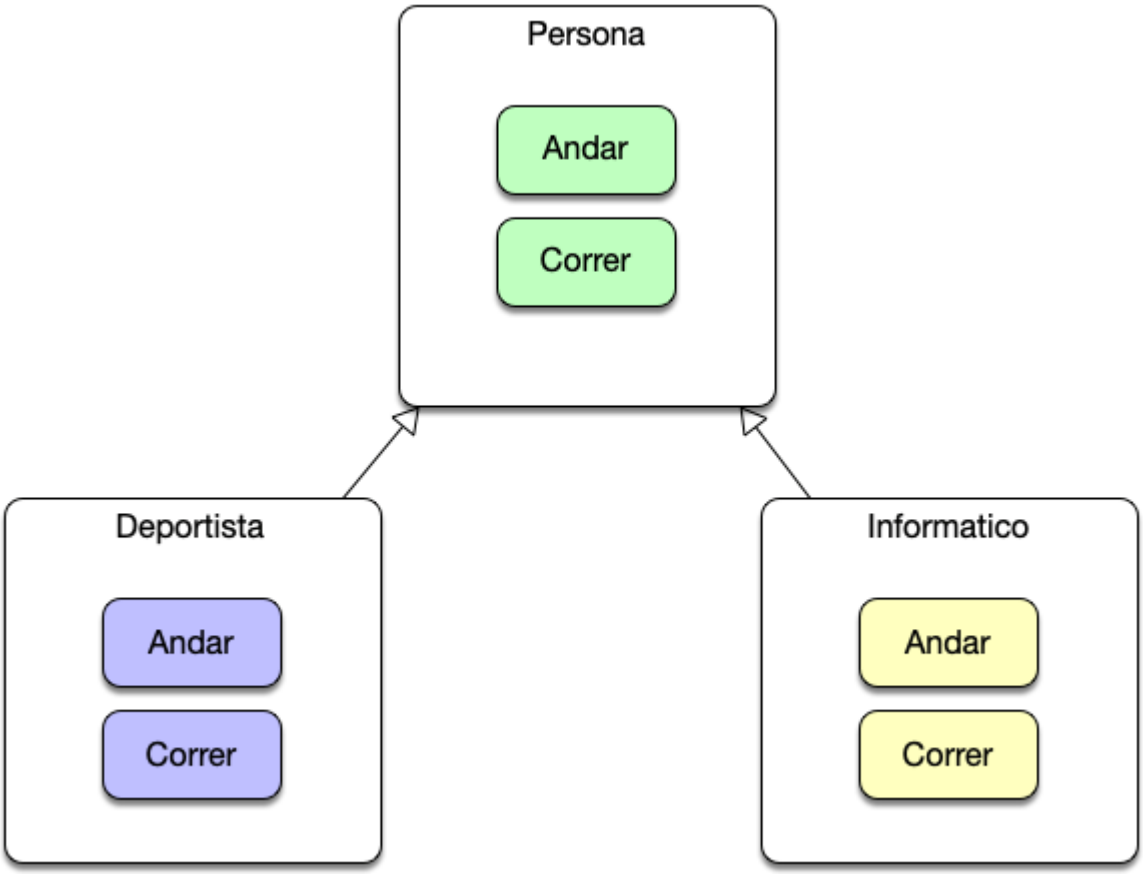
Tabla de Contenidos

- ¿Qué es una clase abstracta?
  - Ejemplo con Persona, Deportista e Informático
  - Subclase Deportista
  - Subclase Informático
  - Ejemplo completo
- Java Clases Abstractas y su uso
- Conclusión
- Otros artículos relacionados:
  - Otros Enlaces de interes:
  - Java interfaces

Java Clases Abstractas y como usarlas . En Java, las clases abstractas son una herramienta muy útil cuando queremos que nuestras clases compartan algunos comportamientos, pero dejando que cada una implemente esos comportamientos de manera distinta. Hoy vamos a ver un ejemplo práctico usando una clase abstracta llamada Persona, y dos subclases, Deportista e Informático, para entender mejor cómo funciona esto.



¿Qué es una clase abstracta?

Una clase abstracta es una clase que no puedes instanciar directamente. Es decir, no puedes crear un objeto a partir de ella, pero sí puedes usarla para definir un “esqueleto” de lo que otras clases deben hacer. Este esqueleto puede tener métodos abstractos (que las subclases deben implementar) y métodos con comportamiento ya definido.

Ejemplo con Persona, Deportista e Informático

Imaginemos que tenemos una clase abstracta Persona, y que todas las personas pueden andar y correr, pero cómo lo hacen depende de quién sea. El método para andar y el de correr no van a ser iguales para un deportista que para un informático, ¿verdad? Así que vamos a dejar esos métodos como abstractos, y haremos que las subclases los implementen a su manera.

```
abstract class Persona {
    // Método abstracto para andar (las subclases deben implementarlo)
    abstract void andar();
    // Método abstracto para correr
    abstract void correr();
}
```

Aquí hemos definido nuestra clase abstracta Persona. Los métodos andar y correr son abstractos, lo que significa que las subclases, como Deportista e Informático, tendrán que decirnos cómo deben comportarse.

Subclase Deportista

Un Deportista, como es lógico, tiene su propio estilo tanto para andar como para correr, y vamos a implementarlo.

```
class Deportista extends Persona {
    @Override
    void andar() {
        System.out.println("El deportista anda rápido.");
    }
    @Override
    void correr() {
        System.out.println("El deportista corre a gran velocidad.");
    }
}
```

Aquí, el deportista camina rápido y, por supuesto, corre aún más rápido.

Subclase Informático

Por otro lado, un Informático probablemente tenga un estilo de vida más sedentario, y esto también se refleja en cómo camina y corre.

```
class Informatico extends Persona {
    @Override
    void andar() {
        System.out.println("El informático anda lentamente.");
    }
    @Override
    void correr() {
        System.out.println("El informático prefiere no correr.");
    }
}
```

Como puedes ver, en el caso del informático, prefiere andar lento y correr, bueno, no es lo suyo.

Ejemplo completo

Ahora que tenemos nuestras subclases implementadas, veamos cómo se comportan cuando las utilizamos.

```
public class Main {
    public static void main(String[] args) {
        Persona deportista = new Deportista();
        deportista.andar();    // Output: El deportista anda rápido.
        deportista.correr();   // Output: El deportista corre a gran
velocidad.

        Persona informatico = new Informatico();
        informatico.andar();   // Output: El informático anda
lentamente.
        informatico.correr();  // Output: El informático prefiere no
correr.
```

```
    }
}
```

```
El deportista anda rápido.
El deportista corre a gran velocidad.
El informático anda lentamente.
El informático prefiere no correr.
```

Como ves, las dos subclases comparten los mismos métodos, pero cada una los implementa de manera distinta. Esto es exactamente lo que nos permite hacer una clase abstracta: definir un contrato (los métodos que deben estar) y dejar que cada subclase decida cómo cumplirlo.

Java Clases Abstractas y su uso

Las clases abstractas son súper útiles cuando:

1. Quieras asegurarte de que todas las subclases compartan algunos comportamientos: En nuestro ejemplo, todas las personas tienen que andar y correr, pero cómo lo hacen es diferente.
2. Necesites que cada subclase implemente esos comportamientos a su manera: En este caso, un Deportista y un Informático andan y corren de formas distintas, pero ambos deben hacerlo.
3. Evitar repetir código: Si varias clases tienen algo en común, puedes definirlo en una clase abstracta para no tener que repetirlo en cada subclase.
   Un ejemplo de evitar repetición de código sería definir nombre y apellidos a nivel de la clase:

```
abstract class Persona {
    // Atributos comunes a todas las personas
    private String nombre;
    private String apellidos;
```

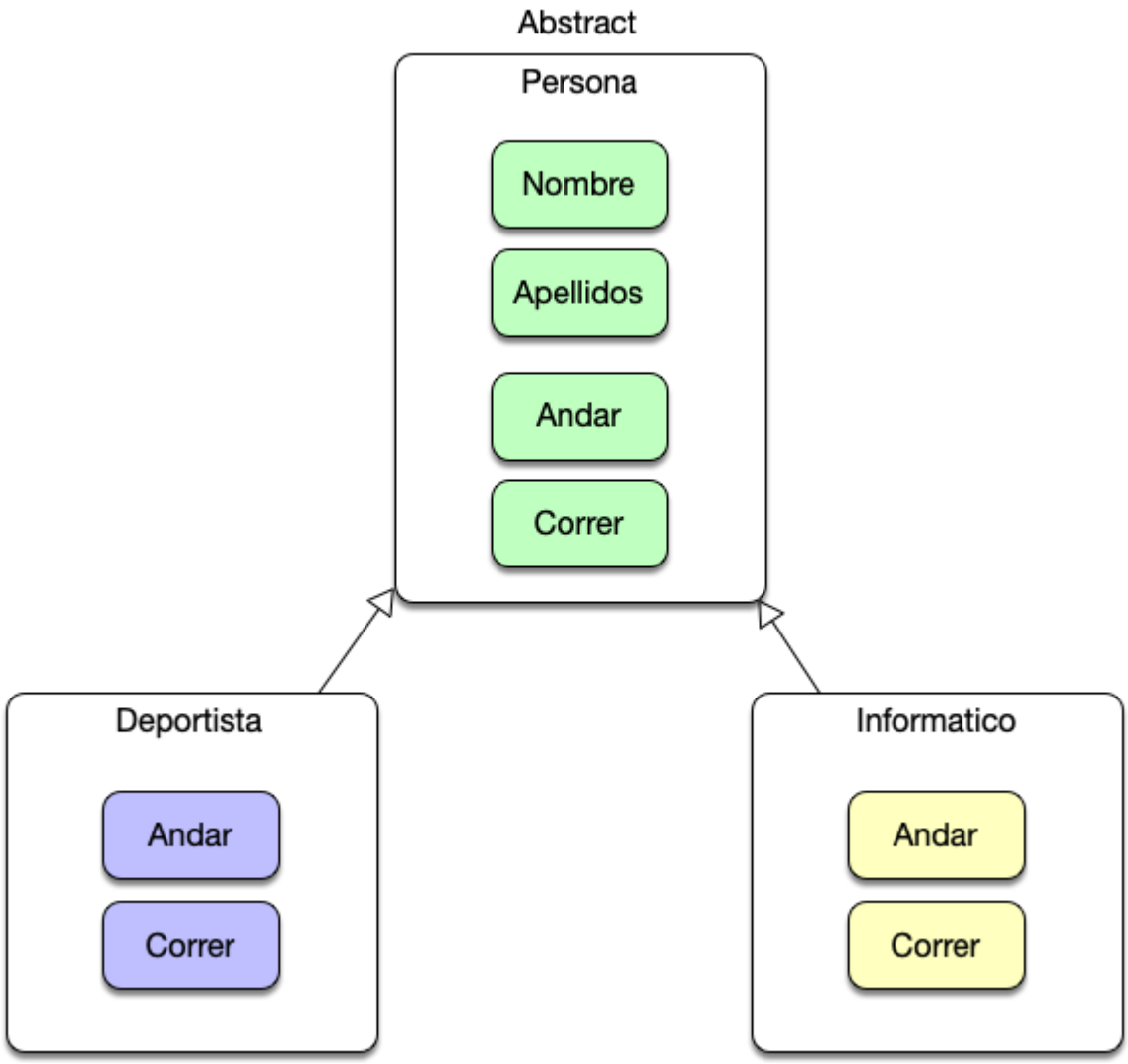
```
    // Constructor para inicializar nombre y apellidos
    public Persona(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
    // Métodos abstractos que las subclases deben implementar
    abstract void andar();
    abstract void correr();
    // Getters y Setters
    public String getNombre() {
        return nombre;
    }

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

    public String getApellidos() {
        return apellidos;
    }

    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
}
```

Las clases hijas los heredarían automaticamente.



Conclusión

Las clases abstractas en Java son una excelente herramienta cuando necesitas definir comportamientos comunes, pero que puedan ser personalizados por cada subclase. En este ejemplo, vimos cómo una clase abstracta Persona obliga a que sus subclases, Deportista e Informático, implementen a su manera los métodos andar y correr.

Con esta estructura, puedes asegurarte de que todas las subclases cumplan con ciertos requisitos, pero sin perder flexibilidad a la hora de definir cómo deben hacerlo. ¡Y eso es lo

que hace que el diseño de clases en Java sea tan flexible!

Otros artículos relacionados:

- Java Polimorfismo, herencia simplicidad y encapsulación.
- Clases Abstractas vs Interfaces
- Java Herencia Multiple y sus opciones
- Patron Factory. ¿Que es? y ¿Como implementarlo?
- REST Nested Resources y su utilidad

Otros Enlaces de interes:

Java interfaces