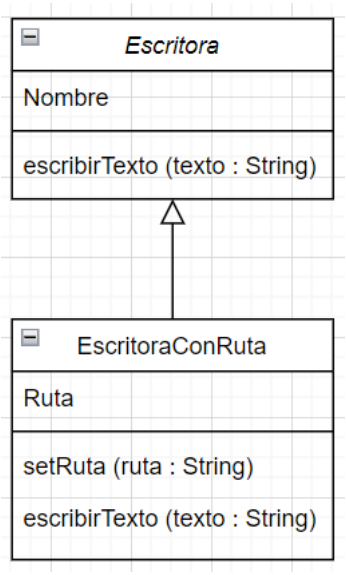
Herencia

En la Programación Orientada a Objetos, la **Herencia** es un mecanismo que permite crear **nuevas clases a partir de otras** ya existentes, que se suponen comprobadas y correctas. Se evita así el problema de tener que andar rediseñando, reescribiendo y modificando una clase que ya existente (y funciona) o duplicando código innecesariamente.



Por ejemplo, tenemos una clase **Escritora** que permite escribir texto un fichero. Esta clase funciona perfectamente, pero nuestro programa necesita una clase que además permita elegir la ruta donde se creará el fichero. En lugar de escribir **EscritoraConRuta** desde cero, podemos utilizar la herencia de clases.

Al hacer que **EscritoraConRuta** herede de **Escritora**, obtiene todo el comportamiento (métodos) y los atributos (variables) de su superclase. Dispondrá del atributo nombre y del método escribirTexto () como si fuese **una copia** de su superclase. Pero eso no impide que modifiquemos cosas en la nueva clase. Si te fijas, **EscritoraConRuta** tiene un atributo ruta y un método setRuta () que su superclase no tiene. Adicionalmente, el hecho de que en el esquema aparezca el método escribirTexto () significa que, aunque lo tenga por herencia, lo hemos **reescrito** y su comportamiento será diferente al de **Escritora**.

A la clase de la que derivan otras clases que extienden su funcionalidad se la denomina clase base, **clase padre** o superclase; mientras que las clases derivadas se denominan **clases hijas** o subclases.

En el caso de Java, TODAS las clases heredan SIEMPRE de otras clases. Esto crea una estructura de árbol jerárquica en la que la clase **Object** se encuentra en la raíz. Cuando creas una nueva clase del tipo que sea, se te plantean dos opciones: no indicar nada, lo que hace que tu clase herede de Object; o indicar específicamente cuál es tu superclase.

En Java NO ES POSIBLE que una clase herede de dos superclases a la vez.

Herencia sencilla - Extends

Para indicar que una clase hereda de otra, basta con indicarlo con la palaba reservada **extends**. Mira el ejemplo siguiente. La clase Escritora dispone de un atributo nombre, un método escribirTexto () y un getter-setter.

```
public class Escritora {

    private String nombre = null;

    public void escribirTexto (String texto) {
        // Some code...
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Vamos a crear ahora a **EscritoraConRuta** como se indica en el esquema anterior, pero indicando que es una subclase de Escritora:

```
public class EscritoraConRuta extends Escritora{

    private String ruta = null;

    public void escribirTexto (String texto) {
        // Some code...
    }

    public String getRuta() {
        return ruta;
    }

    public void setRuta(String ruta) {
        this.ruta = ruta;
    }

}
```

Vamos a ve qué sucede si intentamos instanciar **Escritora** y **EscritoraConRuta**.

```
public static void main(String[] args) {

    Escritora escritora = new Escritora ();
    escritora.setNombre("nombre");
    escritora.setRuta("ruta");
    escritora.escribirTexto("some text");

    EscritoraConRuta escritoraConRuta = new EscritoraConRuta();
    escritoraConRuta.setNombre("nombre");
    escritoraConRuta.setRuta("ruta");
    escritoraConRuta.escribirTexto("some text");
}
```

Si nos fijamos, el código contiene un error evidente: no podemos usar método setRuta () de la variable **escritora** porque la clase **Escritora** no lo tiene definido.

Sin embargo, si te fijas, **escritoraConRuta** no genera un error al usar setNombre (). Esto se debe a que la clase de **EscritoraConRuta** lo hereda de **Escritora** y, por tanto, ella también lo tiene accesible.

Finalmente, ambas clases tienen un método que se llama escribirTexto (). En este caso, cada variable lanzará el método que se corresponde **con su clase** en concreto.

Herencia compleja - Abstracción

La **abstracción** es una manera ocultar los detalles de una implementación, de manera que el programador sólo acceda a las partes del código que son realmente importantes. Se usa principalmente para evitar errores en los diseños de las aplicaciones que tienes cientos de clases. Si tú evitar que otro programador acceda a lo que no debe, o le fuerzas a programar ciertas clases siguiendo un formato; te estarás evitado muchos problemas.

Hay dos formas de conseguir la **abstracción** en Java.

- Clases Abstractas: No te deja instanciar la clase.
- Interfaces: Obliga a otras clases a implementar ciertos métodos.

Independientemente de que hagas una cosa o la otra, las reglas que hemos visto de herencia se siguen aplicando. Por lo tanto, una Interfaz puede heredar de otra Interfaz y una Clase Abstracta puede heredar de otras clases.

Clases Abstractas

Vamos a convertir a **Escritora** en una Clase Abstracta. Esto es tan sencillo como hacer esto:

```
public abstract class Escritora {

    private String nombre = null;

    public void escribirTexto (String texto) {
```

Ahora, si intentamos instanciar la clase **Escritora**, nos dará un error.

```
public static void main(String[] args) {
    Escritora escritora = new Escritora ();
    escritora.setNombre("nombre");
    escritora.escribirTexto("some text");
}
```

Este error se produce porque una clase abstracta no se puede instanciar.

También es posible definir un **método abstracto**. En este caso, los métodos definidos como abstractos no tienen cuerpo. No obstante, toda clase que herede de una clase con métodos abstractos debe de implementarlos, o se generará un error. A menos claro que sea a su vez una clase abstracta, con lo que se repite el proceso.

Por ejemplo, añadamos a la clase **Escritora** un método leerTexto (). Este método no se va a usar en **Escritora**, pero deberá ser implementado por todas las clases que hereden de ella, como **EscritoraConRuta**. Por tanto, podemos hacer:

```
public abstract class Escritora {

    private String nombre = null;

    public abstract String leerTexto();

    public void escribirTexto (String texto) {
```

Veremos entonces que **EscritoraConRuta** nos genera un error:

```
public class EscritoraConRuta extends Escritora{

    private String ruta = null;
```

Este error se debe que estamos obligados a implementar **leerTexto ()**.

```
public class EscritoraConRuta extends Escritora{

    private String ruta = null;

    @Override
    public String leerTexto() {
        // TODO Auto-generated method stub
        return null;
    }
```

@Override es una **anotación**. Sirve para que el programados sepa que ese método está sobrescribiendo un método heredado de la clase padre. No es necesario, pero deberíamos marcar todos los métodos nosotros mismos si el IDE no lo hace automáticamente. Distintas anotaciones tienen diferentes usos, que se verán en otras asignaturas.

Interfaz

Una interfaz se emplea generalmente para 'obligar' a una clase a implementar ciertos métodos. Se usa principalmente en los proyectos para conseguir que el código de clases similares siga el mismo formato y contengan los mismos métodos. Esto permitirá facilitar la búsqueda de errores, la automatización, el mantenimiento, etc.

Una Interfaz es una clase, sólo que se escribe de forma diferente. Vamos a crear una interfaz para nuestro ejemplo, llamada **FicheroInterfaz**. Esta interfaz va a obligar a implementar los siguientes métodos: leerTexto (), escribirTexto (), borrarTexto ().

```
public interface FicheroInterfaz {

    public String leerTexto();

    public void escribirTexto (String texto);

    public void borrarTexto (String texto);
}
```

Ahora, vamos a hacer que **EscritoraConRuta** implemente **FicheroInterfaz**.

```
public class EscritoraConRuta implements FicheroInterfaz{
```

Nos genera un error: **EscritoraConRuta** debe implementar todos los métodos de **FicheroInterfaz**, tanto si nos gusta como si no.

```
public class EscritoraConRuta implements FicheroInterfaz{

    private String ruta = null;

    @Override
    public void escribirTexto (String texto) {
        // Some code...
    }

    @Override
    public String leerTexto() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void borrarTexto(String texto) {
        // TODO Auto-generated method stub

    }
```

Mezclando Clases Abstractas e Interfaces

Es posible mezclar Clases Abstractas e Interfaces. Dicho esto, debe hacerse con cuidado y siempre durante la fase de diseño de un proyecto.

Lo que nunca debe hacerse es:

- Añadir código ejecutable en un interfaz.
- Forzar una estructura de herencia innecesaria.
- Generar clases abstractas 'por si acaso las uso más adelante'.

Clases Genéricas

En ocasiones, es posible que queramos definir una clase (o interfaz) que trabaja con un tipo de dato en concreto. Lo que sucede es que **no conoceremos** el tipo de dato **hasta que el programa esté en ejecución**. O lo que es lo mismo, queremos hacer clases (o interfaces) que manejen cualquier tipo de dato de forma genérica.

```
public class EscritoraGenerica <T> {

    private T t = null;

    public EscritoraGenerica (T t) {
        this.t = t;
    }

    public void escribir (T t) {
        // Some code...
    }

    public T leer () {
        // Some code...
        return null;
    }

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}
```

La clase **EscritoraGenérica** es una clase que es capaz de escribir a y leer desde de un fichero. Es capaz de gestiona cualquier tipo de objeto, representado por la T. Dicha T es simplemente el nombre del parámetro genérico, cuyo tipo concreto se conocerá cuando el programa se ejecute. De esta forma, **EscritoraGenérica** podrá leer y escribir String, Integer, Alumnos, etc. La forma correcta de instanciar esta clase es:

```
EscritoraGenerica esc1 = new EscritoraGenerica();

EscritoraGenerica<String> esc2 = new EscritoraGenerica<String>("Hola, Mundo!");
EscritoraGenerica<Integer> esc3 = new EscritoraGenerica<Integer>(4);
EscritoraGenerica<Alumno> esc4 = new EscritoraGenerica<Alumno>(new Alumno());
```

Si te fijas, una clase que emplea genéricos necesita conocer el tipo de objeto al ser instanciada. De no ser así, no sabría con qué substituir la T.

Existen una serie de convenciones para nombrar a los genéricos:

- E – Element (usado bastante por Java Collections Framework)
- K – Key (Llave, usado en mapas)
- N – Number (para números)
- T – Type (Representa un tipo, es decir, una clase)
- V – Value (representa el valor, también se usa en mapas)
- S, U, V etc. – usado para representar otros tipos.

Uso de Genéricos – Instanceof

En Java, uno de los mayores problemas que se nos presentan es manejar objetos de un tipo concreto en un punto del programa; para vernos obligados a considerarlos genéricos en otro; y finalmente tener que volver a decirle a que ese objeto en realidad es de un tipo concreto.

Un caos. Por eso, Java nos deja hacer cosas como esta:

```
Alumno alumno = new Alumno ();
Profesor profesor = new Profesor ();
Bedel bedel = new Bedel ();

List <Persona> personas = new ArrayList <Persona> ();
personas.add(alumno);
personas.add(profesor);
personas.add(bedel);
```

¿Cómo es posible que no de error? Estamos metiendo en una lista de objetos Persona una serie de objetos que NO son Persona. Bien, esto se debe que Alumno, Profesor y Bedel heredan de la clase Persona.

El problema que se nos plantea ahora es recorrer la lista de Persona. Fíjate en este código. Aparentemente es correcto y no tiene errores de compilación:

```
for (Persona persona : personas) {
    persona.getNombre();
}
```

Pero **el programa podría fallar** en tiempo de ejecución. Esto se debe a que el objeto que sacas de la lista lo sacas como Persona, pero en realidad en la lista tenemos Alumnos, Profesores y Bedeles. Y lo que es peor, como son Persona hay métodos de Alumno que no podemos ejecutar porque, para empezar, no sabemos si es un Alumno o no.

Una forma de solucionarlo es mediante el **instanceof**, que nos permite averiguar la clase de un objeto en concreto. Si acertamos, bastaría con **convertir el objeto de un tipo a otro**. (Casting)

```
for (Persona persona : personas) {
    if (persona instanceof Alumno) {
        Alumno alumno = (Alumno) persona;
        // Es un alumno...

    } else if (persona instanceof Profesor) {
        Profesor profesor = (Profesor) persona;
        // Es un profesor...

    }else if (persona instanceof Bedel) {
        Bedel bedel = (Bedel) persona;
        // Es un bedel...

    } else {
        System.out.println("Error!");
    }
}
```