Capítulo 10

Herencia y polimorfismo

Motivaciones

Supongamos que definimos las clases necesarias para modelar círculos, rectángulos y triángulos. Estas clases tienen muchas características comunes. ¿Cuál es la mejor manera de diseñar estas clases para evitar la redundancia? La respuesta es usar herencia.

Objetivos

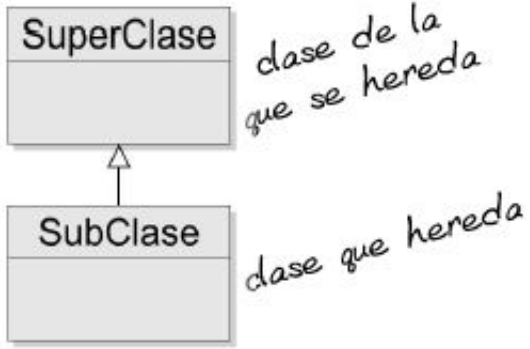
- Conceptos de Herencia.
- Definir una subclase de una superclase a través de la herencia.
- Invocar los métodos y constructores de la superclase utilizando la palabra clave super.
- Sobreescribir los métodos de instancia en la subclase.
- Distinguir las diferencias entre sobreescritura y sobrecarga.
- Descubrir polimorfismo y vinculación dinámica.
- Habilitar datos y métodos en una superclase accesible desde subclases utilizando el modificador de visibilidad protected.
- Evitar la extensión de clases y anulación de método utilizando el modificador final.
- Explorar los métodos de Object de igualdad y toString().

La Herencia en Java

La herencia es el mecanismo en Java por el cual una clase hija o subclase puede heredar las características (atributos y métodos) visibles de otra clase padre o superclase.

En el lenguaje de Java, la clase de la que se hereda se denomina **superclase** y la que hereda se llama **subclase**.

Por lo tanto, una subclase es una versión especializada de una superclase. Hereda todos los miembros (atributos y métodos) definidos por la superclase y agrega sus propios elementos únicos. Esto aumenta su funcionalidad a la vez que evita la repetición innecesaria de código, permitiendo reutilizar el código y funcionalidades y simplificando su desarrollo.



La Herencia en Java

Java soporta la herencia permitiendo a una clase incorporar otra clase en su declaración mediante el uso de la palabra clave **extends**.

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

Ejemplo: Disponemos de la clase Persona y construimos la clase Empleado y le añadiremos un nuevo atributo salario.

```
class Empleado extends Persona {
   double salario;
   ...
}
```

Un objeto de Empleado dispondrá de los atributos heredados y propios.

```
Empleado e = new Empleado();
e.nombre = "Sancho"; //atributos heredados
e.estatura = 1.80;
e.edad = 25;
e.salario = 1725.49; //atributo propio
```

Control Acceso a Miembros en Herencia

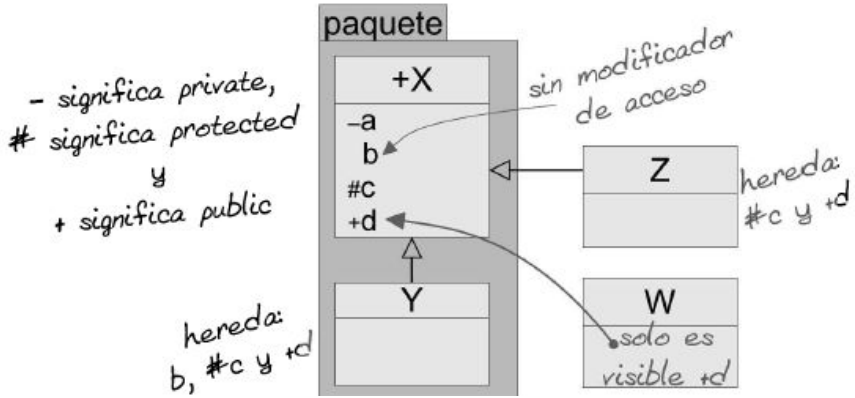
A menudo una variable de instancia de una clase se declarará privada (**private**) para evitar su uso no autorizado o alteración. Heredar una clase no anula la restricción de acceso privado. Por lo tanto, aunque una subclase incluye a todos los miembros de su superclase, no puede acceder a los miembros de la superclase que se han declarado privados si no es a través de un método no privado.

Para que un miembro sea visible desde una subclase podemos hacer uso de un nuevo modificador de acceso, **protected**, que facilita la herencia. Es similar a la visibilidad por defecto, pero los miembros protegidos serán visibles para las clases que hereden, independientemente de si la superclase y la subclase son vecinas o externas.

Un miembro protected es visibles desde las vecinas, no lo es desde las externas, pero siempre es visible desde una clase hija.

Control Acceso a Miembros en Herencia

	Visible desde...			
	la propia clase	clases vecinas	subclases	clases externas
`private`	✓			
sin modificador	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓




```
public class X {
   private int a;    //invisible fuera de la clase
   int b;            //visibilidad por defecto: visible en el paquete
   protected int c; //visibilidad en el paquete y por las subclases
   public int d;     //visibilidad total
}
```

Redefiniendo una subclase

Una subclase hereda una serie de miembros de una superclase, sobre los que podemos:

- ✔Agregar nuevos atributos y nuevos métodos
- ✔Sustituir o sobreescribir los métodos de la superclase
- ✔Ocultar atributos

Ejemplo: Partimos de la superclase **Persona**

```
class Persona {
   String nombre;
   byte edad;
   double estatura;

   void mostrarDatos() {
      System.out.println(nombre);
      System.out.println(edad);
      System.out.println(estatura);
   }
}
```

Redefiniendo una subclase

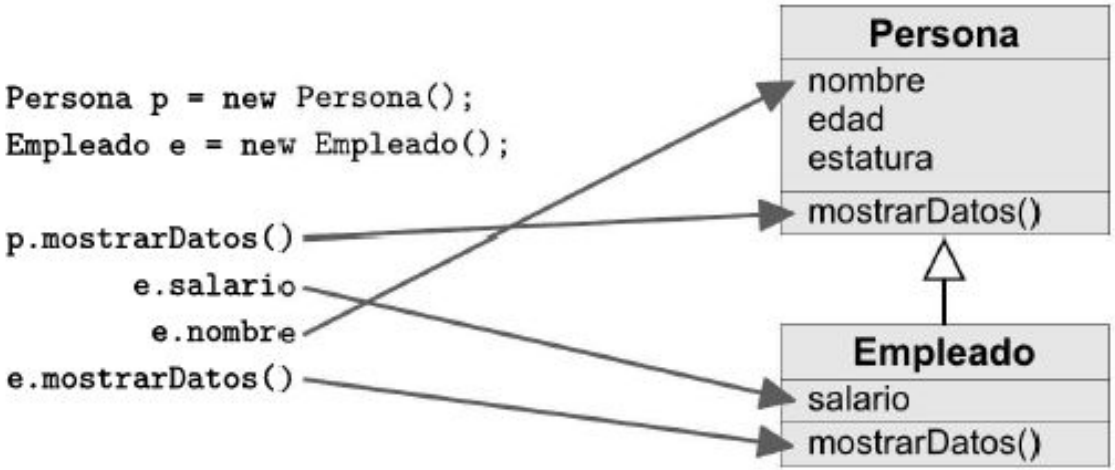
Definimos una subclase **Empleado** que hereda de Persona, donde ocultemos el atributo estatura de tipo double de forma que sea un String que contendrá la talla del uniforme (XXL, XL, L, M,…) y sustituiremos el método mostrarDatos() incluyendo el atributo salario.

Un método de instancia puede ser sobreescrito sólo si es accesible.

```
class Empleado extends Persona {
   double salario;
   String estatura;

   @Override //significa: sustituye un método de la superclase
   void mostrarDatos() {
      System.out.println(nombre);
      System.out.println(edad);
      System.out.println(estatura);
      System.out.println(salario);
   }
}
```

Redefiniendo una subclase



Un método privado no puede sobreescribirse, porque no es accesible fuera de su propia clase. Si un método definido en una subclase es privado en su superclase, los dos métodos están completamente ajenos.

Un método estático puede ser heredado pero no puede ser sobreescrito. Si un método estático definido en la superclase se redefine en una subclase, se oculta el método definido en la superclase.

super y super()

La palabra **super** hace referencia a la superclase de la clase donde se utiliza. Así pues, podríamos redefinir Empleado como:

```
class Empleado extends Persona {
   double salario;

   @Override
   void mostrarDatos() {
      super.mostrarDatos(); //método de la superclase
      System.out.println(salario); //muestra el atributo propio
   }
```

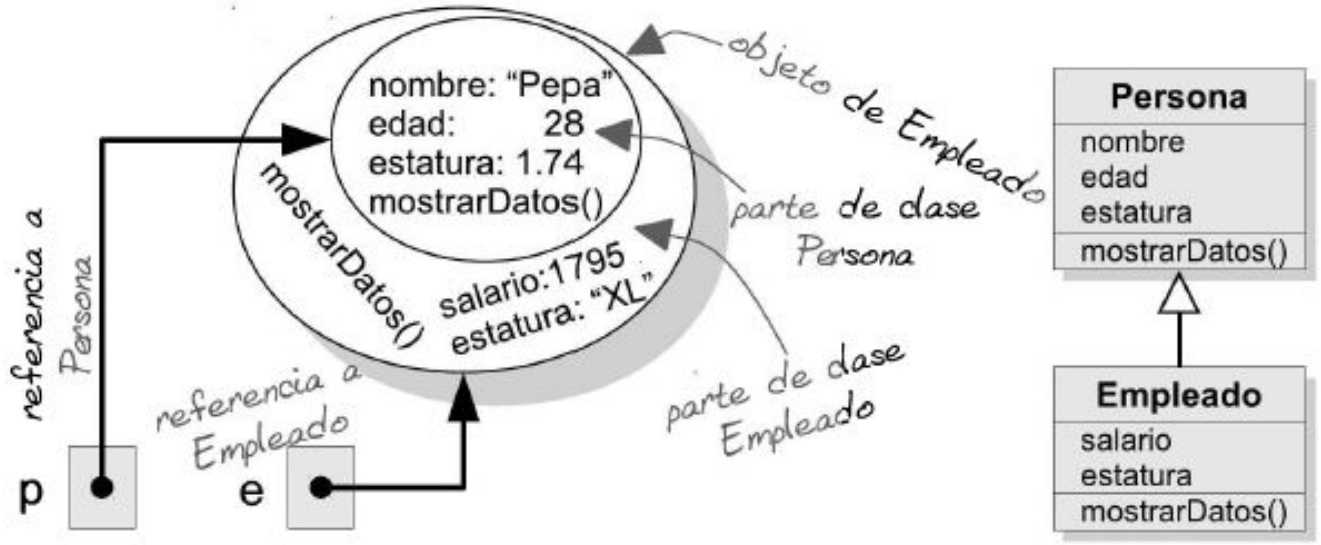
El método **super()** invoca al constructor de la superclase de la clase donde se utiliza. En caso de estar sobrecargado, podemos variar sus parámetros de entrada en número y tipo. Así pues, podríamos redefinir el constructor de Empleado como:

```
Empleado (String nombre, byte edad, double estatura, double salario) {
   super(nombre, edad, estatura); //constructor de Persona
   this.salario = salario;
}
```

Selección dinámica de métodos

Los objetos de una subclase son también objetos de la superclase. Así pues, un objeto Empleado será, al mismo tiempo, un objeto Persona y por tanto, podemos referenciar un objeto Empleado usando una variable Persona:

```
Empleado e = new Empleado();
Persona p = e;
```



Selección dinámica de métodos

Los atributos accesibles son los definidos en la variable (ocultación).

```
p.estatura //atributo de Persona de tipo double
e.estatura //atributo de Empleado de tipo String
```

Los métodos accesibles sin embargo, ejecutan la versión del objeto referenciado, es decir de la subclase Empleado (overriding).

```
p.mostrarDatos(); //método de Empleado
e.mostrarDatos(); //método de Empleado
```

La selección de métodos en tiempo de ejecución nos permite ejercer el ***polimorfismo***.

En POO, se llama polimorfismo a la capacidad que tienen los objetos de distinto tipo (de distintas clases) de responder al mismo método.

El polimorfismo permite que los métodos se utilicen genéricamente para una amplia gama de argumentos de objeto. Esto se conoce como *programación genérica*.

Selección dinámica de métodos

Por ejemplo, supongamos que una tercera clase Cliente hereda de Persona:

```
class Cliente extends Persona {
    ...
    @Override
    void mostrarDatos() {
        ...
    }
}
```

Ejemplo: Si creamos una variable de tipo Persona con ella podemos referenciar tanto objetos de clase Empleado, como Cliente, como Persona. Para todos ellos disponemos de mostrarDatos(), y se ejecutará una u otra versión, según el objeto.

```
Persona p;
p = new Persona();
p.mostarDatos(); //se ejecuta el método de Persona
p = new Empleado();
p.mostarDatos(); //se ejecuta el método de Empleado
p = new Cliente();
p.mostarDatos(); //se ejecuta el método de Cliente
```

Compatibilidad entre clases mediante herencia

Cuando tenemos una relación de herencia, la subclase genera un tipo compatible con el de la superclase, ello nos permite usar un objeto de la subclase en cualquier lugar donde usemos la superclase (propiedad llamada polimorfismo).

Ejemplo: la clase Persona, que define entre otras cosas un atributo llamado nombre, y la clase Estudiante, que hereda de Persona y define otros atributos propios de un estudiante (curso, notas, etc). Creamos un objeto de tipo Estudiante y lo referenciamos con una variable de tipo Persona:

Persona p = new Estudiante(…);

Mientras lo tengamos en esa variable sólo podremos acceder a los miembros definidos en la clase Persona (aunque el objeto está completo y sigue teniendo sus notas, etc)

Compatibilidad entre clases mediante herencia

Si queremos usarlo como un objeto de tipo Estudiante y acceder a sus notas, por ejemplo, tendremos que hacerle un casting.

```
Estudiante e = (Estudiante)p;
```

Normalmente a esta operación se le llama *downcasting*, ya que hacemos una conversión de tipo hacia abajo en el árbol de herencia.

Incluso podríamos tener, por ejemplo, un array de Personas en el que algunos elementos sean objetos de tipo Persona y otros de tipo Estudiante, y podríamos hacer un recorrido de dichos elementos tratándolos a todos como objetos de tipo Persona:

```
Persona[] personas = new Persona[…];
personas[0] = new Persona(…);
personas[1] = new Estudiante(…);
...
```

Compatibilidad entre clases mediante herencia

```
for (Persona p : personas) {
System.out.println(p.getNombre();
}
```

Si quisiéramos distinguir si un objeto determinado es de la clase Estudiante, podemos usar el operador ***instanceof***, que nos devuelve true si el primer operando es del tipo especificado por el segundo operando, y false en caso contrario:

```
if (p instanceof Estudiante) {
    Estudiante e = (Estudiante)p;
    System.out.println(e.getCurso());
}
```

Si intentáramos hacer un downcasting incorrecto, como por ejemplo, convertir a Estudiante un objeto de tipo Persona, se generaría una excepción.

El Modificador final

Los modificadores se usan en clases y miembros de la clase (datos y métodos), excepto que el modificador final también se puede usar en variables locales en un método. Una variable local final es una constante dentro de un método.

- La clase `final` no puede ser extendida:

```
final class Math {
  ...
}
```

- Una variable `final` es una constante:

```
final static double PI = 3.14159;
```

- Un método `final` no puede ser anulado por sus subclases.

¿Qué se puede hacer en una Subclase?

- Los campos heredados se pueden usar directamente, al igual que cualquier otro campo.
- Declarar nuevos campos en la subclase que no están en la superclase.
- Los métodos heredados se pueden usar directamente tal como son.
- Podemos escribir un nuevo método de instancia en la subclase que tenga la misma firma que el de la superclase, anulándolo
- Se puede escribir un nuevo método estático en la subclase que tiene la misma firma que el de la superclase, escondiéndolo así.
- Podemos declarar nuevos métodos en la subclase que no están en la superclase.
- Podemos escribir un constructor de subclase que invoca el constructor de la superclase, ya sea implícitamente o mediante la palabra clave super.
- Una subclase no puede reducir la accesibilidad

Observaciones sobre Constructores

Los constructores no se heredan.

Debe usar la palabra clave super para llamar al constructor de la superclase. Invocar el nombre de un constructor de una superclase en una subclase provoca un error de sintaxis. Java requiere que la declaración que usa la palabra clave super aparezca la primera en el constructor de la subclase.

En caso de estar sobrecargado, podemos variar sus parámetros de entrada en número y tipo.

Sintaxis:

super(argumentos constructor superclase)

Encadenamiento de Constructores

La construcción de una instancia de una clase invoca todos los constructores de las superclases a lo largo de la cadena de herencia. Esto se conoce como ***encadenamiento de constructores***, de ahí la importancia del constructor no-arg. Por ejemplo, la codificación siguiente genera un error:

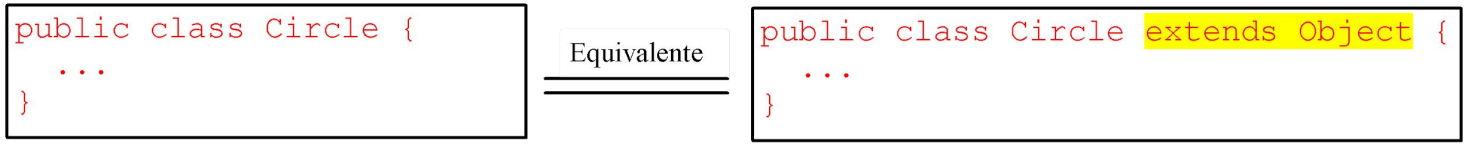
```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Se invoca el constructor de Fruit");
  }
}
```

NOTA: Como no se define ningún constructor en Apple, el constructor no arg predeterminado de Apple se define implícitamente. Como Apple es una subclase de Fruit, el constructor predeterminado de Apple invoca automáticamente el constructor no arg de Fruit.
Sin embargo, Fruit no tiene un constructor no-arg, porque Fruit tiene un constructor explícito definido. Por lo tanto, el programa genera errores y no se puede compilar.

La Clase Object y sus Métodos

Cada clase en Java desciende de la clase java.lang.Object. Si no se especifica ninguna herencia cuando definimos una clase, la superclase de nuestra clase será Object.



¿Qué se consigue con que todas las clases hereden de Object?

- Todas hereden un conjunto de métodos de uso universal como realizar comparaciones entre objetos, clonarlos o representar un objeto como cadena.
- Poder referenciar cualquier objeto, de cualquier tipo, mediante una variable de tipo Object.

Así pues, una variable de cualquier tipo definido, tendrá todos los atributos y métodos de su clase más los heredados de Object.

Ejercicios básicos de Herencia de clases

Ejercicio 1a: Clase base y subclase básica

Crea una clase llamada Animal con los atributos nombre y edad. Agrega un método llamado hacerSonido() que imprima "El animal hace un sonido". Luego crea una subclase llamada Perro que herede de Animal y sobrescriba el método hacerSonido() para imprimir "El perro ladra".

Ejercicio 1b: Herencia con métodos adicionales

Extiende la clase Animal para incluir una subclase llamada Gato. El Gato debe tener un método adicional llamado cazarRatones(), que imprima "El gato está cazando ratones".

Ejercicio 1c: Polimorfismo

Crea un array de 5 animales que incluya instancias de Perro y Gato. Recorre el array y llama al método hacerSonido() para cada animal.

Ejercicios básicos de Herencia de clases

Ejercicio 2: Herencia con Constructor y Método

Crea una clase Vehiculo con los atributos marca y modelo. Agrega un método llamado mostrarDetalles() que imprima esos atributos. Crea una subclase Coche que añada un atributo adicional llamado puertas y sobrescriba el método mostrarDetalles() para incluir el número de puertas.

Crea un array que contenga varios objetos de la clase Coche. Recorre el array e imprime los detalles de cada coche.

Ejemplo 3: Herencia con Métodos Adicionales

Crea una clase Persona con atributos nombre y edad. Incluye un método presentarse() que imprima "Hola, soy {nombre} y tengo {edad} años". Crea una subclase Estudiante que añada un atributo carrera y un método adicional estudiar() que imprima "Estoy estudiando {carrera}".

Ejercicios básicos de Herencia de clases

Ejercicio 4: Herencia en Cadena

Crea una clase Electrodomestico con un atributo marca. Luego crea una subclase Televisor que añada un atributo tamañoPantalla. Finalmente, crea una subclase SmartTV que añada un atributo conexionInternet y sobrescriba el método mostrarDetalles() para incluir todos los atributos.

Ejemplo 5: Uso de protected en Herencia

Crea una clase Empleado con un atributo protected llamado salario. Crea una subclase Gerente que añada un atributo bono y un método para calcular el salario total (salario + bono).