaga();
  else
    enciende();
 }
}
```


Herencia y derechos de acceso

```
package principal;
public class Estancia{
 private String color;
 protected boolean lampara;
 ...
}
```

```
package secundario;
import java.util.*;
import principal.*;
public class Dormitorio extends Estancia{
 private ArrayList<Boolean> ventanas;
 private boolean camahecha;
 ...
 public void onOffLuz(){
  if (lampara) //OK: protegida de Estancia => accesible
    apaga();
  else
    enciende();
 }
}
```


Herencia y derechos de acceso

```
package aplicacion;
import principal.*;
import secundario.*;
public class Aplicacion{
 public static void main(String[] args){
  Dormitorio mio=new Dormitorio("verde",2);
  ...
  if(mio.lampara){  // ERROR: lampara no es accesible
    ...
  }
  if(mio.getLampara()){  //CORRECTO
  }
 }
}
```

Herencia y métodos constructores

- Cuando creamos un objeto de una subclase, pasamos una serie de parámetros para inicializar sus variables de instancia.
- Algunos de estos parámetros contienen valores para las variables de instancia definidas en la superclase.
- Para ello realizamos una llamada al constructor de la superclase con la palabra reservada **super**.
- Un constructor de una subclase siempre debe invocar en su primera sentencia al constructor de la superclase.

Los constructores no se heredan

Herencia y métodos constructores

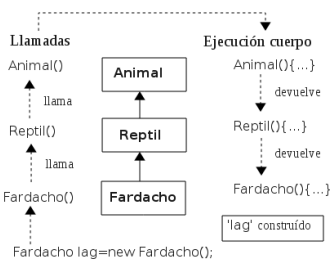
```
package principal;
public class Estancia{
 private String color;
 private boolean lampara;
 public Estancia(String c){  color=c;  }
 ...
}
```

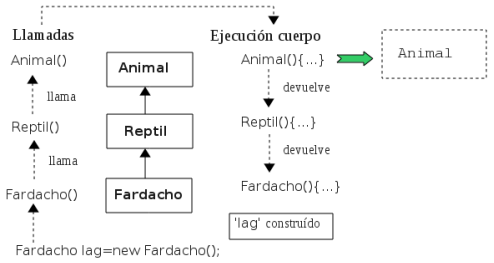
```
package secundario;
import java.util.*;
import principal.*;
public class Dormitorio extends Estancia{
 private ArrayList<Boolean> ventanas;
 private boolean camahecha;
 public Dormitorio(String c,int numv)
 {
   super(c);
   ventanas=new ArrayList<Boolean>();
   numv=numv<0?0:numv;
   for(int i=0;i<numv;i++) ventanas.add(false);
 }
 ...
}
```


Herencia y métodos constructores

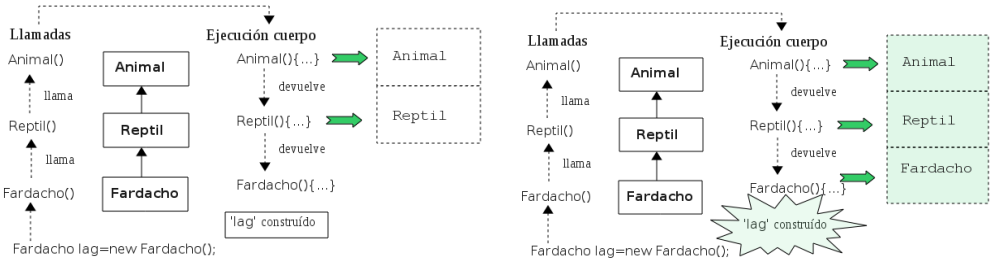
- Los constructores siempre deben ser definidos para las subclases.
- Creación de un objeto de una subclase: se invoca a todos los constructores de su jerarquía de clases.
- Orden de ejecución de los constructores: primero se **ejecuta** el constructor de la superclase y después el de la subclase.

Herencia y métodos constructores





Herencia y métodos constructores



Herencia: ejemplo de superclase

Ciudadano

- **- nombre :** `String`
- **# vida :** `int`
- **- bolsa :** `double`
- **# muerto :** `boolean`
- **- poblacion :** `int`
- **- poblacionViva :** `int`

- **+ Ciudadano(String, int, double)**
- **+ consulta( )**
- **+ muere( )**
- **# daVida(String) :** `int`
- **# quitaVida(String) :** `int`
- **+ alimenta(String, String) :** `int`
- **+ paga(double) :** `boolean`
- **+ getPoblacion( ) :** `int`
- **+ getPoblacionViva( ) :** `int`

Herencia simple (superclase): Ciudadano (I)

```
public class Ciudadano{
 private String nombre;
 protected int vida;
 private double bolsa;
 protected boolean muerto;
 private static int poblacion=0, poblacionViva=0;

 public Ciudadano(String n, int v, double s){
  nombre=n;
  vida=v;
  bolsa=s;
  if(vida>0){
    muerto=false;
    poblacionViva++;
  }
  else
    muerto=true;
  poblacion++;
 }
```


Herencia simple (superclase): Ciudadano (II)

```
public static int getPoblacion(){
  return poblacion;
}

public static int getPoblacionViva(){
  return poblacionViva;
}

public void consulta(){
 System.out.print("Nombre: "+nombre+"; ");
 System.out.print("Vida: "+vida+"; ");
 System.out.println("Bolsa: "+bolsa+"; ");
}

public void muere(){
 if((vida<=0)&&(!muerto)){
   poblacionViva--;
   muerto=true;
 }
}
```

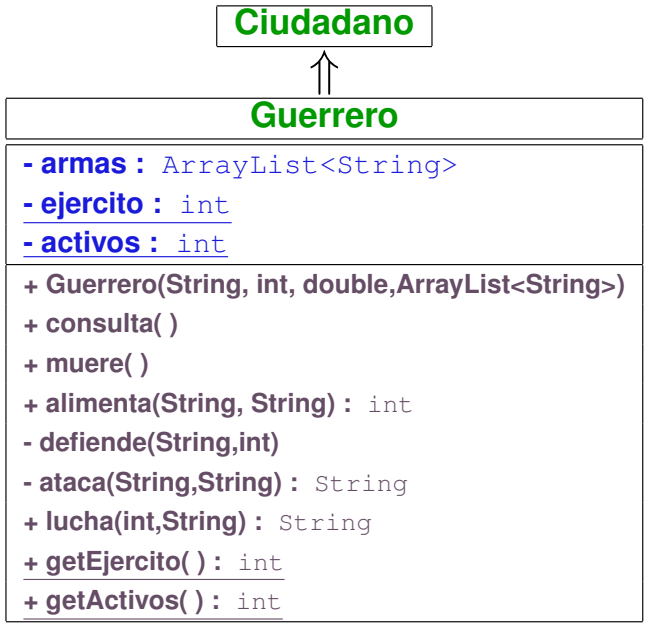
Herencia simple (superclase): Ciudadano (III)

```
protected int daVida(String comida){
  if (comida.equalsIgnoreCase("ambrosia"))
      vida+=2;
  else
     vida+=1;
  return 1;
}
protected int quitaVida(String comida){
  int vivo=1;
  if (comida.equalsIgnoreCase("setas"))
     vida-=2;
  else
     vida-=1;
  if (vida<=0){
   muere();
   vivo=0;
  }
  return vivo;
}
```


Herencia simple (superclase): Ciudadano (IV)

```
public int alimenta(String alimento,String lugar){
  int alimentado=-1;
  if(!muerto&&(alimento!=null)&&(lugar!=null)){
   if (lugar.equalsIgnoreCase("mazmorra"))
     alimentado=quitaVida(alimento);
  else
     alimentado=daVida(alimento);
  }
  return alimentado;
}
 public boolean paga(double importe){
  boolean pagado=false;
  if(!muerto){
    if (importe<=bolsa){
     bolsa-=importe;
     pagado=true;
    }
  }
  return pagado;
 }
} // fin clase Ciudadano
```


Herencia: ejemplo de subclase



Herencia simple (subclase): Guerrero (I)

```
import java.util.*;
public class Guerrero extends Ciudadano{
 private ArrayList<String> armas;
 private static int ejercito=0,activos=0;

 public Guerrero(String n, int v, double s,
                 ArrayList<String> a){
  super(n,v,s);
  armas=new ArrayList<String>();
  if(a!=null){
   for (int i=0;i<a.size();i++)
      armas.add(a.get(i));
  }
  if(!muerto)
    activos++;
  ejercito++;
 }
```

¿Debemos incrementar poblacion y poblacionViva?

Herencia simple (subclase): Guerrero (II)

```
public static int getEjercito(){ return ejercito; }
public static int getActivos() {  return activos;  }
public void consulta(){
 super.consulta();  //---------------> REFINAMIENTO
 System.out.println("Arsenal:");
 for (int i=0;i<armas.size();i++)
   System.out.println("      Arma "+i+": "+armas.get(i));
}
public void muere(){
 if(!muerto){
  super.muere();     //---------------> REFINAMIENTO
  if(muerto)
   activos--;
 }
}
```

- Método con la misma signatura que la superclase que invoca el correspondiente método

Refinamiento

Herencia simple (subclase): Guerrero (III)

```
public int alimenta(String alimento,String lugar){
 int alimentado=-1;
 if(!muerto&&(alimento!=null)&&(lugar!=null)){
  if ((lugar.equalsIgnoreCase("campo batalla"))&&
      (!(alimento.equalsIgnoreCase("ambrosia"))))
    alimentado=quitaVida(alimento);
  else
    alimentado=daVida(alimento);
 }
 return alimentado;
}
```

- Método con la misma signatura que la superclase pero distinta implementación

Reemplazo

Herencia simple (subclase): Guerrero (IV)

```
private void defiende(String atak,int parapeto){
 if(atak!=null){
  if ((atak.equalsIgnoreCase("azote"))||
     (atak.equalsIgnoreCase("piedra voladora"))||
     ...)
   vida-=1;
 else
  if ((atak.equalsIgnoreCase("saeta"))||...)
    vida-=2;
}
if (parapeto>0) vida++;
}
```

- Método completamente nuevo, amplia el comportamiento de la superclase

Extensión

Herencia simple (subclase): Guerrero (V)

```
private String ataca(String atacar,String atak){
 String ataque="";
 defiende(atak,0);
 if(atacar!=null){
  if ((atacar.equalsIgnoreCase("maza")))
    ataque=new String("azote");
  else if ((atacar.equalsIgnoreCase("honda"))
    ataque=new String("piedra voladora");
  else if ((atacar.equalsIgnoreCase("boleadoras"))
    ataque=new String("atadura de pies");
  else if ((atacar.equalsIgnoreCase("arco"))
    ataque=new String("saetada");
  ...
 }
 return ataque;
}
```

Herencia simple (subclase): Guerrero (VI)

```
public String lucha(int minimo,String recibe){
 int escoger=0;
 String contrataque="";
 if(!muerto){
  if (activos>minimo){
    escoger=armas.size()/(ejercito-minimo);
   if((escoger>=0)&&(escoger<armas.size()))
    contrataque=ataca(armas.get(escoger),recibe);
   else defiende(recibe,1);
  }
  else defiende(recibe,2);
 }
 return contrataque;
}
} //Fin clase Guerrero
```


Conversión de tipos entre clases

Principio de sustitución *(Liskov,1987)*

Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado.

- **Subtipo**: Una clase **B**, subclase de **A**, es un subtipo de **A** si podemos sustituir instancias de **A** por instancias de **B** en cualquier situación y sin ningún efecto observable.
- Todos los LOO soportan subtipos:
  - Lenguajes fuertemente tipados: caracterizan los objetos por su clase.
  - Lenguaje débilmente tipados: caracterizan los objetos por su comportamiento.

Conversión de tipos entre clases en Java

- A una variable de una superclase se le puede asignar un objeto de cualquier subclase derivada de ella.
- Sólo se pueden realizar conversiones de tipo entre clases relacionadas por herencia.
- El siguiente ejemplo daría un error:

```
public class Piedra{...}
public class Papel{...}
public class Conversor{
 public static void main(String[] args){
  Piedra peque = new Piedra();
  Papel hoja;

  hoja = (Papel) peque;
 }
}
```

- ¿Cómo hacemos la conversión de tipo entre clases relacionadas por herencia?

Conversión de tipo: *upcasting/downcasting*

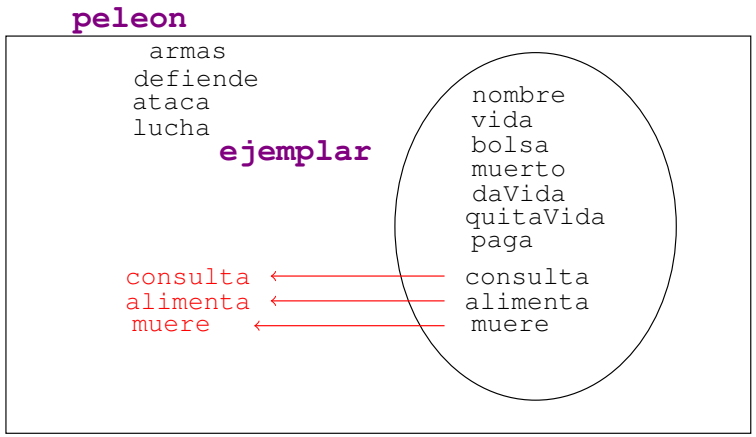
- *Upcasting*: técnica que permite “convertir” un objeto de una subclase en un objeto de su superclase.

```
Ciudadano ejemplar;
Guerrero peleon;

peleon=new Guerrero(...);
ejemplar=peleon;
atq=ejemplar.lucha(4,"azote");  //error
ejemplar.consulta(); //OK: info del guerrero
```

Conversión de tipo (herencia): *upcasting*

- El objeto **`ejemplar`**:
  - Utiliza la interfaz de la clase `Ciudadano` pero accede a los métodos de la clase `Guerrero`.
  - No puede acceder a la interfaz de la clase `Guerrero`.



Conversión de tipo (herencia): *downcasting*

- ¿Cómo podemos acceder a la interfaz de la clase `Guerrero`?

```
Ciudadano primus;
primus=new Guerrero(...);
```

- *Downcasting*: técnica que permite "convertir" un objeto de una superclase en un objeto de una de sus subclases.

```
Guerrero converso;
converso=(Guerrero)primus;
```


Enlace dinámico: selección dinámica del método

- Una variable de tipo `Ciudadano` puede contener un objeto de tipo `Ciudadano`:

```
Ciudadano primus;
primus=new Ciudadano(...);
```

- O bien un objeto de tipo `Guerrero`:

```
Ciudadano primus;
primus=new Guerrero(...);
```

- En función de esto, cuando hagamos `primus.consulta()` mostrará una información u otra.

Enlace dinámico: tiempo de enlace

Tiempo de enlace:

Momento en el que se identifica el fragmento de código a ejecutar asociado a un mensaje (llamada a método) o el objeto concreto asociado a una variable.

- Enlace estático: en tiempo de compilación.

  Ventaja: mayor eficiencia

- Enlace dinámico: en tiempo de ejecución.

  Ventaja: mayor flexibilidad

Tiempo de enlace: objetos

- Enlace estático: el tipo de objeto que contiene una variable se determina en tiempo de compilación.
  - Por ejemplo, en C++:
    `Ciudadano primus(...);`
    crea un objeto de tipo `Ciudadano` en la variable `primus`.
  - En Java no existe.
- Enlace dinámico: el tipo de objeto al que hace referencia un variable no está predefinido, de manera que el sistema gestionará la variable en función del tipo real del objeto que referencie la variable durante la ejecución.
  - C++: cuando las variables son punteros o referencias y sólo dentro de jerarquías de herencia. Por ejemplo,
    `Ciudadano *primus=new Guerrero(...);`
  - Java: sólo dentro de jerarquías de herencia. Por ejemplo,
    `Ciudadano primus=new Guerrero(...);`

Tiempo de enlace: métodos

- Enlace estático: la elección del método encargado de responder a un mensaje se hace en tiempo de compilación, en función del tipo que tenía el objeto destino de la llamada en tiempo de compilación.
  - Por ejemplo, en C++:
    ```
    Guerrero converso(...);
    Ciudadano primus;
    primus=converso;
    primus.consulta(); // método de Ciudadano
    ```
  - En Java no existe.
- Enlace dinámico: la elección del método encargado de responder a un mensaje se hace en tiempo de ejecución, en función del tipo del objeto que referencia la variable mediante la que se invoca al método en el instante de la ejecución del mensaje.
  ```
  Ciudadano primus;
  primus=new Guerrero(...);
  primus.consulta(); // método de Guerrero
  ```


Ventajas del enlace dinámico

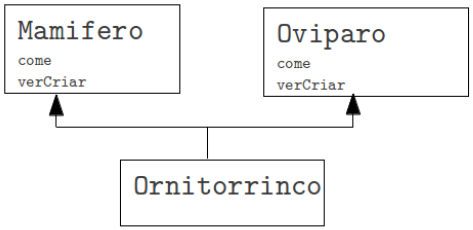
- El reemplazo o refinamiento (sobreescritura) de métodos permite que una clase general especifique métodos que serán comunes a todas sus subclases, permitiendo a éstas definir la implementación específica de alguno de estos métodos.
- Este es uno de los mecanismos más poderosos que ofrece la POO para soportar:
  - La reutilización de código.
  - La robustez.

Herencia de implementación múltiple

- Una subclase hereda todas las variables y métodos de sus **superclases** (más de una).
- Permite modelar problemas en los que un objeto tiene propiedades según criterios diferentes.
- Las estructuras de herencia múltiple son más flexibles, pero más difíciles de manejar.
- Presenta problemas de implementación.
- En Java no está permitida.

Problemas de la herencia múltiple (I)

- Colisión de nombres: ambigüedad en los nombres



- ¿Con cuál me quedo?

Problemas de la herencia múltiple (II)

- Renombrar:

```
class Mamifero{
 protected:
   int tpo_gestacion;
   int tpo_amamanta;
 public:
   virtual void come(char *);
   void verCriar();
}
//fichero Mamifero.h
```

```
class Oviparo{
 protected:
   int tpo_incubacion;
   int tam_huevos;
 public:
   virtual void come(char *);
   void verCriar();
}
//fichero Oviparo.h
```

```
class Ornitorrinco:public Mamifero,public Oviparo{
 public:
   void come(char *);
   void verCriarM(){Mamifero::verCriar();}
   void verCriarO(){Oviparo::verCriar();}
}
//fichero Ornitorrinco.h
```


Problemas de la herencia múltiple (III)

- Redefinir:

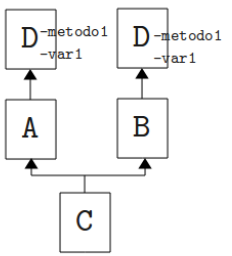
```
class Mamifero{
 protected:
   int tpo_gestacion;
   int tpo_amamanta;
 public:
   virtual void come(char *);
   void verCriar();
}
//fichero Mamifero.h
```

```
class Oviparo{
 protected:
   int tpo_incubacion;
   int tam_huevos;
 public:
   virtual void come(char *);
   void verCriar();
}
//fichero Oviparo.h
```

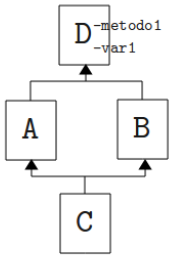
```
void Ornitorrinco::come(char * c){
  //come(c); llamada recursiva
  Mamifero::come(c); //deshace ambigüedad
}
//fichero Ornitorrinco.cc
```


Problemas de la herencia múltiple (IV)

- Herencia de superclases comunes: número de copias de variables (C++).




```
class A: public D
class B: public D
class C: public A,public B
```




```
class A: virtual public D
class B: virtual public D
class C: public A,public B
```

Problemas de la herencia múltiple (V)

- Herencia de superclases comunes: ejecución de métodos.
  - ¿Qué camino se sigue para ejecutar un método que se ha redefinido en varias subclases?
    - ★ No se permiten superclases comunes (Eiffel).
    - ★ Se prioriza un camino (Lisp OO).
    - ★ Se redefinen los métodos de cada camino como en la colisión de nombres (C++).

Aplicaciones de la herencia

- Las aplicaciones más habituales de la herencia son:
  - Especialización
  - Especificación
  - Extensión
  - Variación
  - Combinación

Aplicaciones de la herencia: especialización

- Es el concepto de herencia más común:
  - Si B **es un** A tiene sentido pero hace algo de forma diferente ⇒ es muy probable que B sea una subclase de A.
- Es necesario redefinir como mínimo un método.
- El uso de herencia para especializar garantiza:
  - Todo objeto de una subclase también es un objeto de la superclase.
  - Todas las propiedades de la superclase se conservan en la subclase.

Aplicaciones de la herencia: especificación

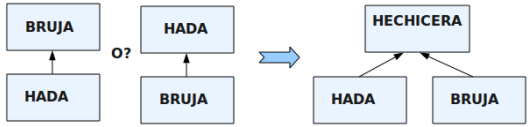
- La superclase no tiene el comportamiento definido y son las subclases las que lo especifican:
  - Se usa la herencia para garantizar que un conjunto de clases tiene un protocolo común (una interfaz común).
  - La superclase puede ser una combinación de operaciones implementadas y de operaciones que se delegan a las subclases.
  - En Java podemos usar interfaces.
  - Por ejemplo disponemos de las clases `Collection` y `Object`.

Aplicaciones de la herencia: extensión

- Agrega capacidades totalmente nuevas.
- Se amplía la subclase añadiendo métodos y/o atributos.
- No es necesario redefinir ningún método.
- Normalmente se usa junto con la especialización.

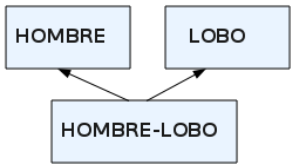
Aplicaciones de la herencia: variación

- A veces se utiliza para reutilizar código.
- Dos o más clases tienen implementaciones similares pero no hay relación jerárquica.
- Consiste en agrupar código común de dos o más clases sin relación de herencia (*es un*).



Aplicaciones de la herencia: combinación

- También conocida como herencia múltiple.
- Combina en una subclase características de dos o más superclases.



Clase abstracta vs. interfaz

Clase abstracta

- Tiene como mínimo un método *abstracto* (sin implementación).
- Puede tener variables de instancia y algunos métodos implementados.
- No se puede crear un objeto de una clase abstracta.
- Una subclase de una clase abstracta debe **sobreescribir** una implementación para cada método abstracto de su superclase, de lo contrario también será una clase abstracta.
- Una clase abstracta solo puede heredar de una clase (herencia simple).

Interfaz

- Tiene todos sus métodos *abstractos* (sin implementación).
- No tiene ningún constructor.
- No puede tener variables de instancia.
- Una interfaz puede heredar de varias interfaces.

Herencia de interfaz

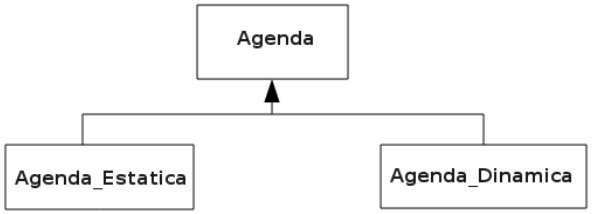
- Una subclase hereda **sólo** la interfaz.
- No se hereda código.
- Objetivos:
  - Reutilización de conceptos: no se comparte código, se comparte el prototipo de método.
  - Garantizar el principio de sustitución.
- En Java:
  - Una interfaz puede heredar de varias interfaces.
  - Una clase puede implementar varias interfaces.
  - Una clase puede heredar de otra clase e implementar una interfaz.

Herencia de interfaz en Java

- **`[public][abstract|final]`** class NombreClase **`implements`** NombreInterfaz
- `public`: Modificador opcional que indica la visibilidad de la clase.
- `abstract`: Modificador opcional que indica que la clase tiene algún método sin especificar y por tanto no se pueden crear instancias de esta clase.
- `final`: Modificador opcional que indica que la clase es *final*, es decir, que no se va a heredar de ella.
- `implements`: Palabra reservada que indica que esta clase es una implementación de una interfaz.

Herencia de interfaz en Java

- Ejemplo:




```
public interface Agenda{
 boolean agrega(Tarea una);
 void elimina(Tarea una);
 void ordenaPorFecha();
 void consulta();
}
```


Herencia de interfaz en Java

```
public class Agenda_Estatica implements Agenda{
 private Tarea[] lista;
 ...
 public Agenda_Estatica(int capacidad){
  capacidad=capacidad<=0?capacidad=10:capacidad;
  lista=new Tarea[capacidad];
 }
 public boolean agrega(Tarea una){
  boolean intro=false;
  if(una!=null){
   for (int i=0;( (i<lista.length)&&(!(intro)) );i++)
    if (lista[i]==null){
     lista[i]=una;
     intro=true;
    }
   }
  return intro;
 }
 ....
}
```

Herencia de interfaz en Java

```
public class Agenda_Dinamica implements Agenda{
 private ArrayList<Tarea> lista;
 ...
 public Agenda_Dinamica(){
  lista=new ArrayList<Tarea>();
 }
 public boolean agrega(Tarea una){
  boolean intro;
  intro=lista.add(una);
  return intro;
 }
 ....
}
```


Herencia de interfaz en Java

```
public class Principal{
 public static void main(String[] args){
  Agenda mia=new Agenda_Estatica(50); //Agenda_Dinamica();
   ...
  boolean hecho=mia.agrega(unatarea);
   ...
  mia.consulta();
 }
}
```

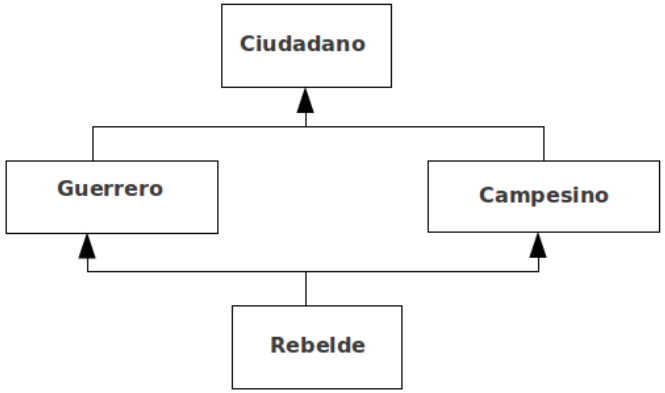
- Podemos modificar toda la aplicación para que use una implementación u otra cambiando `Agenda_Estatica` por `Agenda_Dinamica` en un único lugar.

Simulación de herencia múltiple con interfaces

- Se permite la herencia múltiple entre interfaces:
  - **[public]** interface NombreInterface **extends** Interface1,Interface2,...
- Una clase hereda de otra clase e implementa una interfaz:
  - **[public]** class NombreClase **extends** NomSuperClase **implements** Interface1,...
- Una clase implementa varias interfaces:
  - **[public]** class NombreClase **implements** Interface1, Interface2 ...

Simulación de herencia múltiple con interfaces

- Siguiendo con nuestro ejemplo de herencia:



- Definimos la interfaz correspondiente a la clase `Campesino` y la implementamos en la clase `Rebelde`.

Herencia múltiple: especificación de interfaces

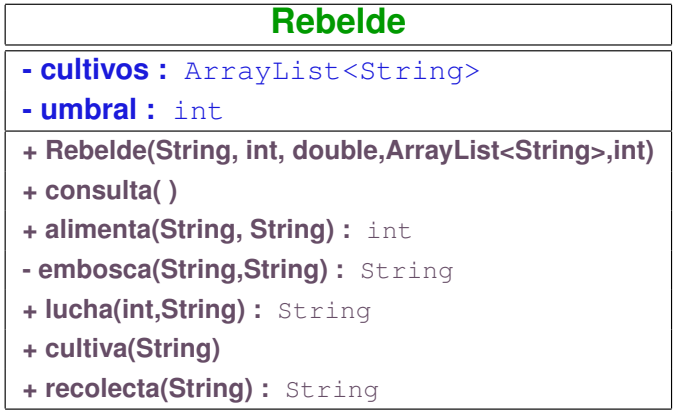
```
package Interfaz;
public interface Ciudadano{
 void consulta();
 void muere();
 int alimenta(String alimento,String lugar);
 boolean paga(double importe);
}
```

```
package Interfaz;
public interface Campesino extends Ciudadano{
 void cultiva(String cultivo);
 String recolecta(String cosecha);
}
```


Herencia múltiple: especificación de subclase

- Reutilización de código: extender la clase `Guerrero` e implementar la interfaz `Campesino`.
- No es necesario volver a implementar los métodos de la superclase extendida.
- Es necesario implementar los métodos heredados de la interfaz para que no sea una clase abstracta.

Herencia múltiple: Rebelde (I)



Herencia múltiple: Rebelde (I)

```
package Clases;
import java.util.*;
import Interfaz.*;
public class Rebelde extends Guerrero
                     implements Campesino{
 private ArrayList<String> cultivos;
 private int umbral;

 public Rebelde(String n,int v,double s,
        ArrayList<String> a,int nc){
   super(n,v,s,a);
   nc=nc>=0?nc:0;
   cultivos=new ArrayList<String>();
   umbral=nc;
 }
```

Herencia múltiple: Rebelde (II)

```
  public int alimenta(String alimento,String lugar){
  int alimentado=-1;
  if(!muerto&&(alimento!=null)&&(lugar!=null)){
   if ((lugar.equalsIgnoreCase("barrizales"))&&
       (!(alimento.equalsIgnoreCase("ambrosia"))))
     alimentado=quitaVida(alimento);
   else
     alimentado=daVida(alimento);
   }
   return alimentado;
  }
  private String embosca(String ofensa,String atacado){
    ...
   return ataque;
  }
  public String lucha(int minimo,String recibe){
    ...
    if(minimo>umbral){
     contrataque=embosca(arma,recibe);
    ...
   return contrataque;
  }
```


Herencia múltiple: Rebelde (III)

```
  public void consulta(){
    ...
  }
  public void cultiva(String cultivo){
   int i=0;
   if(!muerto && (cultivo!=null)){
     cultivos.add(cultivo);
     umbral=cultivos.size();
   }
  }
  public String recolecta(String cosecha){
    ...
  }
} // Fin clase Rebelde
```


Herencia múltiple: la tribu

```
package Clases;
import Interfaz.*;
public class Tribu{
 public static void main(String[] args){
  ...
  Ciudadano[] tribu= new Ciudadano[3];
  tribu[0]=new Ciudadano("Neme",10,6.5);
  tribu[1]=new Guerrero("Ber",10,35.2,armas1);//upcasting
  tribu[2]=new Rebelde("Hood",10,2,armas2,5);//upcasting
  for(int i=0;i<tribu.length;i++){
    Ciudadano tranqui=tribu[i];
    tranqui.consulta();
    if (tranqui instanceof Guerrero){
     Guerrero peleon=(Guerrero)tranqui; //downcasting
     repele=peleon.lucha(1,ataques[0]);
    }
    else if (tranqui instanceof Rebelde){
     Rebelde hoo=(Rebelde)tranqui; //downcasting
     hoo.cultiva("centeno");
    }}
 }
} // Fin clase Tribu
```


Beneficios de la herencia

- Reutilización de código.
- Compartición del código.
- Consistencia de interfaz.
- Construcción de componentes.
- Modelado rápido de prototipos.
- Ocultación de información.
- Polimorfismo.

Costes de la herencia

- Velocidad de ejecución.
- Tamaño del programa.
- Sobrecarga de paso de mensajes.
- Complejidad del programa.

Herencia vs. Composición

- La herencia es una relación entre clases, mientras que la composición es una relación entre objetos.
- Donde se detecta una relación de herencia **siempre** es posible transformarla en una relación de composición:
  - Un panadero es un trabajador $\rightarrow$ un panadero tiene un trabajador dentro.
  - Una persona es un animal $\rightarrow$ una persona tiene un animal dentro.
- Donde se detecta una relación de composición **no siempre** es posible transformarla en una relación de herencia:
  - Un coche tiene un motor $\not\rightarrow$ un coche no es un motor.
  - Una persona tiene cerebro $\not\rightarrow$ una persona no es un cerebro.

Elección de técnica de reutilización

- Herencia y composición son los dos mecanismos de reutilización de software más comunes.
  - Herencia: Relación SER-UN $\Rightarrow$ entre clases.
    - Significa contener una clase.
    - Ejemplo: un laptop es un ordenador.

```
class laptop extends ordenador{...}
```

  - Composición: Relación TENER-UN $\Rightarrow$ entre objetos.
    - Significa contener un objeto.
    - Ejemplo: un laptop tiene un determinado tipo de disco duro.

```
class laptop {...
private DiscoDuro particular;
...}
```


Ejemplo

- Problema: Dada la clase Cardinal, definir la clase CardinalRomano con la misma funcionalidad añadiendo la acción de conversión a número romano.

```
public class Cardinal{
  private int valor;
  public Cardinal(int v){valor=v;}
  public int getValor(){return valor;}
  public int suma(int num){return valor+num;}
  ...
}
```

Solución por composición (I)

- Un objeto es una encapsulación de datos y comportamiento.
- Una parte de la nueva clase es una instancia de una clase ya existente.
- La nueva clase contiene un objeto de la clase `Cardinal`.

```
public class CardinalRomano{
  private Cardinal numero;
  public CardinalRomano(int v){
     numero=new Cardinal(v);}
  public int getValor(){return numero.getValor();}
  public int suma(int num){
    return numero.suma(num);}
  public String conversion(){
    ...
    valor=numero.getValor();
    ....
  }
}
```


Solución por composición (II)

- La composición no hace ninguna asunción repecto a la posibilidad de sustitución.
- Los métodos que actúan igual en ambas clases deben ser redefinidos igualmente.
- Los tipos de datos `Cardinal` y `CardinalRomano` son totalmente distintos.
- Se supone que ninguno de ellos puede sustituir al otro en ninguna situación.

Solución por herencia (I)

- La clase nueva se declara como subclase de una clase ya existente.
- `CardinalRomano` hereda de `Cardinal`.

```
public class CardinalRomano extends Cardinal{
  public CardinalRomano(int v){
   super(v);
  }
  public String conversion(){
    ...
    valor=getValor();
    ....
  }
}
```


Solución por herencia (II)

- Los métodos que actúan igual en ambas clases no deben ser redefinidos.
- Permite una definición más breve de la clase.
- Se asume que las subclases son subtipos.
- Las instancias de la subclase deben comportarse de manera similar a las instancias de la superclase (principio de sustitución).
- Cualquier nuevo método de la superclase estará inmediatamente disponible para todas sus subclases.

Resumen (I)

- Uno de los objetivos de la programación orientada a objetos es lograr la reutilización del software a través de la herencia (ahorra tiempo de desarrollo y promueve el uso de software de calidad previamente probado y depurado).
- La herencia se puede hacer a partir de bibliotecas de clases ya existentes.
- El programador de una subclase no necesita acceder al código fuente de su superclase, sólo interactuar con la superclase y su código objeto.
- El programador puede hacer que una nueva clase herede las variables y los métodos miembro de una clase definida previamente. En este caso, la nueva clase se conoce como subclase y la clase preexistente como superclase.
- En herencia simple, una clase hereda de una sola superclase. En la herencia múltiple, una subclase hereda de varias superclases o interfaces (que posiblemente no tengan relación entre ellas).

Resumen (II)

- Una subclase por lo general añade sus propias variables y métodos miembro, por lo que habitualmente tiene una definición mayor que su superclase. Una subclase es más específica que su superclase y por lo general representa menos objetos.
- Una subclase no puede acceder a los miembros `private` de su superclase. Sin embargo, una subclase puede acceder a los miembros `public` y `protected` de su clase base.
- Un objeto de una subclase puede ser tratado como un objeto de la superclase correspondiente. Sin embargo, lo opuesto no es cierto.
- Una superclase existe en una relación jerárquica con sus subclases.

Resumen (III)

- El acceso `protected` sirve como nivel intermedio de protección entre el acceso `public` y el acceso `private`. Se utiliza para extender los privilegios de las subclases.
- Una superclase puede ser una superclase directa o indirecta de una subclase. Una superclase directa es aquella que la subclase *extiende* directamente. Una superclase indirecta es aquella que se hereda de varios niveles más arriba en el árbol jeráraquico.
- Cuando un miembro de la superclase no es adecuado para una subclase podemos redefinirlo en la subclase.
- Es posible convertir implícitamente una referencia a un objeto de una subclase en una referencia a un objeto de su superclase.
- Es importante distinguir entre las relaciones *es un* y *tiene un*. En las relaciones *tiene un*, un objeto de una clase tiene como miembro un objeto de otra clase. En las relaciones *es un*, un objeto de una subclase también puede ser tratado como un objeto de su superclase. *Es un* significa herencia. *Tiene un* significa composición.

Resumen (IV)

- Es posible asignar un objeto de una subclase a una variable de su superclase. Este tipo de asignación tiene sentido, puesto que la subclase tiene miembros que corresponden a cada uno de los miembros de la superclase.
- Es posible convertir una referencia a un objeto de una superclase en una referencia a un objeto de una subclase mediante una conversión *cast* explícita. El destino debe ser una variable de la subclase.
- La lectura de un conjunto de declaraciones de subclase puede ser confusa, ya que no todos los miembros de la subclase están presentes en dichas declaraciones. En concreto, los miembros heredados no se especifican en las declaraciones de las subclases, pero estos miembros de hecho están presentes en ellas.
- En el caso de una subclase, su constructor llama a los constructores de las superclases (para crear e inicializar los miembros de la superclase) y luego al de la subclase (que puede llamar a constructores de objetos miembro).