UT 7 UTILIZACIÓN AVANZADA DE CLASES

Composición, herencia, clases abstractas, interfaces y polimorfismo

Relaciones entre clases

Relaciones entre clases (I)

- La clase se describe como un mecanismo de definición para construir objetos.
- Es vital establecer relaciones entre clases al diseñar conjuntos de clases para automatizar el tratamiento de la información.

Relaciones entre clases (II)

Tipos de relaciones entre clases

- **Clientela**: Una clase utiliza objetos de otra clase, por ejemplo, al pasarlos como parámetros.
- **Composición**: Un atributo de una clase es un objeto de otra clase.
- **Anidamiento**: Definición de clases dentro de otra clase.
- **Herencia**: Una clase comparte características con otra (clase base), añadiendo funcionalidades específicas (especialización).

Relaciones entre clases (y III)

- La relación de clientela se utiliza comúnmente en programación Java.
- La relación de composición implica que una clase tiene objetos de otra clase como atributos.
- La relación de anidamiento implica declarar clases dentro de otras para un mayor encapsulamiento.
- La relación de herencia se ve en clases derivadas de otras, por ejemplo en interfaces gráficas.
- Composición y anidamiento son casos particulares de clientela.
- La implementación de clases utiliza estas relaciones, centrándose en herencia para establecer relaciones más complejas.

Composición (I)

- Se da una relación de composición si la entidad A contiene a otra entidad B como una de sus partes.
- Es decir, la clase A contiene uno o varios objetos de la clase B.
- Ejemplos de Composición:
  - Un país contiene comunidades autónomas, una comunidad autónoma contiene provincias, y una provincia contiene municipios.
- La composición puede encadenarse hasta llegar a objetos básicos del lenguaje o tipos primitivos.

Composición (II)

- Definir clases mediante otras clases ya definidas anteriormente sirve para reutilizar el código de forma eficiente
- Clases que describen entidades distinguibles y con funciones claramente definidas pueden reutilizarse para representar objetos similares.

Composición (III)

¿Cómo identificar la relación de composición?

- Se identifica la relación de composición cuando una clase contiene un atributo que es una referencia a un objeto de otra clase.
- La expresión "tiene un" se usa para determinar la relación: "la clase A tiene uno o varios objetos de la clase B".

Composición (y IV)

- Otros ejemplos de relación de composición
  - Un coche tiene un motor y cuatro ruedas.
  - Una persona tiene un nombre, una fecha de nacimiento y una cuenta bancaria asociada.
  - Un cocodrilo bajo investigación tiene un número de dientes, una edad y coordenadas de ubicación geográfica.
  - Un rectángulo contiene dos objetos de la clase Punto para almacenar vértices.
  - Un empleado contiene un objeto de la clase DNI para almacenar su identificación y otro objeto de la clase CuentaBancaria para guardar su cuenta de nómina.

HERENCIA (I)

- Herencia en Programación Orientada a Objetos (POO) es un mecanismo que permite crear clases basadas en otras ya existentes.
- En Java, se implementa mediante la palabra clave **extends.**
- La herencia simplifica la creación de nuevas clases al reutilizar la funcionalidad de clases existentes (clase padre o superclase).
- La subclase hereda atributos, métodos y clases internas de la clase padre.
- Los constructores no se heredan pero pueden ser invocados desde la subclase.

HERENCIA (II)

Ejemplos de Herencia:

- Un coche es un vehículo.
- Un empleado es una persona.
- Un rectángulo es una figura geométrica en el plano.
- Un cocodrilo es un reptil.
- Una ventana en una aplicación gráfica puede heredar de JFrame.
- Una caja de diálogo puede ser un tipo de JDialog.

Herencia (III)

Expresión Idiomática para Identificar Herencia:

- "La clase A es un tipo específico de la clase B" (especialización).
- "La clase B es un caso general de la clase A" (generalización).

Herencia (y IV)

La clase **Object** en Java

- Define y implementa comportamientos comunes a todas las clases, incluyendo las definidas por el usuario.
- Toda clase en Java deriva, en última instancia, de la clase **Object**.
- Existe una jerarquía de clases que tiene a la clase **Object** en la raíz.

Composición vs. Herencia (i)

- **Composición**: Una clase está formada por objetos de otras clases. Los objetos incluidos son atributos miembros de la clase que se está definiendo.
- **Herencia**: Una clase cumple todas las características de otra. La clase derivada es una especialización de la clase base.

Composición vs. Herencia (ii)

Ejemplo de Uso: Clase Punto y Clase Círculo

- **Herencia**: La clase Círculo deriva de la clase Punto para aprovechar sus atributos x1 e y1, y añadir otros atributos y métodos específicos como el radio y cálculos de área y perímetro.
- **Composición**: ¿es más apropiado establecer una relación de composición en lugar de herencia?

Composición vs. Herencia (iii)

- "Un círculo es un punto" (Herencia): Plantea problemas de conceptualización y jerarquía de clases.
- "Un círculo tiene un punto" (Composición): Refleja una relación más precisa entre las clases.
- Es importante considerar si una clase es un tipo de otra (herencia) o si contiene elementos de otra (composición).
- La pregunta clave: **"¿A es un tipo de B?" o "¿A contiene elementos de tipo B?"**.

COMPOSICIÓN

SINTAXIS DE LA COMPOSICIÓN (I)

- No se necesita una sintaxis especial para indicar que una clase contiene objetos de otra clase.
- Cada objeto contenido es un atributo que debe declararse dentro de la clase.
- Sintaxis:

```
class <nombreClase> {
    [modificadores] <NombreClase1> nombreAtributo1;
    [modificadores] <NombreClase2> nombreAtributo2;
}
```

SINTAXIS DE LA COMPOSICIÓN (Y II)

- Ejemplo: declaramos una clase Rectangulo que contiene puntos en el plano
- La clase Rectangulo tiene una relación de composición con la clase Punto, pues un rectángulo está compuesto de 2 puntos en el plano para definirlo

```
class Rectangulo {
        private Punto vertice1;
        private Punto vertice2;
}
```

USOS DE LA COMPOSICIÓN (I)

- Cuando se escriben clases que contienen objetos de otras clases hay que tener precaución con aquellos métodos que devuelvan información acerca de los atributos/campos de la clase (métodos get)
- Cuando hablamos de las propiedades con los métodos get y set vimos que a través de estos métodos se controlaba el acceso a los atributos/campos de la clase y su actualización
- Hasta ahora solamente hemos tratado con tipos básicos a la hora de definir atributos/campos

USOS DE LA COMPOSICIÓN (II)

- Al devolver un tipo básico (tipo por valor) en un método get se devuelve automáticamente una copia del dato, por lo que el valor original dentro de la clase permanece sin alterar independientemente de lo que haga el código que llama al método get
- Cuando se devuelve un objeto (tipo por referencia) en un método get se está devolviendo una referencia al dato original dentro de la clase, por lo cual se podría modificar directamente desde código fuera de la clase sin pasar por el método set

USOS DE LA COMPOSICIÓN (III)

- Para evitar ese tipo de situaciones lo anterior hay diversas alternativas para evitar la devolución directa de un atributo que sea un objeto:
  - Devolver siempre tipos primitivos.
  - Crear un nuevo objeto que sea una copia del atributo que quieres devolver y utilizar ese objeto como valor de retorno. Es decir, crear una copia del objeto especialmente para devolverlo.
- Es posible que en algunos casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos).

USOS DE LA COMPOSICIÓN (IV)

```
public Punto obtenerVertice1 ()   // Creación de un nuevo punto extrayendo sus
atributos
{
        double x, y;
        Punto p;
        x= this.vertice1.obtenerX();
        y= this.vertice1.obtenerY();
        p= new Punto (x,y);
        return p;
}
```

USOS DE LA COMPOSICIÓN (V)

```
public Punto obtenerVertice1 ()  // Utilizando el constructor copia de Punto (si es que
está definido)
{
        Punto p;
        p = new Punto (this.vertice1); // Uso del constructor copia
        return p;
}
```

USOS DE LA COMPOSICIÓN (VI) - CONSTRUCTORES

- A la hora de escribir clases que contengan como atributos objetos de otras clases hay que tener en cuenta que el constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.
- Al pasar un objeto (tipo de dato por referencia) que se pasan como parámetros para rellenar el contenido de los atributos es conveniente hacer una copia de esos objetos y utilizar esas copias
- Si no, sería posible que el código que llama al constructor tuviera acceso a un objeto que sea un campo privado de la clase al guardar la referencia

Usos de la composición (VII) - Constructores

- Si además el objeto parámetro que se pasa al constructor formaba parte de otro objeto y esos objetos son modificados en el futuro desde otra parte del código ajeno a la clase se pueden producir efectos colaterales inesperados
- Los objetos al ser tipos de datos por referencia realmente apunta a la zona de memoria en la que se encuentra el objeto. Por eso, puedes tener un único objeto y múltiples referencias a él, pero modificar un objeto desde cualquier referencia lo modifica para todas

Usos de la composición (VIII) - Métodos set

- Sólo se crean objetos cuando se llama a un constructor (usando new XXX). Al realizar asignaciones o pasos de parámetros no se están copiando o pasando copias de los objetos, sino solamente las referencias al mismo objeto
- Si en un método set que reciba un objeto no hacemos una copia del mismo para almacenarlo como un objeto nuevo, nuestra clase estaría ofreciendo una referencia al objeto interno que posee a un código externo permitiendo modificarlo

USOS DE LA COMPOSICIÓN (IX)

```
public Rectangulo (Punto vertice1, Punto vertice2)
{
	this.vertice1= new Punto (vertice1.obtenerX(), vertice1.obtenerY() );
	this.vertice2= new Punto (vertice2.obtenerX(), vertice2.obtenerY() );
}
public Rectangulo (Punto vertice1, Punto vertice2)
{
	this.vertice1= new Punto (vertice1 );
	this.vertice2= new Punto (vertice2 );
}

public Rectangulo (Rectangulo r) {
	this.vertice1= new Punto (r.obtenerVertice1() );
	this.vertice2= new Punto (r.obtenerVertice2() );
}
```

Clases anidadas (I)

- Es posible definir una clase dentro de otra clase, es decir, dentro del propio bloque de definición de otra clase

```
class claseContenedora { // Cuerpo de la clase

      class claseInterna { // Cuerpo de la clase interna

      }
}
```

Clases anidadas (II)

Tipos de clases internas:

- **Clases internas estáticas** declaradas con el modificador static.
- **Clases internas miembro,** conocidas habitualmente como clases internas.
- **Clases internas locales,** que se declaran en el interior de un bloque de código (normalmente dentro de un método).
- **Clases anónimas,** similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la gestión de eventos en los interfaces gráficos.

Clases anidadas (y III)

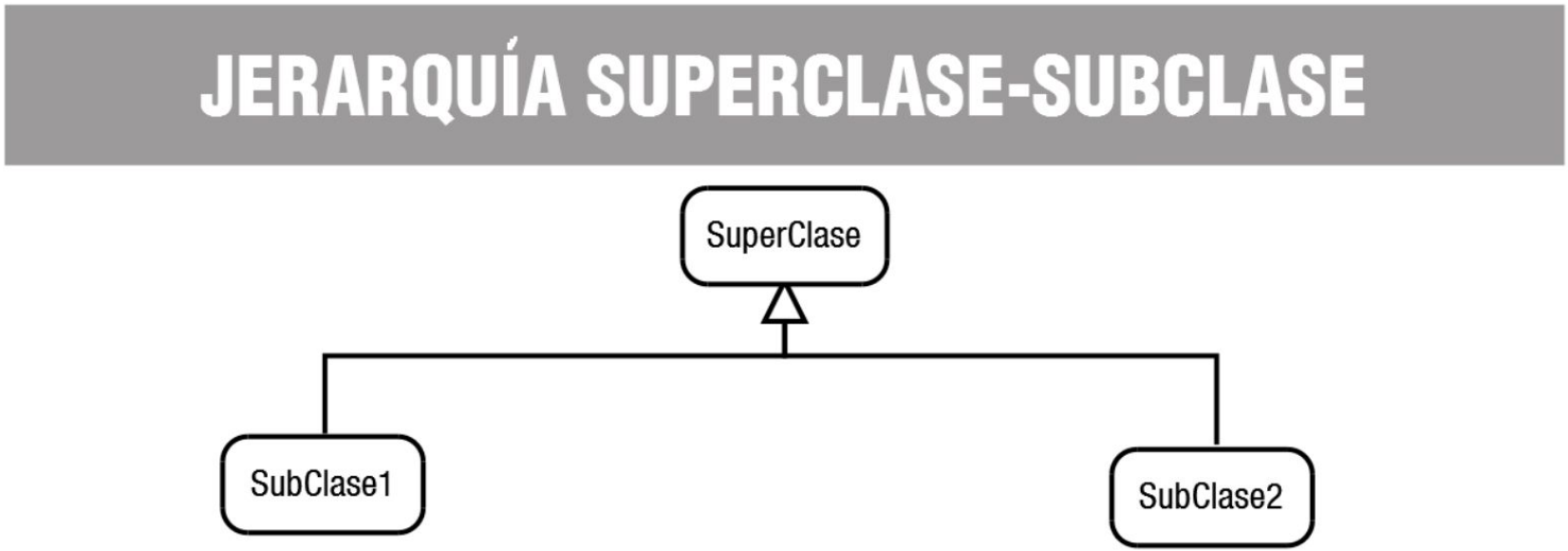
- Las clases anidadasnpueden ser declaradas con los modificadores public, protected, private o de paquete, como el resto de miembros.
- Las clases internas (no estáticas) tienen acceso a otros miembros de la clase dentro de la que está definida aunque sean privados
- Las clases internas estáticas solamente tienen acceso a miembros estáticos también de la clase que las contiene

Herencia

Conceptos de la herencia de clases (I)

- La herencia es el mecanismo que permite definir una nueva clase a partir de otra, pudiendo añadir nuevas características, sin tener que volver a escribir todo el código de la clase base.
- La clase de la que se hereda se llama **clase base, clase padre o superclase**. A la clase que hereda se le suele llamar **clase hija, clase derivada o subclase**.

Conceptos de la herencia de clases (II)



Conceptos de la herencia de clases (III)

- Una clase derivada puede ser a su vez clase padre de otra que herede de ella y así sucesivamente dando lugar a una jerarquía de clases
- Una clase hija no tiene acceso a los miembros privados de su clase padre, tan solo a los públicos (como cualquier parte del código tendría) y los protegidos (a los que sólo tienen acceso las clases derivadas y las del mismo paquete).
- Aquellos miembros que sean privados en la clase base también habrán sido heredados, pero el acceso a ellos estará restringido al propio funcionamiento de la superclase y sólo se podrá acceder a ellos si la superclase ha dejado algún medio indirecto para hacerlo (por ejemplo a través de algún método).

Conceptos de la herencia de clases (IV)

- Todos los miembros de la superclase son heredados por la subclase. Algunos de estos miembros heredados podrán ser redefinidos o sobrescritos (overriden) y también podrán añadirse nuevos miembros.
- Se podría decir que se está “ampliando” la clase base con características adicionales o modificando algunas de ellas (proceso de especialización).
- Una clase derivada extiende la funcionalidad de la clase base sin tener que volver a escribir su códigoya que hereda todas las propiedades y métodos de la clase padre

Sintaxis de la herencia (I)

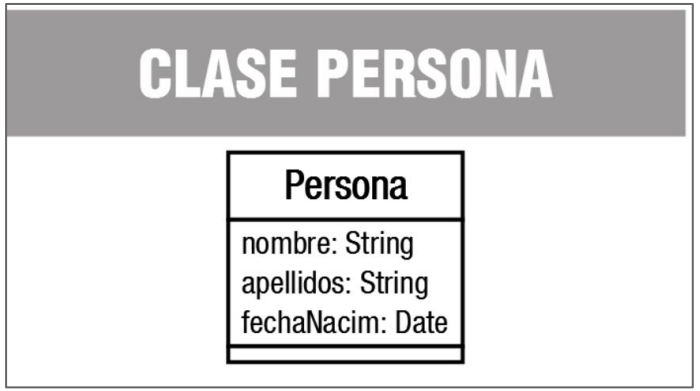
- En Java la herencia se indica mediante la palabra reservada extends

```
[modificador] class ClasePadre {
        // Cuerpo de la clase
}

[modificador] class ClaseHija extends ClasePadre {
        // Cuerpo de la clase
}
```

Sintaxis de la herencia (II)

Ejemplo: clase padre **Persona** y clase **Alumno** que hereda de ella

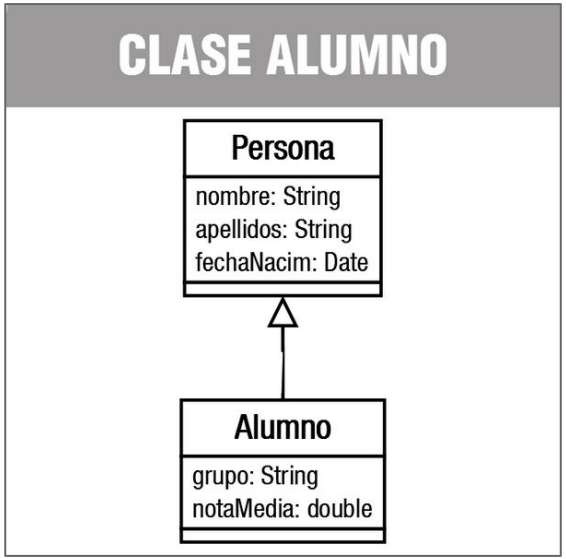
```
public class Persona {
        String nombre;
        String apellidos;
        LocalDate fechaNacim;

}
```



Sintaxis de la herencia (III)

Un objeto de la clase **Alumno** contendrá los atributos grupo y notaMedia (propios de la clase **Alumno**) y también nombre, apellidos y fechaNacim heredados de la clase **Persona**

```
public class Alumno extends Persona {
        String grupo;
        double notaMedia;

}
```



Acceso a miembros heredados (I)

- No es posible acceder a miembros privados de una superclase.
- Para poder acceder a ellos se pueden hacer públicos pero entonces se daría acceso a cualquier clase y puede no desearse.
- Para ello existe el modificador protected (protegido) que permite el acceso desde clases heredadas, pero no desde otras clases fuera de la jerarquía

Acceso a miembros heredados (II)

What is the difference between public, protected, package-private and private in Java?

	Modificadores			
Ámbito	**Private**	**Default**	**Protected**	**Public**
Proyecto	X	X	X	✓
Clase o Subclase de un paquete diferente	X	X	✓	✓
Subclase del mismo paquete	X	✓	✓	✓
La propia clase	✓	✓	✓	✓

Acceso a miembros heredados (III)

- Ejemplo: al definir los atributos de la clase **Persona** como private la clase **Alumno** que hereda de **Persona** no tiene acceso a los mismos

```
public class Persona {
    private String nombre;
    private String apellidos;
    private LocalDate fechaNacim;
}
```

Acceso a miembros heredados (y IV)

- Si por otro lado se usa el modificador **protected** , las clases que heredan pueden tener acceso. Sin modificador las clases hijas y las del mismo paquete tendrían también acceso

```
public class Persona {
    protected String nombre;
    protected String apellidos;
    protected LocalDate fechaNacim;
}
```

```
public class Persona {
    String nombre;
    String apellidos;
    LocalDate fechaNacim;
}
```

Utilización de miembros heredados (I)

- Los atributos/campos heredados por una clase son iguales que aquellos que sean definidos específicamente en la nueva clase derivada.
- Ejemplo: la clase **Persona** dispone de tres atributos y la clase **Alumno** que hereda de ella añadía dos atributos más.
- Se puede considerar que la clase **Alumno** tiene cinco atributos: tres por ser **Persona** (nombre, apellidos, fecha de nacimiento) y otros dos más por ser **Alumno** (grupo y nota media).

Utilización de miembros heredados (y II)

- Igual que los atributos/campos, también se heredan los métodos siendo parte también de los métodos de la clase derivada junto a los definidos específicamente en la subclase
- **Ejemplo**: si en la clase **Persona** hay métodos get y set para cada uno de sus tres atributos (nombre, apellidos, fechaNacim) se tendrían 6 métodos que podrían ser heredados por sus clases derivadas.
- La clase **Alumno** derivada de Persona tendría diez métodos:
  - Seis por ser Persona (getNombre, getApellidos, getFechaNacim, setNombre, setApellidos, setFechaNacim).
  - Cuatro más por ser Alumno (getGrupo, setGrupo, getNotaMedia, setNotaMedia).

Redefinición de métodos heredados (I)

- Una clase puede redefinir algunos de los métodos que ha heredado de su clase base. En ese caso, el nuevo método sustituye al heredado. Esto también se conoce como sobrescritura de métodos.
- Aunque un método sea sobrescrito o redefinido aún es posible acceder a él a través de la referencia super, aunque sólo se podrá acceder a métodos de la clase padre y no a métodos de clases superiores en la jerarquía de herencia.
- Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la que ofrezca el método original de la superclase, pero nunca restringirla.
  - Por ejemplo, si un método es declarado como protected o de paquete en la clase base, podría ser redefinido como public en una clase derivada.
- Los métodos estáticos o de clase no pueden ser sobrescritos. Permanecen inalterables a través de toda la jerarquía de herencia.

Redefinición de métodos heredados (y II)

```
public class Persona {
	protected String nombre;
	protected String apellidos;
	public String getNombreCompleto() {
		return nombre + " " + apellidos;
	}
}
```

```
public class Alumno extends Persona {
	@Override
	public String getNombreCompleto() {
		return "El nombre del alumno es " + nombre + " " + apellidos;
	}
}
```

```
public class Alumno extends Persona {
	@Override
	public String getNombreCompleto() {
		return "El nombre del alumno es " + super.getNombreCompleto();
	}
}
```

Ampliación de métodos heredados (I)

- A veces es necesario ampliar el comportamiento de un método de la superclase en lugar de reemplazarlo por completo.
- La palabra reservada "super" es una referencia a la clase padre de la clase actual y permite invocar métodos de la superclase desde la subclase.
- Al utilizar "super" en un método de la subclase, se puede invocar el método correspondiente de la superclase, permitiendo preservar el comportamiento antiguo y añadir el nuevo.
- Ejemplo: en una clase **Alumno** que hereda de **Persona** se invoca desde su código el método de la superclase para mostrar la información de **Persona** y luego agregar la información específica del alumno.

Ampliación de métodos heredados (II)

```
public class Persona {
	public void mostrar () {
		System.out.printf ("Nombre: %s\n", this.nombre);
		System.out.printf ("Apellidos: %s\n", this.apellidos);
		System.out.printf ("Fecha de nacimiento: %02d/%02d/%04d%n",
			this.fechaNacim.getDayOfMonth(), this.fechaNacim.getMonthValue(),
			this.fechaNacim.getYear() );
		}
}
public class Alumno extends Persona {
	@Override
	public void mostrar () {
		super.mostrar (); // Llamada al método "mostrar" de la superclase
		// A continuación mostramos la información "especializada" de esta subclase
		System.out.printf ("Grupo: %s\n", this.grupo);
		System.out.printf ("Nota media: %5.2f\n", this.notaMedia);
	}
}
```

Ampliación de métodos heredados (y III)

```
public class Profesor extends Persona {
	private String especialidad;
	private float salario;

	@Override
	public void mostrar () {
		super.mostrar (); // Llamada al método "mostrar" de la superclase

		// A continuación mostramos la información "especializada" de esta subclase
		System.out.printf ("Especialidad: %s\n", this.especialidad);
		System.out.printf ("Salario: %7.2f euros\n", this.salario);

	}
}
```

Constructores y herencia (I)

- Ya hemos visto en la UT5 que en una clase, un constructor puede llamar a otro constructor de la misma clase utilizando la referencia **this**.
- La llamada a **this** solo puede hacerse en la primera línea del constructor.
- Un constructor de una clase derivada puede llamar al constructor de su clase base utilizando la palabra clave **super**. Esto permite inicializar los atributos heredados antes de los específicos de la clase derivada.

Constructores y herencia (II)

- Si no se incluye una llamada explícita a **super()** dentro del constructor de la clase derivada, el compilador la añade automáticamente.
- Esto resulta en una llamada en cadena de constructores de la superclase hasta llegar a la clase **Object** en la jerarquía de Java.
- El constructor por defecto, creado por el compilador si no se ha escrito ninguno incluye una llamada a **super()** como primera acción. Esta llamada asegura que los constructores de las superclases se ejecuten correctamente antes de la inicialización de los atributos de la clase derivada.

Constructores y herencia (y III)

Ejemplo

```
public class Persona {
	public Persona (String nombre, String apellidos, LocalDate fechaNacim) {
	    this.nombe= nombre;
	    this.apellidos= apellidos;
	    this.fechaNacim= LocalDate.of(fechaNacim.getYear(),  fechaNacim.getMonthValue(),
	            fechaNacim.getDayOfMonth());
	}
}

public class Alumno {
	public Alumno (String nombre, String apellidos, LocalDate fechaNacim, String grupo,
	double notaMedia) {
	        super (nombre, apellidos, fechaNacim);
	        this.grupo= grupo;
	        this.notaMedia= notaMedia;
	}
}
```

La clase Object en Java (I)

Todas las clases en Java son descendentes de la clase Object.

Esta clase define los estados y comportamientos básicos que deben tener todos los objetos.

Entre estos comportamientos, se encuentran:

- La posibilidad de compararse.
- La capacidad de convertirse a cadenas.
- La habilidad de devolver la clase del objeto.

La clase Object en Java (y II)

Entre los métodos que incorpora la clase Object y que por tanto hereda cualquier clase en Java están:

Método	Descripción
`Object ()`	Constructor.
`clone ()`	Método **clonador**: crea y devuelve una copia del objeto ("clona" el objeto).
`boolean equals (Object obj)`	Indica si el objeto pasado como parámetro es igual a este objeto.
`void finalize ()`	Método llamado por el **recolector de basura** cuando éste considera que no queda ninguna referencia a este objeto en el entorno de ejecución.
`int hashCode ()`	Devuelve un código **hash** para el objeto.
`toString ()`	Devuelve una representación del objeto en forma de `String`.

Herencia múltiple

- La herencia múltiple permite heredar los miembros de múltiples clases para formar una nueva clase derivada
- La herencia múltiple puede causar ambigüedades cuando hay miembros con el mismo identificador en clases base diferentes.¿Qué miembro se hereda en caso de ambigüedad?
- La herencia múltiple no está disponible en Java debido a la complejidad y los problemas de diseño que puede causar
- En su lugar, se fomenta el uso de interfaces para lograr la funcionalidad similar a la herencia múltiple en Java

CLASES ABSTRACTAS

Concepto de clase Abstracta (I)

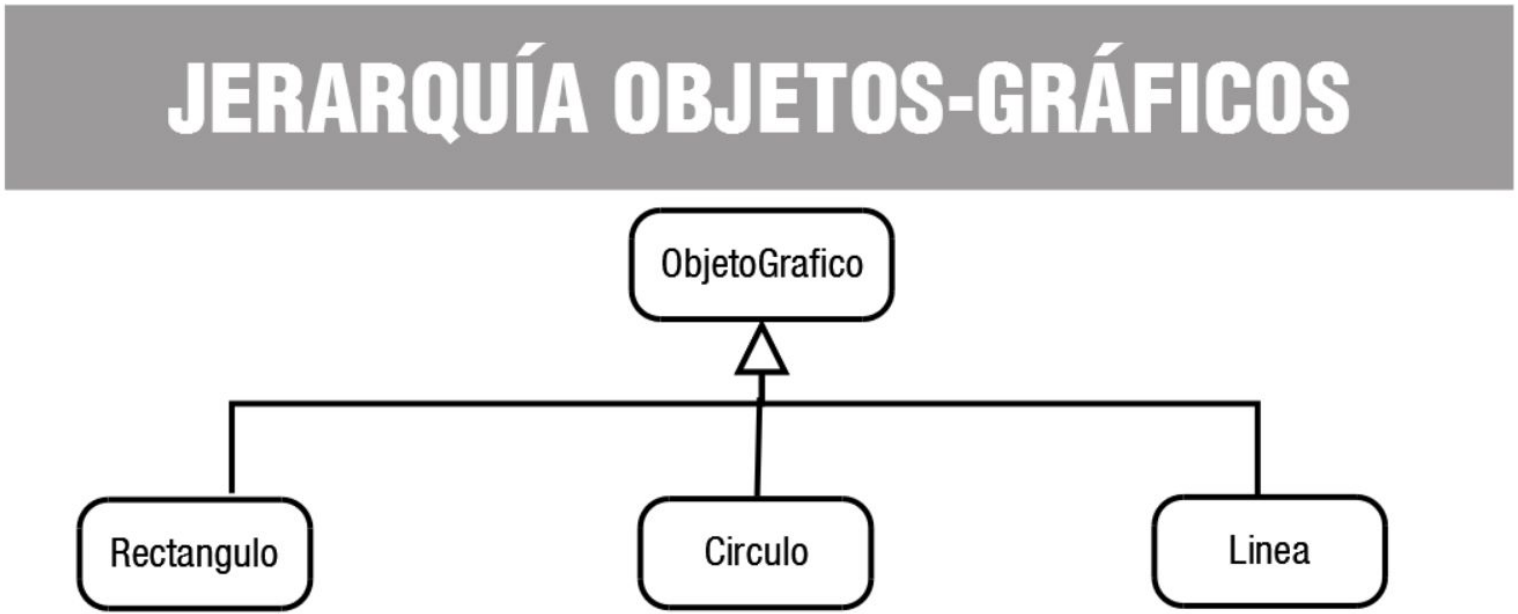
- Las clases abstractas representan conceptos demasiado abstractos para tener instancias concretas.
- Sirven como marco o modelo para clases derivadas dentro de una jerarquía de herencia.
- Su utilidad es proporcionar líneas generales de cómo es una clase sin implementar todos sus métodos.
- Son útiles cuando las clases derivadas deben proporcionar los mismos métodos pero con implementaciones específicas.

Concepto de clase Abstracta (II)

Ejemplos:

- En entornos de manipulación de objetos gráficos (donde objetos como líneas, círculos, y rectángulos comparten atributos y métodos comunes) una clase abstracta "objeto gráfico" puede definir atributos comunes y algunos métodos genéricos, dejando otros métodos sin implementar.
- En un centro educativo que utilice las clases de ejemplo Alumno y Profesor, ambas subclases de Persona. La clase Persona es útil como clase base para construir otras clases que hereden de ella, pero no como una clase instanciable de la cual vayan a existir objetos

Concepto de clase Abstracta (y III)



Declaración de una clase abstracta (I)

- Una clase abstracta es una clase que no se puede instanciar, es decir, que no se pueden crear objetos a partir de ella.
- La idea es permitir que otras clases deriven de ella, proporcionando un modelo genérico y algunos métodos de utilidad general.
- Las clases abstractas se declaran mediante el modificador **abstract**

```
[modificador_acceso] abstract class nombreClase [herencia] [interfaces] {

}
```

Declaración de una clase abstracta (II)

- Una clase puede contener en su interior métodos declarados como **abstract**. Para estos métodos sólo se indica la cabecera pero no se proporciona su implementación.
- En el caso anterior la clase tendrá que ser forzosamente también abstract y los métodos tendrán que ser posteriormente implementados en sus clases derivadas.
- Una clase abstracta puede contener métodos totalmente implementados (no abstractos), los cuales se heredan en las subclases y pueden utilizarse con normalidad

Declaración de una clase abstracta (y III)

Trabajando con clases abstractas hay que tener en cuenta:

- Una clase abstracta sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un new de una clase abstracta pues se produciría un error de compilación.
- Una clase abstracta puede contener métodos totalmente definidos (no abstractos) y métodos sin definir (métodos abstractos).

```
public abstact class Persona {
      protected String nombre;
      protected String apellidos;
      protected LocalDate fechaNacim;
}
```

Métodos abstractos (I)

- Un método abstracto es un método cuya implementación no se define, sino que se declara únicamente su interfaz (cabecera) para que su cuerpo sea implementado más adelante en una clase derivada.
- Un método se declara como abstracto mediante el uso del modificador **abstract** (como en las clases abstractas)

[modificador_acceso] **abstract** <tipo> <nombreMetodo> ([parámetros]) [excepciones];

Métodos abstractos (II)

- Los métodos abstractos tienen que ser obligatoriamente definidos en las clases derivadas.
- Si en una clase derivada se deja algún método abstracto sin implementar, esa clase derivada será también una clase abstracta. O lo que es lo mismo: cuando una clase contiene un método abstracto tiene que declararse como abstracta obligatoriamente.

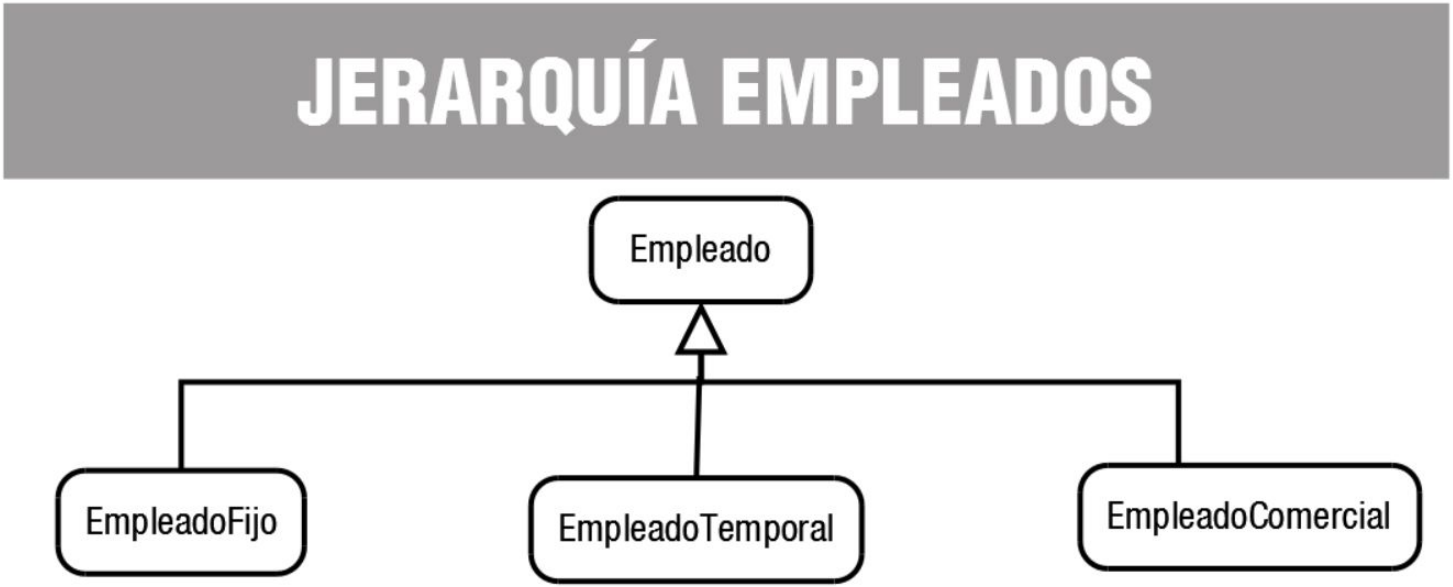
```
public abstract class Persona {
    protected abstract void mostrar ();
}
```

Métodos abstractos (III) - Ejemplo (I)

Ejemplo:

- Hay una clase **Empleado** abstracta con tres subclases **EmpleadoFijo**, **EmpleadoTemporal** y **EmpleadoComercial**.
- La clase **Empleado** contiene un método abstracto llamado **calcularNomina**.
- Este método abstracto es esencial para cualquier tipo de empleado, ya que todo empleado cobra una nómina.
- Cada clase especializada derivada de **Empleado** implementa de manera específica el cálculo de la nómina ya que difiere según el tipo de empleado: fijo, temporal o comercial

Métodos abstractos (IV) - Ejemplo (y II)



Métodos abstractos (y V)

- Un método abstracto implica que la clase a la que pertenece tiene que ser abstracta, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.
- Un método abstracto no puede ser privado (no se podría implementar, dado que las clases derivadas no tendrían acceso a él).
- Los métodos abstractos no pueden ser estáticos, pues los métodos estáticos no pueden ser redefinidos (y los métodos abstractos necesitan ser redefinidos).

Clases final

- Una clase declarada como final no puede ser heredada, es decir, no puede tener clases derivadas.
- La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas)

[modificador_acceso] final class nombreClase [herencia] [interfaces]
public **final** class Alumno { }

MÉTODOS FINAL

- Un método también puede ser declarado como final, cuyo caso no podrá ser redefinido en una clase derivada

[modificador_acceso] final <tipo> <nombreMetodo> ([parámetros]) [excepciones]
public **final** void mostrar ();

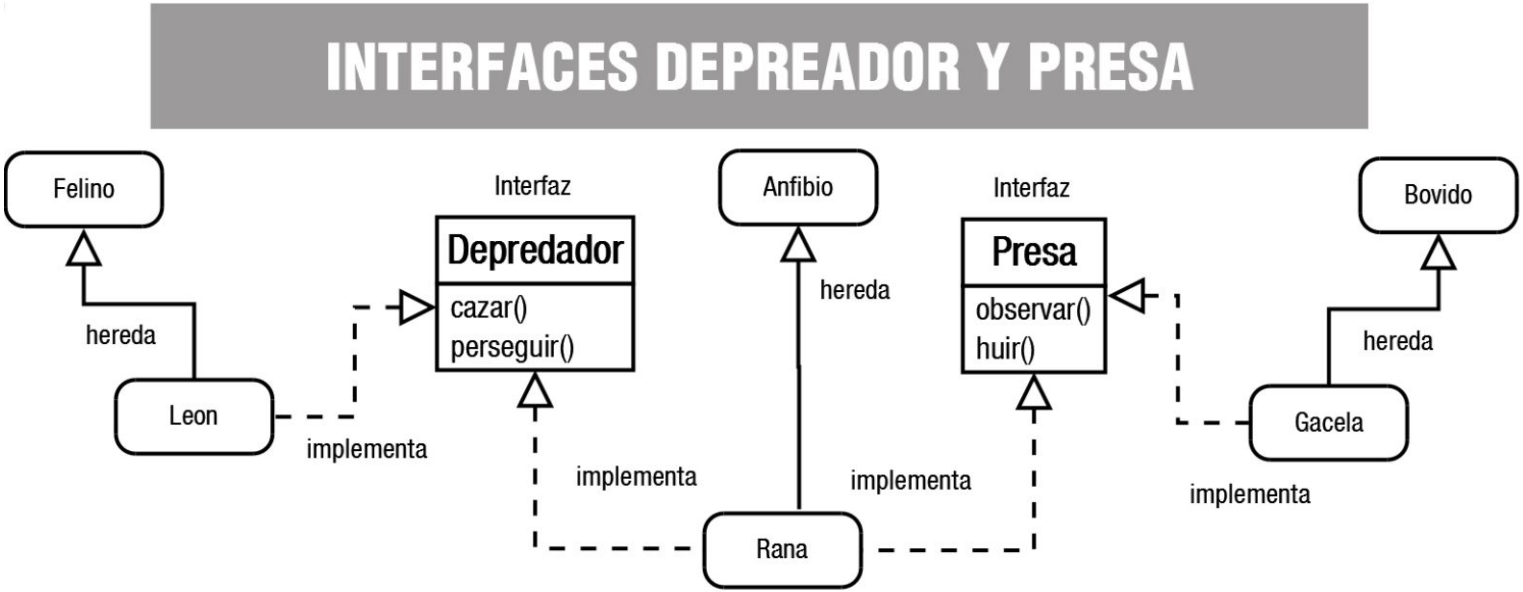
Interfaces

Concepto de Interfaz (I)

- La idea de una interfaz es disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener los objetos que formen parte de una clasificación no necesariamente jerárquica
- Una interfaz consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento.
- Una interfaz define una relación "de implementación de comportamientos". Por ejemplo: la clase A implementa los métodos establecidos en la interfaz B, o los comportamientos indicados por B son llevados a cabo por A; pero no significa que A sea de clase B

Concepto de Interfaz (II)

Ejemplo:



Concepto de Interfaz (III)

- Una interfaz en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.
- Una interfaz solamente indica la forma del método, pero no la implementación
- Sirven para indicar acciones o capacidades que el objeto debe ser capaz de realizar. Por esto el nombre de muchas interfaces en Java termina con sufijos del tipo "-able", "-or", "-ente" y similares, que significan capacidad o habilidad para hacer o ser receptores de algo
- Ejemplos de nombres de interfaz: Configurable, Serializable, Modificable, Clonable, Executable, Administrator, Server…
- **Una clase puede implementar varias interfaces.**

Concepto de Interfaz (y IV)

Ejemplo:

- La clase **Coche** es una subclase de **Vehículo**.
- Los coches, al ser vehículos a motor, requieren acciones específicas como arrancar y detener el motor.
- Se define una interfaz llamada **Arrancable** que define los métodos típicos de un objeto a motor, como **arrancar** y **detener**.
- La clase **Coche**, siendo subclase de **Vehículo**, implementaría los comportamientos definidos en la interfaz **Arrancable**.
- Otras clases como **Motocicleta** o **Motosierra** también podrían implementar la interfaz **Arrancable**, cada una con su propia implementación de los métodos.

Clase abstracta o interfaz (I)

- Las interfaces y las clases abstractas comparten similitudes en su uso para definir contratos de comportamiento en Java.
- Las diferencias principales son:
  - Una clase puede implementar múltiples interfaces pero solo heredar de una clase abstracta.
  - Las interfaces solo pueden declarar métodos, mientras que las clases abstractas pueden tener métodos implementados.
  - Las interfaces permiten establecer comportamientos comunes entre clases sin necesidad de herencia directa.
  - Las interfaces tienen su propia jerarquía independiente de la jerarquía de clases.

Clase abstracta o interfaz (y II)

- Las interfaces permiten compartir comportamientos entre clases no relacionadas sin imponer una relación de herencia.
- Proporcionan una forma flexible de definir contratos de comportamiento común.
- Es recomendable utilizar interfaces cuando solo se necesitan métodos abstractos sin implementación.
- Si se requieren implementaciones parciales o completas de métodos se pueden usar clases abstractas.

Definición de interfaces (I)

La declaración de una interfaz en Java es parecida a la de una clase aunque con las siguientes diferencias

- Se utiliza la palabra reservada **interface** en lugar de class.
- Puede utilizarse el modificador **public**. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo .java en el que se encuentra
- Si no se indica el modificador **public**, el acceso será por omisión el "de paquete"
- Todos los miembros de la interfaz (atributos y métodos) son public de manera implícita. No es necesario indicar el modificador **public**, aunque puede hacerse.
- Todos los atributos son de tipo final y public (tampoco es necesario especificarlo). Es decir, constantes y públicos por lo que hay que darles un valor inicial.
- Todos los métodos son abstractos también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

Definición de interfaces (y II)

```
[public] interface <NombreInterfaz> {
        [public] [final] <tipo1> <atributo1>= <valor1>;
        [public] [final] <tipo2> <atributo2>= <valor2>;

        [public] [abstract] <tipo_devuelto1>  <nombreMetodo1> ([lista_parámetros]);
        [public] [abstract] <tipo_devuelto2>  <nombreMetodo2> ([lista_parámetros]);
}
```

```
public interface Depredador {
   int MAX_PRESAS = 100;
   void localizar (Animal presa);
   void cazar (Animal presa);
}
```

Implementación de interfaces (I)

- Para que una clase implemente una interfaz se utiliza la palabra reservada **implements**

```
class NombreClase implements NombreInterfaz { }
```

- Es posible indicar varios nombres de interfaces separándolos por comas

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2,... { }
```

Implementación de interfaces (II)

- Cuando una clase implementa una interfaz, tiene que redefinir sus métodos nuevamente con acceso público. Con otro tipo de acceso se producirá un error de compilación. Del mismo modo que no se podían restringir permisos de acceso en la herencia de clases, tampoco se puede hacer en la implementación de interfaces.
- Una vez implementada una interfaz en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro método, sin ninguna diferencia, pudiendo ser invocados, heredados, redefinidos, etc.

Implementación de interfaces (y III)

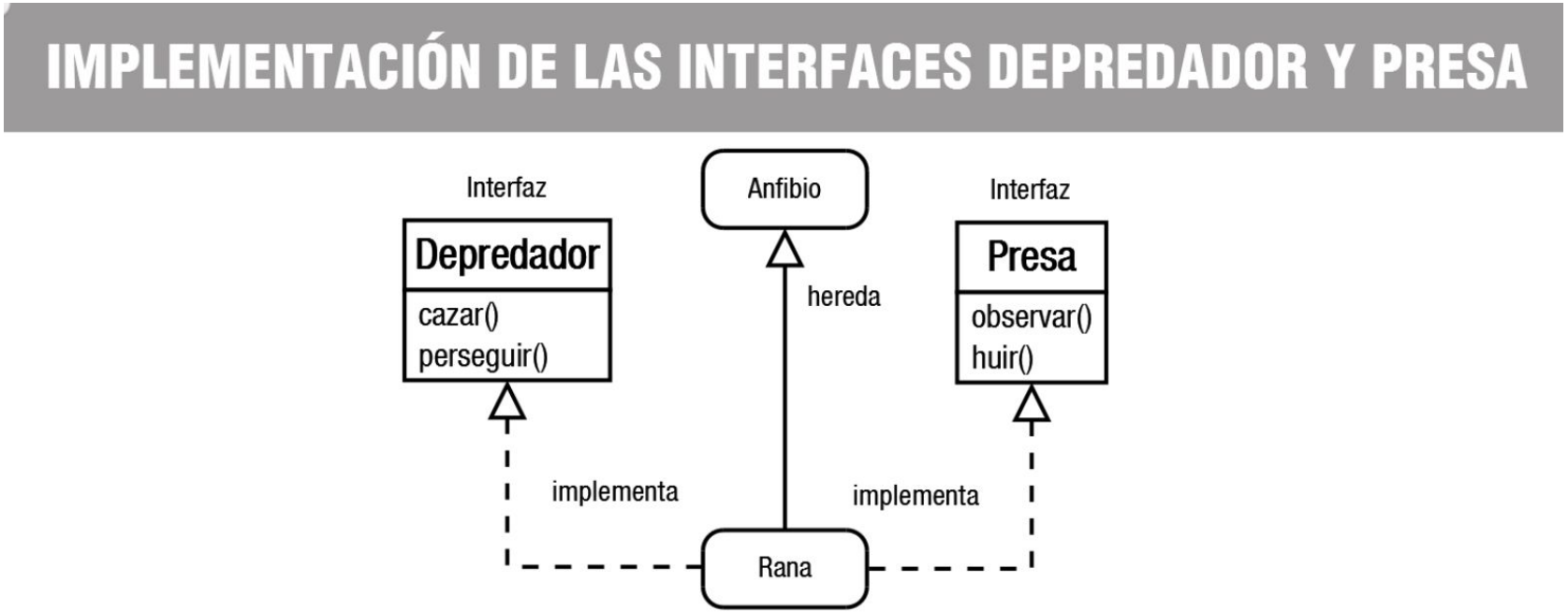
Ejemplo

```
class Leon implements Depredador {
    void localizar (Animal presa) {
        // Implementación del método localizar para un león
        ...
    }

    void cazar (Animal presa) {
        // Implementación del método cazar para un león
        ...
    }
}
```

Simulación de herencia múltiple con interfaces (I)

- Java no permite la herencia múltiple de clases, pero es posible simularla utilizando interfaces.
- Se pueden definir interfaces separadas (A, B, C) que representan diferentes comportamientos que una clase (X) desea heredar.
- Al implementar múltiples interfaces la clase adquiere los contratos de comportamiento definidos en ellas.
- Ejemplo: La clase **Rana** puede implementar las interfaces **Depredador** y **Presa** para obtener comportamientos específicos. Si se necesitan más comportamientos se pueden añadir nuevas interfaces que la clase Rana puede implementar.

Simulación de herencia múltiple con interfaces (II)



Simulación de herencia múltiple con interfaces (y III)

Con el mecanismo "una herencia pero varias interfaces" podrían conseguirse resultados similares a los obtenidos con la herencia múltiple.

Ahora bien puede darse el problema de la colisión de nombres al implementar dos interfaces que tengan un método con el mismo identificador. En tal caso puede suceder lo siguiente:

- Si los dos métodos tienen diferentes parámetros no habrá problema aunque tengan el mismo nombre pues se realiza una sobrecarga de métodos.
- Si los dos métodos tienen un valor de retorno de un tipo diferente, se producirá un error de compilación (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).
- Si los dos métodos son exactamente iguales en nombre, parámetros y tipo devuelto, entonces solamente se podrá implementar uno de los dos métodos. En realidad se trata de un solo método pues ambos tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

Herencia de Interfaces

- Las interfaces también permiten la herencia entre sí. Para indicar que una interfaz hereda de otra se indica nuevamente con la palabra reservada **extends**.
- En este caso sí se permite la herencia múltiple de interfaces. Si se hereda de más de una interfaz se indica con la lista de interfaces separadas por comas

```
public interface InterfazUno {
        // Métodos y constantes de la interfaz Uno
}
public interface InterfazDos {
        // Métodos y constantes de la interfaz Dos
}
public interface InterfazCompleja extends InterfazUno, InterfazDos {
        // Métodos y constantes de la interfaz compleja
}
```

Polimorfismo

Concepto de polimorfismo (I)

- El polimorfismo permite manipular objetos de clases diferentes pero relacionadas de manera uniforme a través de su superclase.
- Ofrece flexibilidad al trabajar con objetos de distintas subclases sin la necesidad de conocer su tipo exacto durante la compilación.
- Se logra mediante la definición de métodos en la superclase y su redefinición en las subclases, lo que permite que un objeto de la superclase se comporte según su subclase durante la ejecución.
- Facilita el desarrollo de programas genéricos que pueden trabajar con diferentes tipos de objetos sin necesidad de saber sus tipos específicos.

Concepto de polimorfismo (II)

- El polimorfismo permite que una referencia a un objeto de la superclase pueda asumir la forma de una referencia a un objeto de cualquiera de sus subclases durante la ejecución del programa.
- Simplifica el diseño y la implementación de programas al permitir tratar con objetos de manera más abstracta.
- Facilita la creación de código reutilizable y modular al fomentar la interoperabilidad entre clases relacionadas.
- El polimorfismo puede llevarse a cabo tanto con superclases (abstractas o no) como con interfaces.

Concepto de polimorfismo (y III)

```
// Declaración de una referencia a un objeto de tipo X
ClaseX obj;  // Objeto de tipo X (superclase)

// Zona del programa donde se instancia un objeto de tipo A (subclase) y se le asigna a la referencia obj. La variable obj adquiere la forma de la subclase A.
obj = ClaseA();

// Aquí se instancia un objeto de tipo B (subclase) y se le asigna a la referencia obj.
// La variable obj adquiere la forma de la subclase B.
obj = ClaseB();

// Zona donde se utiliza el método m sin saber realmente qué subclase se está utilizando. (Sólo se sabrá durante la ejecución del programa)

obj.m () // Llamada al método m (sin saber si será el método m de A o de B).
```

Enlace dinámico (I)

- El enlace se refiere a la conexión que ocurre durante una llamada a un método en un programa. En el contexto de la programación orientada a objetos, existen dos formas principales de enlace: estático y dinámico.
- El enlace estático:
  - Se lleva a cabo durante el proceso de compilación.
  - En los lenguajes no orientados a objetos, es la única forma de resolver el enlace.
  - La versión del método a invocar se determina en tiempo de compilación, basada en el tipo declarado de la referencia.

Enlace dinámico (y II)

El enlace dinámico:

- También se llama vinculación tardía o enlace tardío.
- En los lenguajes orientados a objetos, como Java, es posible gracias al polimorfismo.
- La versión del método a invocar se determina en tiempo de ejecución, basada en el tipo real del objeto al que apunta la referencia, obtenido durante la ejecución del programa.
- Permite que el método invocado sea el definido en la subclase del objeto, no en la clase de la referencia.

Polimorfismo y enlace dinámico

- El polimorfismo depende del enlace dinámico para su funcionamiento.
- Permite tratar objetos de diferentes subclases de manera uniforme, invocando el método adecuado según el tipo de objeto en tiempo de ejecución.
- En el ejemplo de una clase X y sus subclases A y B, el enlace dinámico es esencial para determinar qué versión del método m debe ser invocada, ya que esta resolución solo puede hacerse durante la ejecución del programa.

Limitaciones del enlace dinámico (I)

El polimorfismo en la programación orientada a objetos permite utilizar referencias de tipos más generales (superclases) para apuntar a objetos de tipos más específicos (subclases). Sin embargo, existe una importante restricción en el uso de esta capacidad.

- No es posible acceder directamente a los miembros específicos de una subclase a través de una referencia de superclase.
- Se pueden utilizar únicamente los miembros declarados en la superclase, incluso si la referencia apunta a un objeto de la subclase en tiempo de ejecución.

Limitaciones del enlace dinámico (y II)

- Si tienes una clase B que es subclase de A y declaras una variable como referencia a un objeto de tipo A, solo podrás acceder a los miembros heredados de A, incluso si la variable hace referencia a un objeto de tipo B (subclase).
- En el ejemplo de las clases **Persona**, **Profesor** y **Alumno**, el polimorfismo permite declarar variables de tipo **Persona** y referenciar objetos de tipo **Profesor** o **Alumno**.
- Sin embargo, solo se pueden acceder a los métodos que se sabe que existen en la superclase **Persona**, independientemente de si la referencia apunta a un objeto de tipo **Profesor** o **Alumno**.

Interfaces y polimorfismo (I)

- El polimorfismo puede lograrse utilizando interfaces. En este enfoque, un objeto puede tener una referencia cuyo tipo sea una interfaz. Sin embargo, para que el compilador lo permita, la clase cuyo constructor se utilice para crear el objeto debe implementar esa interfaz ya sea directamente o a través de alguna superclase.
- Un objeto cuya referencia sea de tipo interfaz solo puede utilizar los métodos definidos en la interfaz; no puede acceder a los atributos y métodos específicos de su clase.
- Las referencias de tipo interfaz permiten estandarizar la forma de interactuar con objetos que implementan la misma interfaz, independientemente de sus clases concretas.

Interfaces y polimorfismo (II)

Ejemplo 1

- Si tienes una variable de referencia de tipo interfaz "**Arrancable**", podrías instanciar objetos de tipo **Coche** o **Motosierra** y asignarlos a esa referencia, incluso si las clases no tienen una relación de herencia directa.
- Sin embargo, solo podrás utilizar los métodos y atributos definidos en la interfaz "**Arrancable**" en ambos casos, y no los métodos específicos de las clases **Coche** o **Motosierra**.

Interfaces y polimorfismo (III)

Ejemplo 2

- Con el ejemplo de las clases Persona, Alumno y Profesor se podría declarar variables del tipo de una interfaz común, por ejemplo, "**Imprimible**", para referenciar objetos de las clases **Persona**, **Alumno** o **Profesor** que implementen esta interfaz.
- Esto permite una gestión uniforme de objetos de clases distintas pero que comparten una funcionalidad común definida en la interfaz.

Interfaces y polimorfismo (IV)

```
public interface Imprimible {
	String devolverContenidoString();
}

public class Persona  implements Imprimible{
	protected String nombre;
	protected String apellidos;

	@Override
	public String devolverContenidoString() {
		return String.format("Nombre:%s, apellidos:%s", nombre, apellidos);
	}
}
```

Interfaces y polimorfismo (V)

```
public class Alumno extends Persona {
	String grupo;
	double notaMedia;

	@Override
	public String devolverContenidoString() {
	return String.format("%s,grupo:%s, notaMedia:%f", super.devolverContenidoString(),
		grupo, notaMedia);
	}
}
```

Interfaces y polimorfismo (VI)

```
public class Profesor extends Persona  {

        private String especialidad;
        private float salario;

        @Override
        public String devolverContenidoString() {
        return String.format("%s,especialidad:%s, salario:%f", super.devolverContenidoString(),
                especialidad, salario);
        }
}
```

Interfaces y polimorfismo (y VII)

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

        Imprimible  imprimible = new Persona();
        System.out.println(imprimible.devolverContenidoString());

        imprimible = new Alumno();
        System.out.println(imprimible.devolverContenidoString());

        imprimible = new Profesor();
        System.out.println(imprimible.devolverContenidoString());

}
```

Conversión de objetos (I)

- No se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase
- Si quieres tener acceso a todos los métodos y atributos específicos del objeto subclase tendrás que realizar una conversión explícita (casting) que convierta la referencia de la superclase a la del objeto de subclase.
- Para realizar conversiones entre distintas clases es obligatorio que exista una relación de herencia entre ellas
- Se realizará una conversión implícita o automática de subclase a superclase siempre que sea necesario, pues un objeto de tipo subclase siempre contendrá toda la información necesaria para ser considerado un objeto de la superclase.

Conversión de objetos (II)

- La conversión en sentido contrario (de superclase a subclase) debe hacerse de forma explícita y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una excepción de tipo ClassCastException.

```
class ClaseA {
        public int atrib1;
}
class ClaseB extends ClaseA {
        public int atrib2;
}
```

Conversión de objetos (III)

```
ClaseA obj;  // Referencia a objetos de la ClaseA
obj= new ClaseB ();
// Referencia a objeto ClaseA, pero apunta realmente a objeto ClaseB (polimorfismo)
```

- El objeto que se crea como instancia de **ClaseB** (subclase de **ClaseA**) contiene más información que la que la referencia obj te permite en principio acceder sin que el compilador genere un error (pues es de **ClaseA**).
- En concreto los objetos de **ClaseB** disponen de atrib1 y atrib2, mientras que los objetos de la ClaseA sólo de atrib1.
- Para acceder a esa información adicional de la clase especializada (atrib2) tendrás que realizar una conversión explícita (casting)

Conversión de objetos (IV)

```
// Casting de ClaseA a ClaseB (funcionará porque el objeto es realmente de ClaseB)
System.out.printf ("obj.atrib2=%d\n", ((ClaseB) obj).atrib2);
```

Sin embargo si se hubiera tratado de una instancia de **ClaseA** y hubieras intentado acceder al miembro atrib2, se habría producido una excepción de tipo ClassCastException:

Conversión de objetos (y V)

```
ClaseA obj;  // Referencia a objetos de ClaseA
obj= new ClaseA ();
// Referencia a objetos de ClaseA, y apunta realmente a un objeto de ClaseA

// Casting del tipo ClaseA al tipo ClaseB (puede dar problemas porque el objeto es realmente del tipo ClaseA):
// Funciona (ClaseA tiene atrib1)
System.out.printf ("obj.atrib2=%d\n", ((ClaseB) obj).atrib1);

// ¡Error en ejecución! (ClaseA no tiene atrib2). Producirá una ClassCastException.
System.out.printf ("obj.atrib2=%d\n", ((ClaseB) obj).atrib2);
```