Unidad 3. Java Orientado a Objetos

Contenido

1 Actividad inicial 2
1.1 Introducción. 2
2 Lenguaje Java orientado a objetos 3
2.1 Pilares de la programación orientada a objetos 5
2.1.1 Abstraction 6
2.1.2 Encapsulación 7
2.1.3 Herencia 8
2.1.4 Polimorfismo 9
2.2 Java como lenguaje orientado a objetos. 9
2.2.1 El operador this, referencia a Objeto vs referencia a clase . 14
2.2.2 Creación de objetos 16
2.2.3 Propiedades públicas frente a privadas. Encapsulación. ... 18
2.2.4 Métodos. Encapsulación II. Privado vs Público. 20
2.2.5 Métodos de acceso. 24
2.2.6 Metodos y clases estáticas. 29
2.3 Herencia en Java 34
2.4 Jerarquía de clases 36
2.4.1 Notas Cornell 42
3 Actividad Independiente. Jerarquía educación 42
3.1 Polimorfismo 43
3.2 Sobrecarga. 43
3.3 Actividad guiada Overriding 46
3.3.1 Clases abstractas 47
3.4 Actividad independiente. Overriding 52
3.5 Interfaces. Actividad guiada 52
3.6 Actividad de refuerzo. Modelo de clases 58
4 Clases wrap para tipos primitivos. Conversión de tipos 59
5 Clases e interfaces con tipos genéricos en Java 62
5.1 Interfaces genéricos 67
6 Interfaces funcionales y lambdas. Tipos complejos en Java 69
6.1 Clases anónimas 71
6.2 Clases anónimas a partir de una Clase. Constructores 71
6.2.1 Clases anónimas para interfaces. 73
6.3 Interfaces funcionales 75
6.4 Expresiones Lambda. 80
6.5 Interfaces predefinidos en Java 85
6.5.1 Composición de funciones 85
6.5.2 Interfaz funcional de predicado 86
6.5.3 El interface Consumer 90
6.5.4 Supplier Interface 94
6.5.5 El interface funcional Function 97
6.5.6 The Unary Operator Interface 100
6.5.7 Interfaz funcional Bifunction 102

6.5.8 Interfaces funcionales para tipos primitivos 103
7 Programación funcional 106
7.1 Categorización de paradigmas de programación 106
7.2 Programación funcional 106
7.3 Functional Programming Advantages 107
7.4 Programación funcional Java 8 109
7.4.1 Conceptos básicos de la programación funcional 109
7.5 Las funciones son objetos de primera clase / miembros de primera clase del lenguaje 110
7.6 Funciones puras, sin estado. 114
7.6.1 Función de orden superior 116
7.6.2 Sin estado 120
7.6.3 Sin efectos secundarios 121
7.6.4 Objetos inmutables 122
7.6.5 Favorecer la recursividad sobre los bucles 124
9 Bibliografía y referencias web 134

1 Actividad inicial

Realizamos una pregunta sobre el mundo real de los alumnos. Pedimos a los alumnos que características físicas o de otro tipo han heredado de sus padres y abuelos, y que las vayan apuntando en el bloc de notas. Preguntamos también que comportamientos han heredado de sus padres y abuelos y que lo apunten

Herencia de padres y abuelos de los alumnos	
Características	Comportamientos
Ojos marrones (padre)	Gusto por la música (abuelo)
Pelo rizado (abuelo)	Calmado (abuelo)

Cuando se explique herencia podremos realizar una analogía con esta tabla

1.1 Introducción.

En este **capítulo vamos a explicar los diferentes patrones o acercamientos que usaremos para resolver nuestros problemas de programación en programación orientada a objetos**. Empezaremos por los patrones más sencillos e iremos avanzando hacia patrones más complicados.

Recordamos antes los pilares de la programación orientada a objeto que son cuatro:

1. **Abstracción: abstraer en orientación a objetos es representar la realidad a través de nuestras clases.** Representamos de manera precisa los detalles necesarios sobre un contexto real, omitiendo el resto.
2. **Herencia: permitimos crear clases heredando de otras**. De esta manera hay código ya escrito, lo heredamos, y lo utilizamos en las nuevas clases sin volver a escribirlo.
3. **Encapsulacion**: cuando creamos clases en nuestros programas ofrecemos para cada clase unos métodos que son servicios. **El funcionamiento interior de esa clase es irrelevante para las clases** que **usan ese servicio**. Por ejemplo, para **arrancar mi coche no necesito saber como esta compuesto el sistema eléctrico**, sólo **la llave para arrancarlo y girarla**. Ese es el servicio que me ofrecería una clase SistemaDeArranque.
4. **Polimorfismo**: tiene que ver como su etimología indica con múltiples formas, múltiples comportamientos. En **herencia vamos a tener una clase principal y múltiples subclases**. La **clase principal o el interfaz ofrece un comportamiento común que las subclases hijas** modifican **adaptándose a sus propias características.**

2 Lenguaje Java orientado a objetos

La programación orientada a objetos se basa en cómo, en el mundo real, los objetos a menudo se componen de muchos tipos de objetos más pequeños. Esta capacidad de combinar objetos, sin embargo, es sólo un aspecto muy general de la programación orientada a objetos. La programación orientada a objetos proporciona varios otros conceptos y características para que la creación y el uso de objetos sean más fáciles y flexibles, y la más importante de estas características es la de las clases.

Una *clase* es una plantilla para varios objetos con características similares. Las clases incorporan todas las características de un conjunto determinado de objetos. Cuando se escribe un programa en un lenguaje orientado a objetos, no se definen objetos reales. Las clases de objetos se definen.

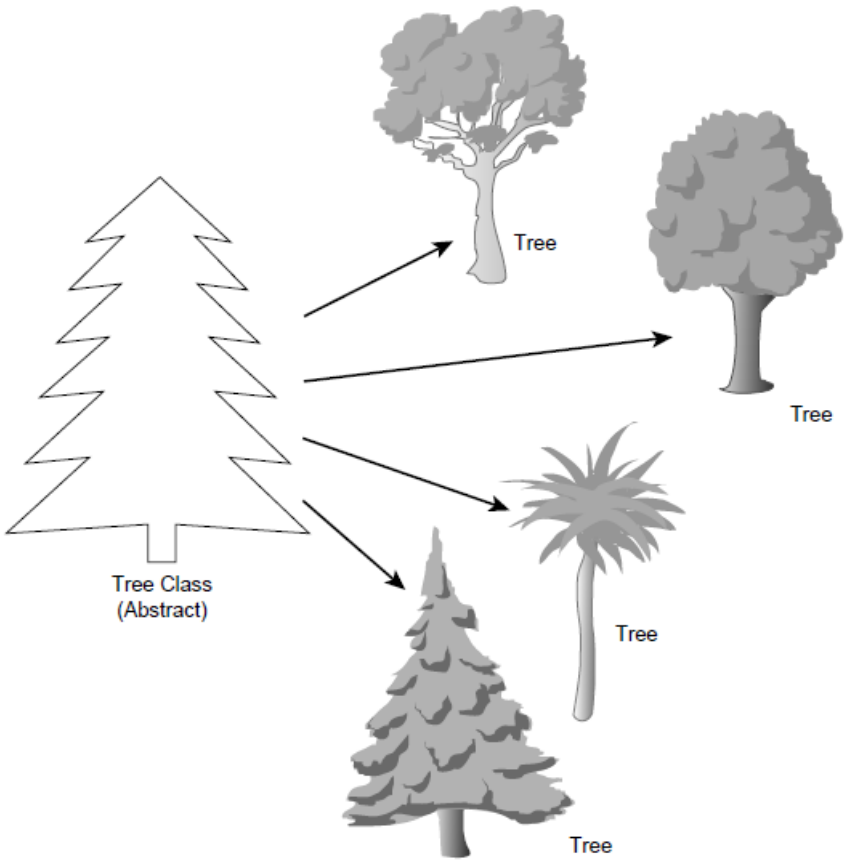
Por ejemplo, podría tener una clase **Tree** que describa las características de

todos los árboles (tiene hojas y raíces, crece, crea clorofila). La clase **Tree** sirve como un modelo abstracto para el concepto de árbol: para alcanzar y agarrar, o interactuar con, o cortar un árbol, debe tener una instancia concreta de ese árbol. Por supuesto, una vez que tenga una clase de árbol, puede crear muchas instancias diferentes de ese árbol, y cada instancia de árbol diferente puede tener diferentes características (hojas cortas, altas, tupidas, gotas en otoño), mientras sigue comportándose como y siendo inmediatamente reconocible como un árbol.

Una instancia de una clase es otra palabra para un objeto real. Si las clases son una representación abstracta de un objeto, una instancia es su representación concreta. Entonces, ¿cuál es exactamente la diferencia entre una instancia y un objeto? Nada, en realidad. Objeto es el término más general, pero tanto las instancias como los objetos son la representación concreta de una clase.

De hecho, los términos instancian y objeto a menudo se usan indistintamente en el lenguaje de POO (Programación orientada a objetos). Un instancia de un árbol y un objeto de árbol son la misma cosa. En un ejemplo más cercano al tipo de cosas que podría querer hacer en la programación Java, puede crear una clase para el elemento de la interfaz de usuario denominada botón.

La clase Button define las características de un botón (su etiqueta, su tamaño, su apariencia) y cómo se comporta (¿necesita un solo clic o un doble clic para activarlo, cambia de color cuando se hace clic, qué hace cuando se activa?). Una vez definida la clase Button, puede crear fácilmente instancias de ese botón (es decir, objetos de botón) que todos toman las características básicas del botón según lo definido por la clase, pero que pueden tener diferentes apariencias y comportamientos en función de lo que desea que haga ese botón en particular. Al crear una clase, no tiene que seguir reescribiendo el código para cada botón individual que desee usar en el programa, y puede reutilizar la clase **Button** para crear diferentes tipos de botones a medida que los necesite en este programa y en otros programas.



2.1 Pilares de la programación orientada a objetos

La programación orientada a objetos se basa en cuatro pilares, conceptos que la diferencian de otros paradigmas de programación. Estos pilares, algunos principios que daremos a conocer en unidades posteriores, reglas y convenciones para la programación orientada a objetos se originan a partir de principios de los años 90.
Las compañías de software y los ingenieros de software buscan código de alta calidad para ofrecer mejores productos de software y reducir el precio del mantenimiento y la fabricación.

En esa búsqueda, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides escribieron un libro ***Design Patterns: Elements of Reusable Object-Oriented Software***. Ha sido influyente en el campo de la ingeniería de software y es considerado como una fuente importante para la teoría y la práctica del diseño orientado a objetos. Se han vendido más de 500.000 ejemplares en inglés y en otros 13 idiomas.

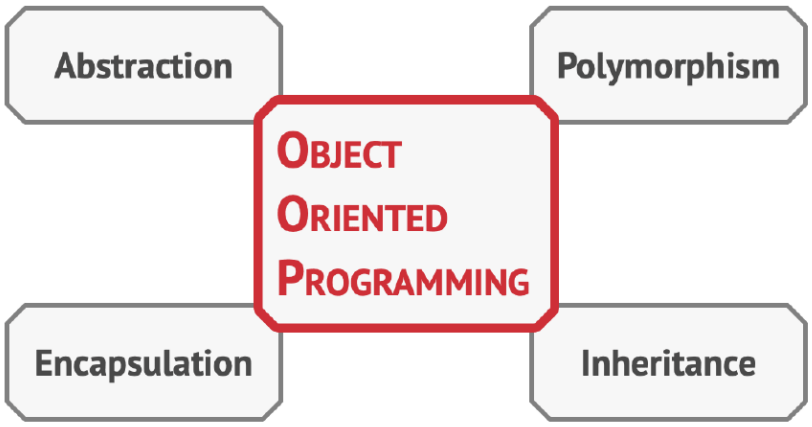
Asimismo, en los años 90 se celebraron muchas conferencias sobre ingeniería

de software, para mejorar la producción de software y sistemas de información. Seguiremos tantas de estas "reglas" que la mayoría de los desarrolladores y académicos reconocidos y admirados en Ciencias de la Computación ha fijado desde ese período.

Los conceptos de OOP nos permiten crear interacciones específicas entre objetos Java. Permiten reutilizar el código sin crear riesgos de seguridad ni hacer que un programa Java sea menos legible.

Aquí están los cuatro principios principales con más detalle.

1. abstracción
2. encapsulación
3. herencia
4. polimorfismo



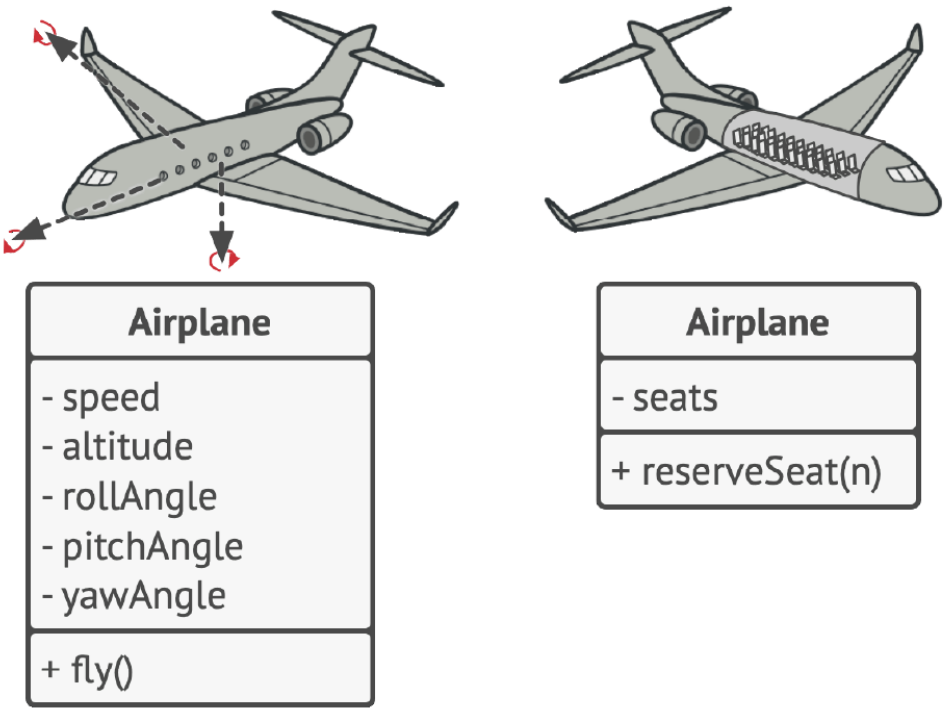
2.1.1 Abstraction

La abstracción es un modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los detalles relevantes para este contexto con alta precisión y omite todo lo **demás.**

La mayoría de las veces, cuando se crea un programa con POO, se da forma a los objetos del programa en función de objetos del mundo real. Sin embargo, los objetos del programa no representan los originales con una precisión del 100% (y rara vez se requiere que lo hagan). En su lugar, los objetos solo *modelan* atributos y comportamientos de objetos reales en un contexto específico,

omitiendo el resto.

Por ejemplo, una clase Airplane probablemente podría existir tanto en un simulador de vuelo como en una aplicación de reserva de vuelos. Pero en el primer caso, albergaría detalles relacionados con el vuelo real, mientras que en la segunda clase solo le importaría el mapa de asientos y qué asientos están disponibles.



2.1.2 Encapsulación

Para arrancar el motor de un coche, sólo tiene que girar una tecla o pulsar un botón. No es necesario conectar cables bajo el capó, girar el cigüeñal y los cilindros, e iniciar el ciclo de potencia del motor. Estos detalles se esconden bajo el capó del coche. Solo tienes una interfaz simple: un interruptor de arranque, un volante y algunos pedales. Esto ilustra cómo cada objeto tiene una interfaz, una parte pública de un objeto, abierta a interacciones con otros objetos.

La encapsulación es la característica de cada módulo en nuestro código para ocultar datos u operaciones que no son necesarias para revelar a otros módulos de nuestro código. Por ejemplo, suponiendo que tengo una llamada de módulo StoreEmployees que almacena datos de empleados. Otros módulos de nuestra aplicación, como windows (la interfaz gráfica) utilizarán este módulo para

guardar datos. Sin embargo, esta interfaz gráfica no necesita saber si StoreEmployees guarda los datos en un archivo o una base de datos. Esos detalles de implementación están ocultos, encapsulados. Por lo tanto, el Window sólo necesita una función storeEmployeeData(Employee data) para utilizar esta funcionalidad.

La encapsulación en la programación orientada a objetos es la capacidad de un objeto para ocultar partes de su estado y comportamientos de otros objetos, exponiendo sólo una interfaz limitada al resto del programa.

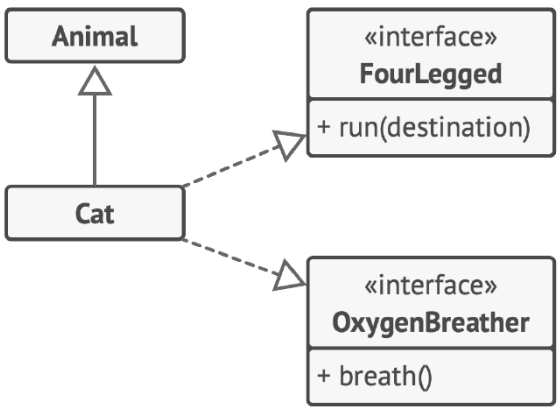
Encapsular algo significa hacerlo private y, por lo tanto, accesible solo desde dentro de los métodos de su propia clase.

Hay un modo un poco menos restrictivo llamado protected que hace que un miembro de una clase también esté disponible para las subclases. Las interfaces y las clases/métodos abstractos de la mayoría de los lenguajes de programación se basan en los conceptos de abstracción y encapsulación.

2.1.3 Herencia

La herencia es la capacidad de construir nuevas clases sobre las existentes. La principal ventaja de la herencia es la reutilización de código. Si se desea crear una clase que es ligeramente diferente de una existente, no es necesario duplicar el código. En su lugar, se extiende la clase existente y se coloca la funcionalidad adicional en una subclase resultante, que hereda los campos y métodos de la superclase.

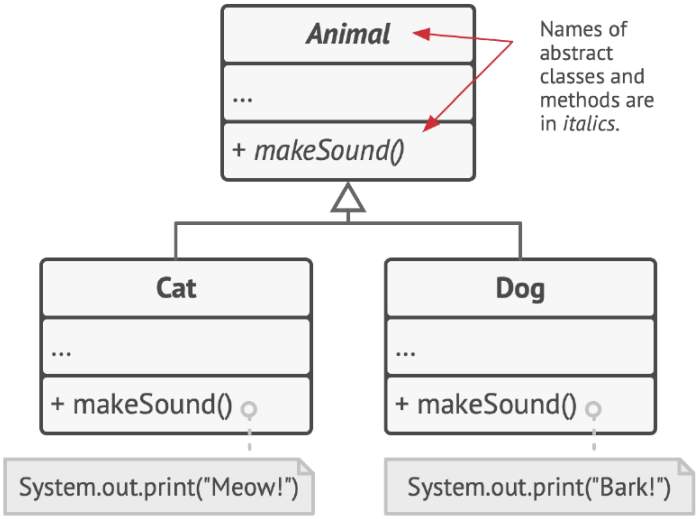
La consecuencia de utilizar la herencia es que las subclases tienen la misma interfaz que su clase primaria. No se puede ocultar un método en una subclase si se declaró en la superclase. Tú también debe implementar todos los métodos abstractos, incluso si no tienen sentido para la subclase.



2.1.4 Polimorfismo

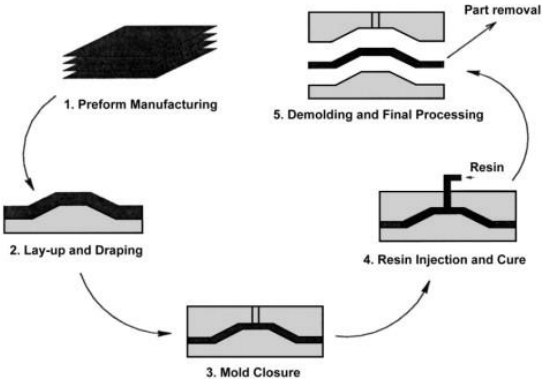
La palabra polimorfismo proviene de las palabras griegas para "muchas formas". Un método polimórfico, por ejemplo, es un método que puede tener diferentes formas, donde "forma" se puede considerar como tipo o comportamiento.

Veamos algunos ejemplos de animales. La mayoría de los animales pueden hacer sonidos. Podemos anticipar que todas las subclases necesitarán reemplazar el método makeSound base para que cada subclase pueda emitir el sonido correcto; por lo tanto, podemos declararlo *abstracto* de inmediato. Esto nos permite omitir cualquier implementación predeterminada del método en la superclase, pero forzar a todas las subclases a crear las suyas propias.



2.2 Java como lenguaje orientado a objetos.

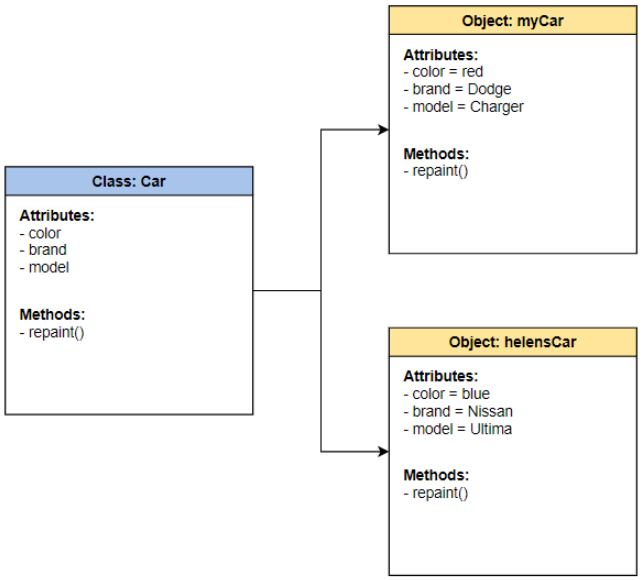
Java es orientado a objetos en su naturaleza. Significa que los programas se componen de objetos. Sin embargo, para crear o crear objetos, necesitaría una plantilla. Una analogía para describir clases y objetos podría hacerse de la fabricación. Cuando se necesita construir cien mil pomos de puertas para automóviles, debe seguir algunos pasos. En primer lugar, se debe diseñar el trabajo de carrocería del coche. Luego construyes un molde para esa pieza. Y por último, se deber ir a producción en masa.



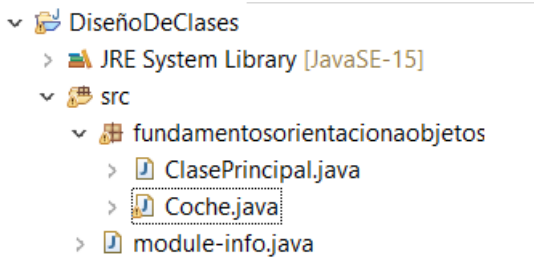
Para programar en un lenguaje orientado a objetos, se necesita su molde, su plantilla para crear un conjunto de objetos. Por ejemplo, digamos que creamos una clase, `Car`, para contener todas las propiedades que debe tener un automóvil, `color`, `marca` y `modelo`. A continuación, cree una instancia de un objeto de tipo `Car`, `myCar` para representar mi coche específico.

A continuación, podríamos establecer el valor de las propiedades definidas en la clase para describir mi coche, sin afectar a otros objetos o la plantilla de clase. A continuación, podemos reutilizar esta clase para representar cualquier número de coches.

Ya describimos esta acción. Es el primer pilar de la Programacion Orientada a Objetos. *La abstracción* es un modelo de un objeto o fenómeno del mundo real. Hemos abstraído el coche del mundo real, en una representación de codificación llamada clase Car. Dado que estamos representando el coche a través de software, agregamos las propiedades y los comportamientos que necesitamos controlar en nuestro programa. Suponiendo que estamos vendiendo coches nuevos, no necesitamos almacenar el cuentakilómetros, que señalan el número de kilómetros de nuestro coche. Hemos tomado la decisión de dejar fuera de nuestra representación esta característica.



Vamos a crear este proyecto DiseñoDeClases en Eclipse. Incluiremos estas dos clases, pero primero agregaremos la clase principañ ClasePrincipal.java.




```
package oppfundamentals;

public class ClasePrincipal {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

		System.out.println("Hello there. Howdy. My first java program on
the running");

	}

}
```

Cada archivo de código fuente en Java contiene menos una clase con el modificador public. Por lo tanto, para programar en Java necesitamos agregar clases a nuestro código. Para explicar el objeto y las clases lo primero que haremos es añadir uno a este proyecto. Por favor, agregue el coche de clase al proyecto y copie el código siguiente en el archivo.

```
package fundamentosorientacionaobjetos;
public class Coche {
	private String color;
	private String marca;
	private String modelo;

	public Coche (String color, String marca, String modelo) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;

	}

	public void repintar(String color) {


		this.color=color;
	}

	@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}



}
```

Vamos a desglosar la clase en cada elemento diferente. Resaltado en amarillo, puede encontrar propiedades o atributos de **clase**. Están destinados a almacenar información sobre cada coche individual. Esta información es lo que llamamos el estado del objeto. Por ejemplo, suponiendo que el coche es rojo, su estado actual es ser rojo. Si repintamos el coche cambiamos su estado. Las propiedades de clase se comportan como variables que estudiamos en la Unidad 1 y en este documento. Tienen un tipo y un nombre. En este ejemplo, el tipo es String.

```
	private String color;
	private String marca;
	private String modelo;
```

En el ejemplo anterior, la clase tenemos tres propiedades: color, marca y modelo. Propiedades como se puede ver, describe el coche. Esto es menos de la mitad de un objeto. La otra mitad es lo que el objeto puede hacer, su comportamiento. El coche en nuestro programa puede ser repintado. Para dotar a los objetos de comportamiento, que son las acciones que pueden realizar, en las clases podemos definir métodos. Se puede inferir que los métodos son similares a las subrutinas. Tiene razón, son casi lo mismo. Sin embargo, estas funciones pertenecen a objetos y sólo se pueden aplicar a través de objetos.

La primera función que se encuentra normalmente en una clase es el constructor. Este es el método más singular porque permite a los programadores crear objetos como explicaremos más adelante. Es muy particular ya que no tiene un tipo de valor devuelto. El tipo de valor devuelto es implícito, un objeto de esta clase.

```
public Car (String color, String marca, String modelo) {

                this.color=color;
                this.marca=marca;
                this.modelo = modelo;

      }
```

```
public  NO DEVOLVER TIPO AQUÍ Car (String color, String marca, String modelo)
{
```

Después del constructor, los programadores suelen escribir métodos para agregar una operación o un comportamiento al objeto. Para agregar operaciones a una clase usamos métodos. Recuerde que los métodos pueden ser procedimientos o funciones. El método de repintado permite que el coche sea repintado. Cambia el comportamiento del coche. Esta es una característica intrínseca al coche. Una persona no puede ser repintada, ni el mar.

```
public void repintar (String color) {

            this.color=color;
```

```
	}
```

Por último, el método toString() devuelve una descripción del contenido del objeto en string. Java utiliza el método toString para poder imprimir el método en la salida. De lo contrario, imprimiría una dirección de memoria, como explicamos más adelante.

```
@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

En general, hay algunos métodos que permiten crear objetos, otros nos permiten hacer acciones, cálculos o devolver datos procesados. Sin embargo, estamos hablando de objetos, pero no tenemos uno. Tenemos un plano, un molde, pero aún no hemos creado ni objetado.

2.2.1 El operador this, referencia a Objeto vs referencia a clase

Cuando se codifica dentro de una clase, tenemos a nuestra disposición una herramienta, o una clave reservada para acceder al objeto en cualquier lugar. Se denomina operador this. Esto funciona de forma similar a una variable declarada como car, Car myCar. Ir proporciona acceso a todas las propiedades y métodos del objeto. En realidad, señala el objeto que ha creado para esta clase.

```
public abstract class Car {
	protected String color;
	private String marca;
	private String modelo;

	public  Car (String color, String marca, String modelo) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
```

This.color hace referencia a la propiedad protected String color;. Al hacerlo, puede distinguir entre el parámetro Color de Tipo String y la propiedad protected

String color; Debes tener en cuenta que el operador this solo se puede utilizar dentro del código de clase. Además, el operador this hace referencia a un objeto que se ha creado, no a la clase. No funciona a menos que se haya creado un objeto. No podemos usarlo en métodos estáticos.

Para hacer referencia a una clase, debe escribir el nombre de la clase, normalmente seguido de una constante estática o un método estático. Cambie la última línea del constructor Coche, agregando la referencia de clase al método estático. Hace que el código sea más legible y limpio.

```
Coche.incrementarNumeroDeCoches();

 public  Car (String color, String brand, String model{

                    this.color=color;
                    this.brand=brand;
                    this.model = model;


                    Coche.incrementarNumeroDeCoches();

       }
```

Coche.java

```
package fundamentosorientacionaobjetos;
public class Coche {
       private String color;
       private String marca;
       private String modelo;
       private static int numCoches=0;

       public Coche (String color, String marca, String modelo) {

                    this.color=color;
                    this.marca=marca;
                    this.modelo = modelo;
                    Coche.numCoches++;

       }

       public void repintar(String color) {


              this.color=color;
       }

       @Override
       public String toString() {
```

```
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}


	public static void IncrementaNumeroDeCoches() {

		Coche.numCoches++;
	}

}
```

2.2.2 Creación de objetos

Volvamos al programa principal. Crearemos un objeto de coche. El proceso se denomina creación de instancias o creación de instancias de un objeto; Por lo tanto, un objeto es una instancia de una clase.

Primero necesitamos definir una variable de programa de esta clase. En segundo lugar, necesitamos llamar al constructor de clase usando una palabra clave reservada, new. Crear un objeto es fácil, por lo tanto, agregue las dos líneas siguientes después de la oración System.out.println.

```
package fundamentosorientacionaobjetos;

public class ClasePrincipal {



public static void main(String[] args) {
		// TODO Auto-generated method stub

		System.out.println("Hola como estais");

//Add the lines after this

Coche miTesla = new Coche("rojo","Tesla","Model S High Performace");

System.out.println(miTesla);

}

}
```

Cada vez que escribe new seguido de una llamada al constructor de clase, creamos un objeto de este coche. Además, puede crear tantos objetos como necesite. Hagámoslo, incluya el próximo coche a su programa.

```
Coche miBMWi = new Coche("azul"," BMW","i5");
```

```
System.out.println(miBMWi);
```

Para utilizar parte del comportamiento del coche en su programa, vuelva a pintar el BMW en blanco.

```
myBMWi.repintar("blanco");

      System.out.println("MYBMW despúes de repintar " + myBMWi);
```

Ejecute su programa y podría obtener un resultado similar a:

```
Car [color=rojo, marca=Tesla, modelo=Model S High Performace]
Car [color=azul, marca= BMW, modelo=i5]
MYBMW después de repintar Car [color=blanco, marca= BMW, modelo=i5]
```

Ejecuta el programa de nuevo, pero primero borre el método toString de la clase Coche, agregaremos de nuevo después de estas ejecuciones.

El resultado de sacar el método toString es esta ejecución:

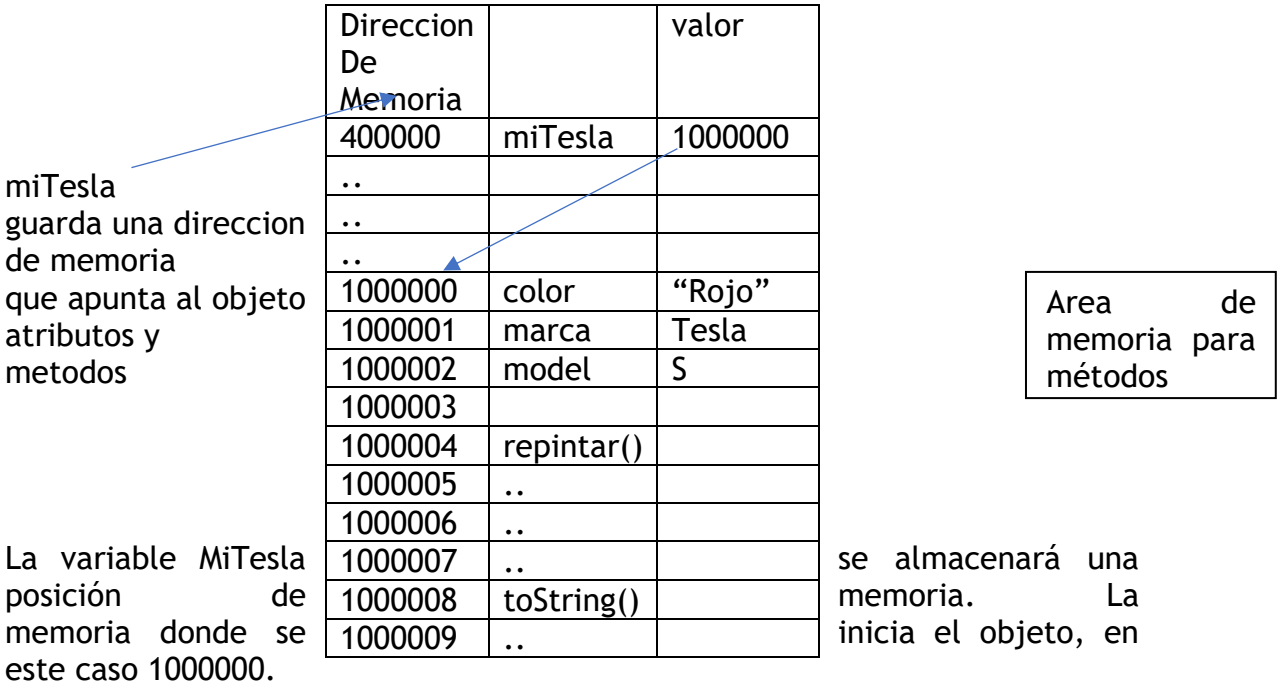
```
fundamentosorientacionaobjetos.Coche@3830f1c0
fundamentosorientacionaobjetos.Coche @39ed3c8d
MYBMW después de repintado Coche.Coche@39ed3c8d
```

Ahora el println, imprimir el contenido real de una variable de objeto, que es una dirección de memoria en hexadecimal. Deshaga la eliminación de toString y deje la clase en su estado inicial. Este ejercicio sirve como introducción a la siguiente pregunta

Pero, ¿qué sucede cuando se crea un objeto en java?

Una variable de objeto es un puntero a una ubicación de memoria, en esta ubicación de memoria se guarda toda la información del objeto, propiedades y código.

RAM Memory



2.2.3 Propiedades públicas frente a privadas. Encapsulación.

Para las propiedades de objeto Java ofrece métodos públicos para ser utilizados por otros objetos, sin embargo, puede ofrecer propiedades. Vamos a cambiar los modificadores de propiedad de coche que declaramos antes.

```
public class Coche {
	public String color;
	public String marca;
	public String modelo;
	public static final String TAG = "Objeto Coche";
```

y ahora agrega esta línea al final de tu programa principal.

```
	System.out.println("acceso a la propiedad color " + myBMWi.color);
```

Dado que hemos cambiado la declaración de propiedad, y debemos agregar el modificador public, ahora puede acceder al atributo color usando el punto "." operador. Este operador proporciona acceso a cualquier declaración pública que haya agregado a las clases. Significa que a través de la variable de objeto (después de crearse) puede acceder a su estado, propiedades o sus métodos. El

objeto variable es una variable que al mismo tiempo contiene una propiedad, tipo de variable.

Desde finales de los años 90, es una convención de programación orientada a objetos y un uso adecuado de lenguajes como java para no declarar propiedades como públicas siguiendo la **Encapsulación.** Encapsulación se basa en algunas medidas o evidencias de la experiencia del desarrollador:

1. El código puede ser más propenso a errores: estáticamente, otro objeto que modifique el estado de otro objeto llevará al programa al riesgo de varios errores para que sea menos predecible.

2. Pone en riesgo la integridad de los datos de su objeto, lo que resulta en que representa una amenaza para la integridad de los datos de su aplicación.

3. Hace que el código sea menos legible y mantenible.

Para concluir, está casi prohibido declarar propiedades públicas para producir código limpio.

Aplicamos el segundo Pilar de la Programación Orientada a Objetos

Aunque, a veces su objeto debe ofrecer variables constantes a otros. Esta práctica está permitida y es recomendable. Por ejemplo, la variable constante TAG se declara correctamente en caso de que otros objetos la necesiten.

```
public static final String TAG = "Objeto Coche";
```

El código limpio es un código que es fácil de leer, entender y mantener. El código limpio hace que el desarrollo de software sea predecible y aumenta la calidad de un producto resultante. A partir de ahora, no declararemos públicas las propiedades.

La idea es que se pueda acceder a los datos de los objetos a través de métodos. Los datos se pueden encapsular de tal manera que sean invisibles para el "mundo exterior".

Para obtener más detalles sobre cómo obtener software de calidad y refactorizar el código, puede revisar este enlace:

https://refactoring.guru/refactoring

Esto da lugar a dos formas de aproximarse a las clases:

1. Una clase es de tipo complejo en Java. Hemos declarado una variable `Coche miTesla` . Y este tipo se compone de otras variables, sus propiedades. Desde el punto de vista de un programador, una clase es un tipo
2. Además, una clase es una abstracción de un objeto real. Estas propiedades son las que definen y representan las características del coche. este objeto en nuestro programa. Para un diseñador de software, una clase es la realización de la realidad, que debe ser administrada por nuestro software.

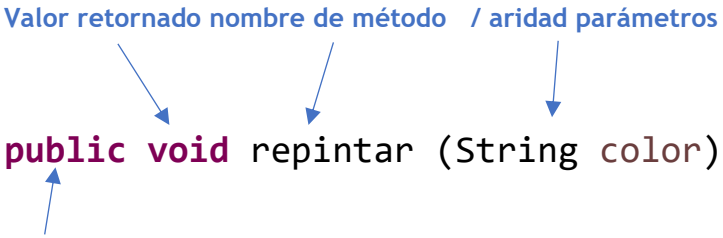
2.2.4 Métodos. Encapsulación II. Privado vs Público.

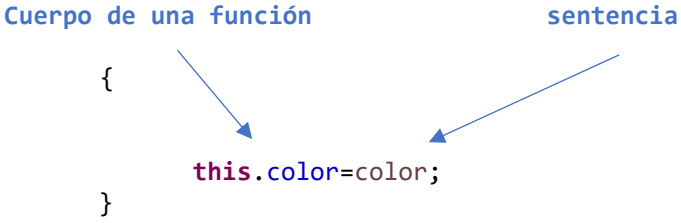
En la sección anterior se ha tratado, datos de objeto, su estado. Usando métodos, proporcionamos a los objetos operaciones, un comportamiento. Ya estudiamos que un método puede ser un procedimiento que realiza una acción pero no devuelve un resultado, o una función que devuelve resultados.

Podemos distinguir dos bloques o partes en un método:

Firma o encabezado de una función. También se conoce como la interfaz del método.
La aridad es la lista de parámetros.

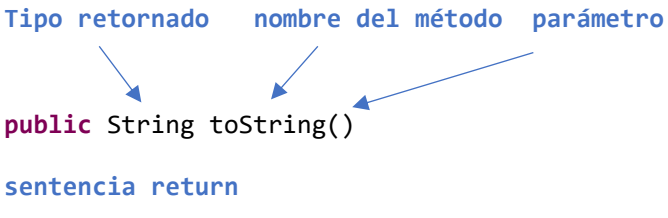
Firma o encabezado





El método repintar no devuelve valores, por lo que es un procedimiento

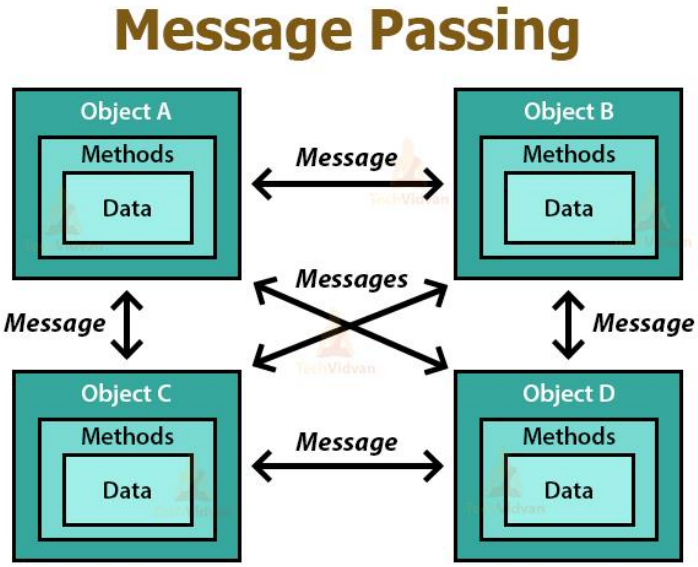
```
public void repintar (String color) {

		this.color=color;
	}
```



sentencia return

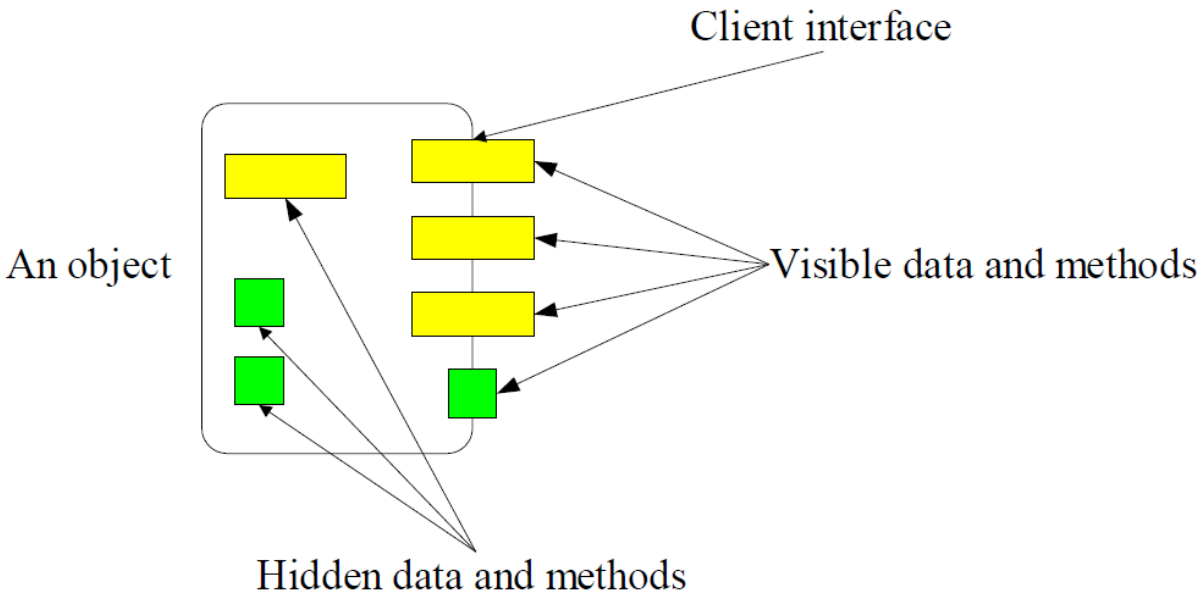
```
{
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

En este caso, el método toString es como una función. Por el contrario, no recibe parámetros. También podría tener métodos funcionan por igual que reciben parámetros. Pero hay otro enfoque para los métodos. Se pueden considerar como herramientas para recibir mensajes de otros objetos y enviar mensajes a otros objetos.

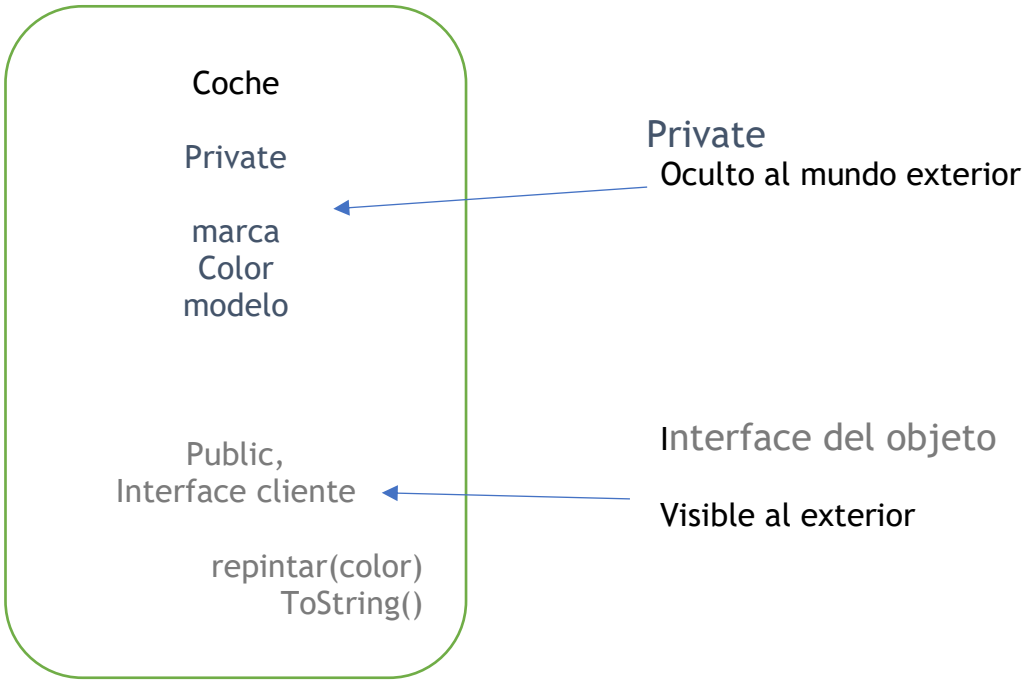


Mi clase principal enviar un mensaje al coche, para conseguir repintado en rojo.

Del mismo modo, los métodos pueden ser un medio para dar acceso a los datos públicos de los objetos. Lo que estamos tratando de lograr a través de la encapsulación es ocultar los datos y el comportamiento del objeto privado que no es necesario para otros objetos. Además, con la encapsulación estamos tratando de ofrecer una interfaz clara de la clase. Como resultado, nuestros objetos ofrecerán los métodos necesarios, "mensajes", para proporcionar su funcionalidad completa, y ocultar lo que no es conveniente mostrar, el objeto interno de trabajo.



¡De lo que el "mundo exterior" no puede ver no puede depender!
• El objeto es un "firewall" entre el objeto y el "mundo exterior".
• Los datos y métodos ocultos se pueden cambiar sin afectar al "mundo exterior".



2.2.5 Métodos de acceso.

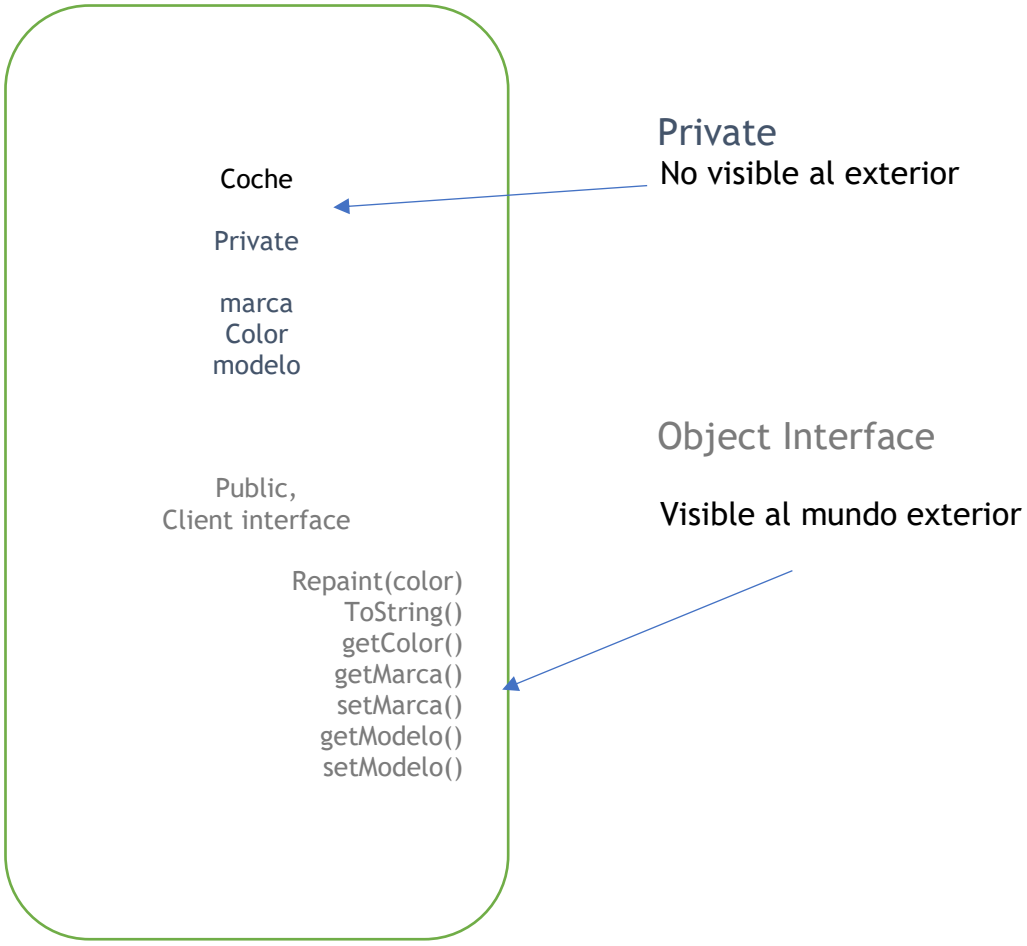
Está claro que, con el diseño actual de nuestra clase, no tenemos acceso a datos valiosos, como marca, modelo o color. Aunque debemos declarar nuestras propiedades como privadas, los programadores se comprometen a ofrecer medios para proporcionar acceso a los datos del objeto. Para hacer show debemos implementar lo que llamamos métodos de descriptor de acceso. Estos métodos proporcionan acceso y permiten realizar cambios en los datos del objeto que deben ser públicos.

Agreguemos a la clase Coche estos métodos. Se conocen como getters y setters debido a que permiten que otros objetos en nuestro programa modifiquen las propiedades de nuestro programa. Como tenemos el método repintar no ponemos el setter de Color setColor()

```
public String getColor() {
		return color;
	}


	public String getMarca() {
		return marca;
	}
```

```
public void setMarca(String marca) {
	this.marca = marca;
}

public String getModelo() {
	return modelo;
}

public void setModelo(String modelo) {
	this.modelo = modelo;
}
```

Hemos ampliado la nueva interfaz de objetos



A partir de este punto, vamos a ampliar nuestra funcionalidad de clase, agregándola nuevas características.

Agregue dos nuevas propiedades y cambie el constructor de clase:

```
public class Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;


	public static final String TAG = "Objeto Coche";

	public Car (String color, String brand, String model, double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste=coste;

	}
```

Dado que no queremos permitir que otros objetos modifiquen o accedan a estas nuevas propiedades, no agregue nuevos captadores y establecedores para estas propiedades. En su lugar, agregaremos los siguientes métodos antes del método toString. Sólo estamos interesados en ofrecer a otros objetos el beneficio obtenido por cada venta de coches.

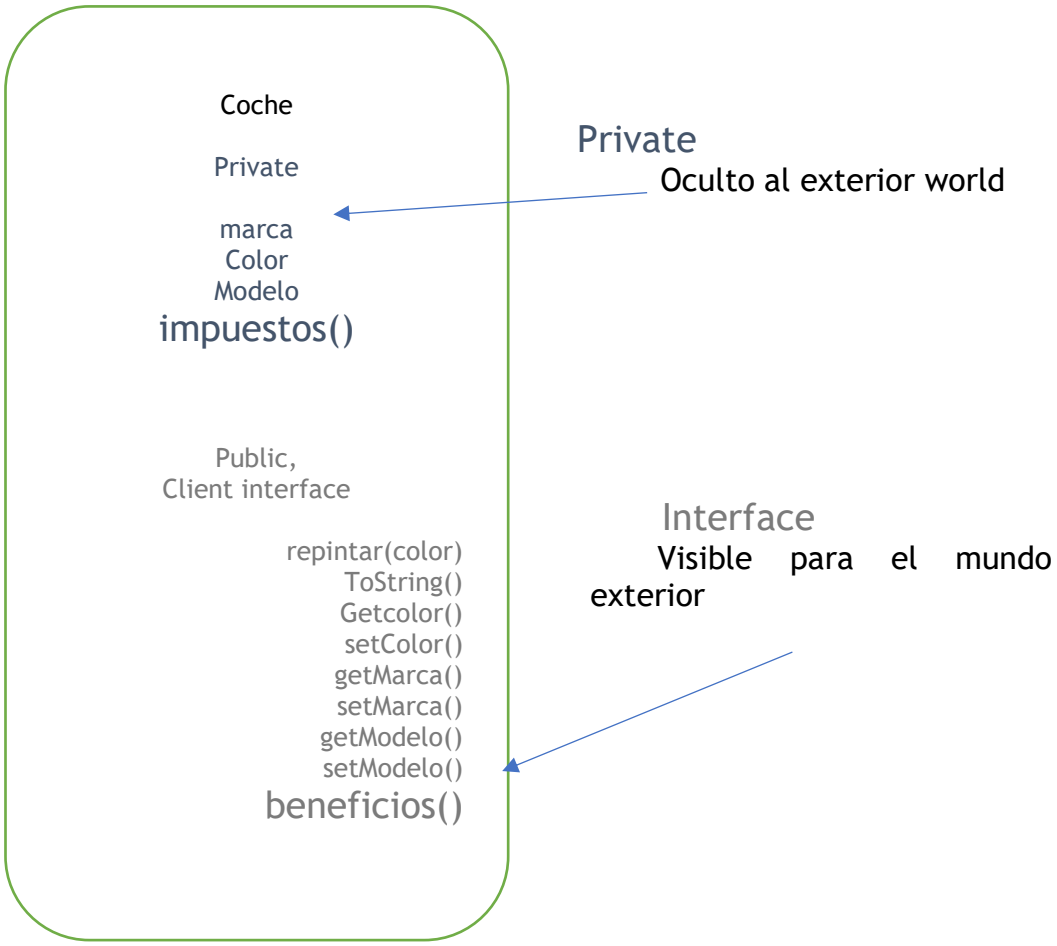
```
public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}


private double impuestos( double iva) {

		return (precio-coste)* iva;
	}
```

De lo contrario, no queremos explicar cómo calculamos los impuestos. Por un lado, el método de los impuestos está destinado a los cálculos internos. En consecuencia, lo declaramos privado, siempre que esté dentro de la clase. Al mundo exterior sólo estamos ofreciendo el método de los beneficios.

Por otro lado, el método de ganancias está dirigido a ofrecer un cálculo a otros objetos, ganancia por venta. Los métodos públicos que hemos definido antes proporcionan acceso al estado del objeto. Por el contrario, el método de ganancias ofrece un algoritmo de ganancia simple, un comportamiento.

Estos métodos, cuyo objetivo es proporcionar cálculos y algoritmos internos, se etiquetan como Métodos Privados en contraste con los Métodos De Acceso, que nos dan acceso al estado de la clase o propiedades. Además, podemos añadir a la ecuación, Métodos de Comportamiento,que ofrecen una funcionalidad al mundoexterno.

La nueva definición del objeto



Para verificar las actualizaciones de nuestro programa, agregua este código al final de su programa principal para probar nuevos métodos y verificar los resultados.

```
public static void main(String[] args) {
            // TODO Auto-generated method stub


            Coche miTesla = new Coche ("rojo","Tesla","Model S High
Performace", 50000,30000);

            System.out.println(miTesla);

            Coche miBMWi = new Coche("azul"," BMW","i5",50000,40000);

            System.out.println(miBMWi);

            miBMWi.repintar("blanco");

            System.out.println("MYBMW despues de repintar " + miBMWi);

            System.out.println("color Property access via Method " +
miBMWi.getColor());


            System.out.println("beneficio de BMW despues de venta:" +
miBMWi.beneficios(0.10));
            System.out.println("beneficio  de   Tesla  después  de  venta:"  +
miTesla.beneficios (0.10));
```

La ejecución

```
Coche [color=rojo, marca=Tesla, modelo=Model S High Performace]
Coche [color=azul, marca= BMW, modelo=i5]
MYBMW despues de repintar Coche [color=blanco, marca= BMW, modelo=i5]
color Property access via Method blanco
beneficio de BMW despues de venta:9000.0
beneficio de  Tesla después de venta:18000.0
```

Aunque declaramos variables y métodos en nuestra clase, métodos y propiedades pertenecen a objetos, son parte de cada objeto. Vale la pena señalar que cada objeto tiene su propio valor para el color propertie , Tesla color = rojo, y BMW color = azul. Posteriormente, cada objeto tiene sus propios métodos. El valor devuelto del método de objeto BMW miBMWi.beneficios(0.10) es 9000, mientras que el método de objeto Tesla miTesla.beneficios(0.10)); da como resultado 18000. Incluso si el algoritmo es el mismo, el método profit depende de las propiedades de los objetos. La clase es sólo una plantilla, los objetos

copian el código (propiedades y métodos) siempre y cuando se crean. Se puede inferir que cada vez que no usamos ningún modificador o el modificador public, private o protected, creamos un plano de una propiedad o método para ser uso de objetos.

Para concluir, en una clase se puede distinguir entre estos tipos de métodos:

- Método constructor
- Private y public(Métodos)
- Accesor (Métodos) getters y setters
- Métodos u operaciones de comportamiento

2.2.6 Metodos y clases estáticas.

En las dos secciones anteriores nos hemos tratado de propiedades y métodos. En nuestras clases, definimos propiedades y métodos para que formarán parte de un objeto, una instancia de clase. Eso se traduce en la capacidad de los objetos para almacenar sus datos o mantener su estado. Además, estos mecanismos, métodos, ofrecen una interfaz, un punto de entrada, para recibir mensajes de otros objetos, o para realizar una operación.

Sin embargo, Java, como en otros lenguajes POO, proporciona el mecanismo para crear propiedades y métodos de clase. El modificador static permite a los programadores definir propiedades y métodos que pertenecen a la clase. Las instancias de objeto no las replican, pero tienen acceso a ellas y varios objetos de la misma clase comparten estos métodos y propiedades estáticos. Para demostrarlo, debemos agregar algo de código a nuestra clase

Se nos hacen algunas adiciones a la clase de coches, resaltado en amarillo. Vamos a enumerar e ilustrar los cambios:

1. Hemos introducido una propiedad estática, una propiedad de clase *numeroDeCoches* .

   ```
   private static int numeroDeCoches =0;
   ```

1. Hemos incluido un método estático que devuelve el número de coches creados, que hacen uso de la propiedad estática numberOfCars. El otro incrementa el número de coches creados por uno En métodos estáticos sólo podemos utilizar propiedades

estáticas

```
public static int numeroDeCochesCreados() {

		return numeroDeCoches;
	}

private static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}
```

1. Hemos modificado el constructor para agregar una unidad a numeroDeCoches cada vez que se crea un automóvil. Esa actualización resulta en mantener una responsabilidad de los automóviles creados en nuestro programa

```
public Coche (String color, String marca, String modelo,double precio, double
coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste= coste;


		Coche.incrementaElNumeroDeCoches();
```

```
package fundamentosorientacionaobjetos;
public class Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;

	private static int numeroDeCoches=0;
```

```
	public Coche (String color, String marca, String modelo,double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste= coste;
			Coche.incrementaElNumeroDeCoches();

	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public String getMarca() {
		return marca;
	}

	public void setMarca(String marca) {
		this.marca = marca;
	}

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String modelo) {
		this.modelo = modelo;
	}


	public void repintar(String color) {


		this.color=color;
	}


public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}



	private double impuestos( double iva) {

		return (precio-coste)* iva;
	}
```

```
	@Override
	public String toString() {
		return "Coche [color=" + color + ", marca=" + marca + ",
modelo=" + modelo + "]";
	}



public static int numeroDeCochesCreados() {

		return numeroDeCoches;
	}

public static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}


}
```

En la memoria, estas propiedades estáticas y métodos se almacenan en un área reservada para la clase Car. Sin embargo, dado que myTesla y myBMW contienen objetos que son miembros de la clase Car, tienen permiso para acceder y utilizar estos componentes estáticos.

Además, dado que `numeroDeCochesCreados()` se ha declarado público, cualquier objeto o clase de su programa puede acceder a ellos. Por el contrario, `incrementaElNumeroDeCoches ()`, declarado como privado, puede ser el uso de los miembros de la clase.

Memoria RAM

.

Memory address		value
200000	Class Coche	
200001	*numerodeCoches*	0
	numeroDeCochesCreados ()	
	incrementaElNumeroDeCoches ()	
40000	myTesla	1000000
40001	myBMW	1200004
..		
..		
..	MiTesla objeto	
1000000	color	red
1000001	brand	Tesla
1000002	model	s
1000003		
1000004	repaint()	
1000005	..	
1000006	..	
1000007	..	
1000008	toString()	
1000009	..	
…		
…	MyBMW object	
1200004	color	blue
1200005	brand	BMW
1200006	model	I5
	…….	

Ahora se puede probar estos cambios en el programa y comprobar que funcionan como estaban dirigidos. Agregue esta línea al final de la función main.

```
System.out.println("Total de coches creados: "+ Coche. numeroDeCochesCreados
());
```

Después de ejecutar el programa y tener que crear dos coches, numeroDeCochesCreados() responder correctamente.

```
Coche [color=rojo, marca=Tesla, modelo=Model S High Performace]
Coche [color=azul, marca= BMW, modelo=i5]
MYBMW despues de repintar Coche [color=blanco, marca= BMW, modelo=i5]
color Property access via Method blanco
beneficio de BMW despues de venta:9000.0
beneficio de  Tesla después de venta:18000.0
```

Si observas la declaración `coche.numeroDeCochesCreados()`, para llamar al método estático `numeroDeCochesCreados` (), primero debe escribir el nombre de la clase, Car.

Trate de llamar desde el coche principal. `numeroDeCochesCreados` (). El compilador solicitará un error. ¿Podría inferir por qué? Extraer una conjutura.

Por último, vamos a revisar su código. Hablábamos de variables constantes en una clase en secciones anteriores. ¿Por qué declaramos esta variable constante con el modificador static? ¿Crees que puedes usarlo en tu programa principal? Suponiendo que puedas, hazlo.

```
public static final String TAG = "Car Object";
```

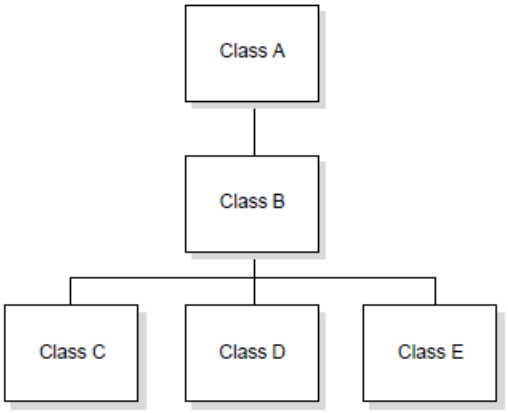
2.3 Herencia en Java

La herencia es uno de los conceptos más cruciales en la programación orientada a objetos, y tiene un efecto muy directo en la forma de diseñar y escribir las clases Java. La herencia es un mecanismo eficaz que significa que cuando se escribe una clase sólo tiene que especificar cómo esa clase es diferente de alguna otra clase, mientras que también le da acceso dinámico a la información contenida en esas otras clases.

Con la herencia, todas las clases: las que escribe, las de otras bibliotecas de clases que utiliza, y los de las clases de utilidad estándar también se organizan

en una jerarquía estricta.

Cada clase tiene una superclase (la clase por encima de ella en la jerarquía), y cada clase puede tener una o más subclases (clases por debajo de esa clase en la jerarquía). Se supone que las clases más abajo en la jerarquía heredan de las clases más arriba en la jerarquía.



Una jerarquía de clases. • Clase A es la superclase de B

• Clase B es una subclase de A

• La Clase B es la superclase de C, D y E

• Las clases C, D y E son subclases de B

Las subclases heredan todos los métodos y variables de sus superclases, es decir, en cualquier clase en particular, si la superclase define el comportamiento que necesita su clase, no tiene que redefinirlo o copiar ese código de alguna otra clase. La clase obtiene automáticamente ese comportamiento de su superclase, esa superclase obtiene el comportamiento de su superclase, y así sucesivamente hasta el final de la jerarquía.

La clase se convierte en una combinación de todas las características de las clases por encima de ella en la jerarquía.En la parte superior de la jerarquía de clases Java se encuentra la clase Object; todas las clases heredan de esta superclase.

Object es la clase más general de la jerarquía; define el comportamiento específico de todos los objetos de la jerarquía de clases Java. Cada clase más abajo en la jerarquía agrega más información y se adapta más a un propósito específico. De esta manera, se puede pensar en una jerarquía de clases como la definición de conceptos muy abstractos en la parte superior de la jerarquía y esas ideas cada vez más concretas cuanto más abajo en la cadena de superclases se va.

La mayoría de las veces cuando se escribe nuevas clases Java, se querrá crear una clase que tenga toda la información que tiene alguna otra clase, además de información adicional. Por ejemplo, es posible que desee una versión de button con su propia etiqueta integrada. Para obtener toda la información de Button, todo lo que tiene que hacer es definir la clase para heredar de Button.

Su clase obtendrá automáticamente todo el comportamiento definido en Button (y en las superclases de Button), por lo que todo lo que tiene que preocuparse son las cosas que hacen que su clase sea diferente de Button. Este mecanismo para definir nuevas clases como las diferencias entre ellas y sus superclases se denomina subclases.

2.4 Jerarquía de clases

Ahora intentaremos crear nuestra propia jerarquía creando nuevas clases que hereden de Coche.

El primero que añadiremos al pack es el SUV.

```
public class SUV extends Coche{

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);

	}

}
```

Para lograr la herencia, debe utilizar la palabra reservada extends seguida del nombre de la clase, `extends Coche`.

Si repasas el Constructor, te darás cuenta de que hay algo nuevo, una llamada a super. Super es una palabra reservada para las clases. Su propósito es llamar al constructor padre. Como resultado, se ejecutará el código de constructor primario o de superclase. Entonces este código se ejecutará:

```
		this.color=color;
                this.marca=marca;
                this.modelo = modelo;
                this.precio=precio;
                this.coste=coste;

                Coche.incrementaElNumeroDeCoches();
```

Una de las muchas ventajas de la herencia es la reutilización del código. No necesitamos volver a escribir todo el código de superclase. Como SUV hereda de Coche, todo el Código de Coche está a su disposición.

Sigamos creando un nuevo coche SUV. Pero primero, agregaremos una nueva principal, llamada AppInheritance.java.

```
package fundamentosorientacionaobjetos;
public class AppHerencia {


	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SUV miTeslaSUV = new SUV("red","Tesla","Model S High
Performace", 50000,30000);



		System.out.println("Mi nuevo SUV:" + miTeslaSUV.toString());

		System.out.println("Total de coches creados: "+
Coche.numeroDeCochesCreados());


		miTeslaSUV.repintar("blanco");


		System.out.println("Mi nuevo SUV repintado:" +
miTeslaSUV.toString());


	}

}
```

El resultado en ejecución debe ser algo como:

```
Mi nuevo SUV:Coche [color=red, marca=Tesla, modelo=Model S High Performace]
Total de coches creados: 1
Mi nuevo SUV repintado:Coche [color=blanco, marca=Tesla, modelo=Model S High
Performace]
```

"Total de coches creados:1" resulta de myTeslaSUV siendo un SUV, que es un coche ya que hereda de th Car Class. En el constructor SUV, llamamos a super, el constructor de superclase, que cuenta el número de coches creados.

Presta atención ahora a este método llame: myTeslaSUV.toString(). Incluso si este método no está implementado en la clase SUV, pueded usarlo, porque hereda todo el código de superclase.

Protected vs private

Se introducirá aquí el modificador protected. La mejor opción para describir los modificadores protected es mediante el uso. Por lo tanto, debe introducir este código en el constructor de SUV

```
package fundamentosorientacionaobjetos;

public class SUV extends Coche{

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);
		this.color= color;
	}
```

La instrucción resaltada produciría un error del compilador que resulta de la declaración de color de propiedad. Siempre que haya declarado la propiedad color como privada, las subclases heredadas no pueden acceder a ella. Parece ser incómodo que SUV ha heredado el código de clase Car, pero puede acceder directamente a las propiedades. Los modificadores de Java proporcionan niveles de seguridad sobre el acceso al código.

Sin embargo, a veces tendrá que manipular las propiedades de superclase en su subclase. No es el procedimiento más conveniente en OOP, aunque puede cambiar el modificador de propiedad de color a protected para lograrlo.

```
public class Car {
	protected String color;
```

```
    private String marca;
    private String modelo;
    private double precio;
    private double coste;
```

Comprobar que el código se está compilando

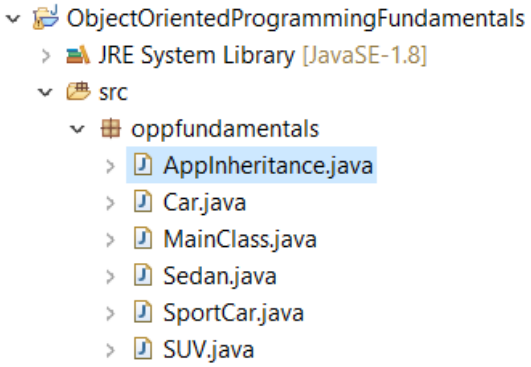
¿Por qué protegido y no público?

Los programadores usan protected en lugar de public, cuando necesitan usar una propiedad de una superclase, pero no desean que sea pública para todas las clases de su modelo de clase. El acceso desde el SUV al color de la propiedad se concede debido a los modificadores protected. Incluso una subclase de SUV, como electric SUV podría tener acceso al color propertie ya que el modificador protected otorga acceso a todas las subclases en la jerarquía, herencia directa e indirecta incluyen. No es una buena práctica, pero a veces es inevitable.

Extensión de la jerarquía de herencia

Es hora de agregar algunas clases más a la jerarquía. Por lo tanto, cree la clase CrossOver, Sedan y SportCar que heredan de Car.

El modelo de clase de proyecto debe ser similar a la siguiente figura:

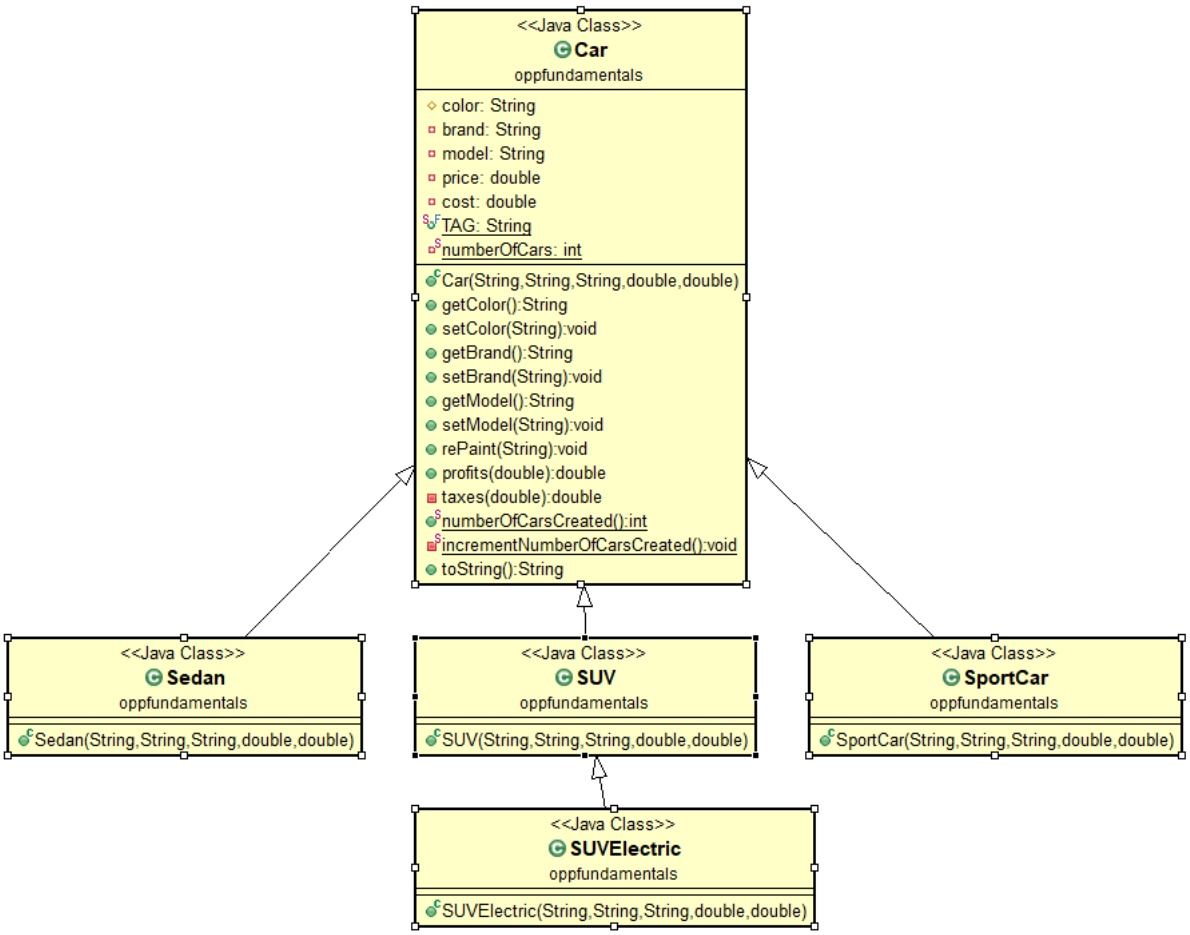


Recuerda que para hacerlos se extiende desde el coche. Finalmente agregue el SUVElectrico.java a la jerarquía. A diferencia de los otros, SUVElectric no heredará de Coche, sino que se extenderá desde SUV

```
package fundamentosorientacionaobjetos;
```

```
public class SUVElectrico extends SUV{

	public SUVElectrico(String color, String marca, String modelo, double
precio, double coste) {
		super(color, marca, modelo, precio, coste);

	}

}
```

Nuestra jerarquía de clases

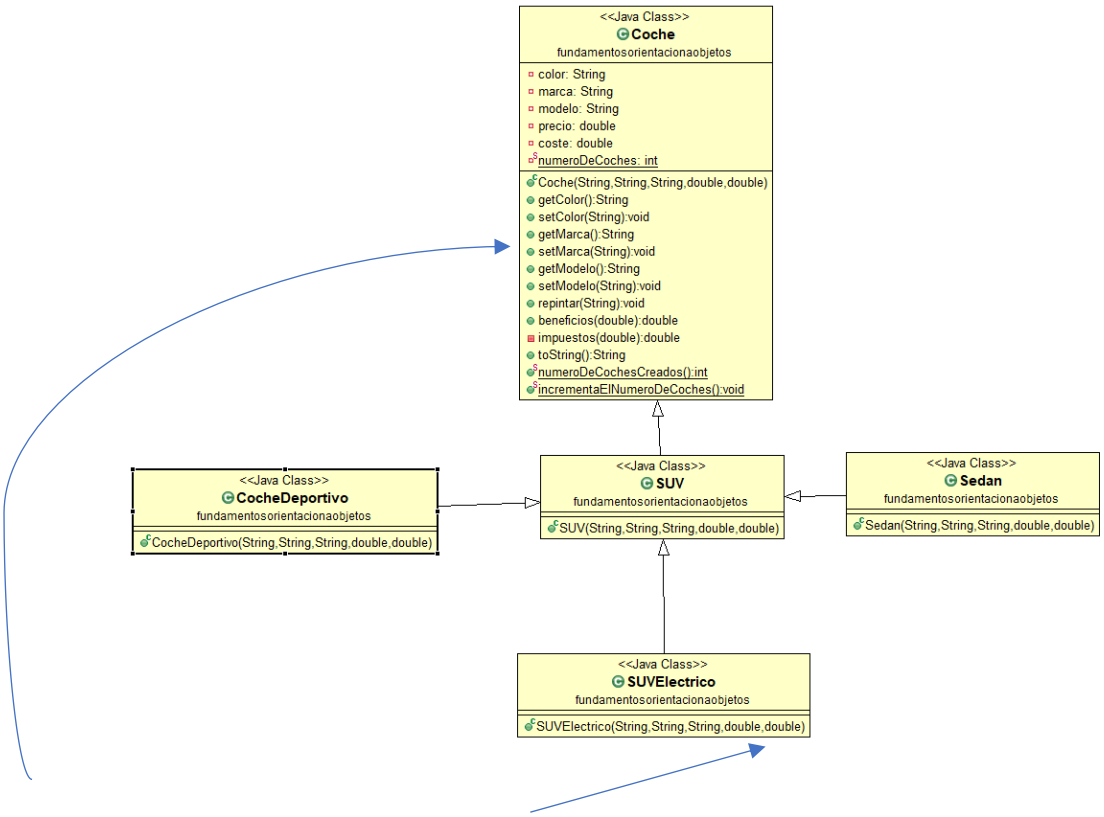


Ten en cuenta que SUVElectrico no hereda directamente de Coche. Sin embargo, a través de SUV heredará el código de la clase Coche. Por otra parte, heredará cualquier adición a SUV que no está incluido en coche. Intenta manipular el la propiedad color en la clase SUVElectrico para que pueda verificar que es plausible.

Presta atención al uso de métodos. Cuando llamamos al método beneficios en SUVElectrico

En el caso de que SUV Electrico llame al método repintar, la llamada se redirige a la definición del método en la clase Car.

```
Car miTeslaSUVElectrico = new SUV("rojo","Tesla","Model S High Performace",
50000,30000);

myTeslaSUVElectric.rePaint("green");
```




```
myTeslaSUVElectric.rePaint("verde");
```

2.4.1 Notas Cornell

Para to**das las líneas de código del ejemplo anterior que estén señaladas con el marcado**r añadir **comentarios explicando que hace cada línea**. Volver a leer los apuntes por si os surgen dudas.

3 Actividad Independiente. Jerarquía educación

Se desea crear una clase EmpleadoEducacion abstracta. Se guardará su nombre, sus apellidos, teléfono, dirección, sueldo, y porcentaje IRPF, horarioPermanencia y funciones.

Métodos abstractos: todos los métodos de acceso más dos métodos abstractos

calculoDeSueldo ()

calculoDeImpuesto ()
función() que devolverá una cadena con la función.
Deseamos también tener varios tipos de empleados:

Conserje: se desea su función será: " realizar labores de atención al publico y mantenimiento". Se desea guardar su horario y posición : planta baja o primera planta.

calculoDeSueldo -> sueldobruto -impuestos
calculoDeImpuesto -> sueldobruto* porcentaje IRPF

Profesor: su función será "Enseñar materias de su especialidad". Se desea guardar especialidad, cuerpo (primaria, secundaria o formación profesional), experiencia. Todas las propiedades tendrán método de acceso.

Métodos:

calculoDeSueldo -> sueldobruto -impuestos + bonusExperiencia
calculoDeImpuesto -> (sueldobruto + bonusExperiencia)* porcentaje IRPF
bonusDeExperiencia -> 20 * experienciaEnAños
Por cada año el bonus de experiencia es 20 euros.

Administrativos su función será "realizar labores administrativas". Se desea guardar clase 3,4 o 5. Su perfil : Contable o legislativo.

calculoDeSueldo -> sueldobruto + categoriaBonus -impuestos
calculoDeImpuesto -> (sueldobruto + bonusExperiencia)* porcentaje IRPF

categoriaBonus - > 3 300 euros, 2 200 euros, 1 100 euros.

3.1 Polimorfismo

Uno de los principales logros o características más avanzadas de la PPO es el Polimorfismo. Significa que objetos similares responden de forma similar al mismo mensaje, que se puede traducir a objetos en una jerarquía que ofrecen el mismo método, por ejemplo, el método toString que presentamos antes. Sin embargo, esta implementación del método podría diferir ligeramente de una clase a una clase.

Por ejemplo, considera un programa de dibujo que permite al usuario dibujar líneas, rectángulos, óvalos, polígonos y curvas en la pantalla. En el programa, cada objeto visible en la pantalla podría ser representado por un objeto de software en el programa. Habría cinco clases de objetos en el programa,unopara cada tipo de objeto visible que se puede dibujar. Todas las líneas pertenecerían a una clase, todos los rectángulos a otra clase, etc. Estas clases están obviamente relacionadas; todas ellas representan"objetos**dibujables".** Por el contrario, la forma en que se dibujan es diferente. El método draw() **debe diferir de una subclase a una subclase,b**aunque todos ellos sean objetos dibujables.

Dos conceptos deben ser introducidos aquí

3.2 *Sobrecarga.*

Una de las técnicas para lograr algo similar al polimorfismo en nuestras clases es aplicar sobrecarga. La idea básica de la sobrecarga es proporcionar varias implementaciones del mismo método. Lo que permite al compilador Java diferencia las versiones es la aridad del método, el tipo y el número y los parámetros que define cada versión del método. Obviamente, no podemos escribir dos versiones del mismo método con la misma aridad en la misma clase.

Los nombres de atributo de una clase pueden ser los mismos que los de otra clase, ya que se tiene acceso a ellos de forma independiente. Un atributo *x* de una clase no tiene necesariamente ninguna relación semántica con otra, ya que tienen ámbitos diferentes y no impide el uso del mismo atributo allí.

Dentro de una construcción de clase Java, los métodos pueden compartir el mismo nombre siempre que pueden distinguirse por:
1. el número de parámetros, o
2. diferentes tipos de parámetros.

Este criterio requiere que un mensaje con parámetros asociados coincida de forma única con la definición de método deseada.

Vamos a incluir esta nueva versión del método de impuestos(doble iva, doble impuestoLocal) a la clase Car. Como puede ver en el siguiente fragmento de código, el mismo método de nombre puede coexistir en la misma clase, tan pronto como muestren una aridad diferente, un tipo diferente de parámetros. En este caso, los impuestos pueden considerar sólo el vanRate, como se describe en la primera versión del método, impuestos(doble iva)

```
private double impuestos( double iva) {

            return (precio-coste)*iva;
      }
```

Sin embargo, nos gustaría que nuestro programa tomara en consideración los impuestos locales, que algunas regiones o estados aplican a las ventas de automóviles. Para ofrecer esta funcionalidad diferente, lo único que tenemos que hacer es agregar esta versión diferente del método de impuestos. Afortunadamente, el compilador de Java es capaz de distinguir sobre estas dos llamadas a métodos, debido a que el número y/o el tipo de parámetros son diferentes.

```
private double impuestos( double iva, double impuestoLocal) {

            return (price-cost)*iva + (price-cost)* impuestoLocal;


}
```

La llamada al método this.impuestos(0.20) haría referencia a la primera versión del método resaltada en azul.

La llamada al método this.impuestos(0.16, 0.5) haría referencia al resaltado en amarillo.

La sobrecarga no es un polimorfismo estricto o puro debido al hecho de que el compilador sabe en tiempo de compilación a qué método estamos haciendo referencia.

En Java, podemos sobrecargar un método tantas veces como necesitemos. Además, también podemos hacerlo para el constructor, como se muestra en el código siguiente. Reemplace la versión anterior del constructor car con estas dos nuevas versiones. Al hacerlo, estamos ofreciendo más posibilidades al programador para crear un objeto de coche.

```
public  Coche (String color, String marca, String modelo) {

            this.color=color;
            this.marca=marca;
            this.modelo = modelo;

      }


      public  Coche (String color, String marca, String modelo, double
precio, double coste) {

                  this.color=color;
                  this.marca=marca;
                  this.modelo = modelo;
                  this.precio=precio;
                  this.coste=coste;

                  Coche.IncrementaNumeroCochesCreados();

      }
```

Actividad Guiada

También se recomienda en Java, para fines de serialización, añadir una versión sin parámetros de los constructores de clase. Debe empezar a agregar esta versión del constructor a todas las clases. Por ejemplo para la clase Coche.

```
public  Coche () {

}
```

Hagámoslo con todas las clases del modelo, SUV, SUVElectric y así sucesivamente.

3.3 Actividad guiada Overriding

Reemplazar significa tener dos métodos con el mismo nombre de método y parámetros (es decir, *firma de método)*. Uno de los métodos está en la clase primaria y el otro está en la clase secundaria. Reemplazar permite a una clase secundaria proporcionar una implementación específica de un método que ya se proporciona su clase primaria.

Para seguir ilustrando el polimorfismo, estamos para explicar lo primordial. Como resultado, cambiaremos un poco nuestras subclases. En primer lugar, modificaremos el método to string en nuestras subclases.

Agregar este método a SUV

```
@Override
	public String toString() {
		return "SUV [color=" + this.getColor()+   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";;
	}
```

Y éste a Sedán

```
	@Override
	public String toString() {
		return "Sedan [color=" + this.getColor()+   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";
	}
```

Lo primero que se debe tener en cuenta es la etiqueta @Override. Si implementamos un método que ya está implementado en la superclase, necesitamos agregar el @Override. Estamos sobrescribiendo un nuevo código para este método, que es similar al de la superclase, ya que recibe los mismos parámetros. Además, el tipo devuelto es el mismo, String, y también de message es bastante similar. Compárelo con el original:

```
	@Override
	public String toString() {
		return "Car [color=" + color + ", marca=" + marca + ", modelo="
+ modelo + "]";
	}
```

La idea es que cada subclase puede tener un comportamiento diferente aunque todos sean similares. Las respuestas para cada clase podrían ser ligeramente diferentes. Imprimimos cada tipo de coche de manera diferente.

Estos son algunos hechos importantes acerca de la invalidación y la sobrecarga:

1. El tipo de objeto real en tiempo de ejecución, no el tipo de la variable de referencia, determina qué método reemplazado se utiliza en *tiempo de ejecución*. Por el contrario, el tipo de referencia determina qué método sobrecargado se utilizará en *tiempo de compilación*.
2. El polimorfismo se aplica a la invalidación, no a la sobrecarga.
3. La invalidación es un concepto en tiempo de ejecución, mientras que la sobrecarga es un concepto en tiempo de compilación.

Para representar completamente esta diferencia, estamos en el siguiente ejemplo. Agregar este código a su AppHerencia

```
Coche coche1 = new SUVElectrico("rojo","Tesla",
                    "Model S High Performace", 50000,30000);

        Coche coche2 = new Sedan("azul","BMW",
                    "320", 50000,30000);

        coche1.toString();

        coche2.toString();
```

En la siguiente sección, continuaremos representando la idea de polimorfismo a clases abstractas y subclases.

3.3.1 Clases abstractas

Suponiendo que la jerarquía crece y que está más especializada, no necesitaríamos crear instancias de clase Car. Por el contrario, la idea es crear objetos de subclases, como suv y coches deportivos. Como resultado, la clase Car se convierte más en una especie de plano o una plantilla para subclases. La clase abstracta todavía está implementando lo que es común a todas las

subclases, debido al hecho de que evitamos el código repetido en OPP.

Hay dos características interesantes de OPP y herencia:

1. En primer lugar un polimorfismo ya se menciona en la sección anterior.
2. Otra gran característica o ventaja de la herencia es la capacidad de una superclase para obligar a implementar métodos a subclases, con el modificador abstract.

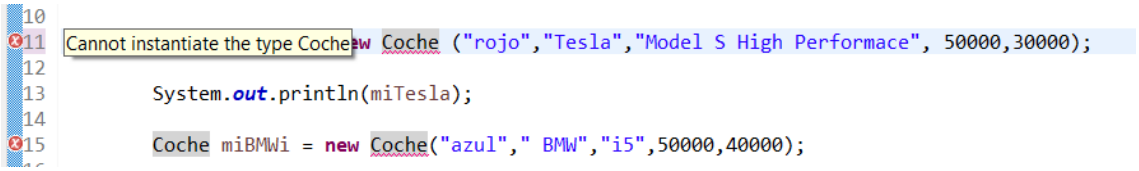
Pongámonos a trabajar:

1. Añade el modificador abstract a tu clase Car

```
public abstract class Coche
```

Las clases abstractas son clases a partir de las cuales no podemos crear objetos. A partir de ahora, el programa en MainClass debe producir un error del compilador si intenta crear un objeto Coche.

```
public abstract class Coche {
```



2. Cambie estas línea a:

```
            Coche miTesla = new SUV ("rojo","Tesla","Model S High
Performace", 50000,30000);

            System.out.println(miTesla);

            Coche miBMWi = new Sedan("azul"," BMW","i5",50000,40000);
```

3. Y también este método abstracto.

```
public abstract String getTipoCoche();
```

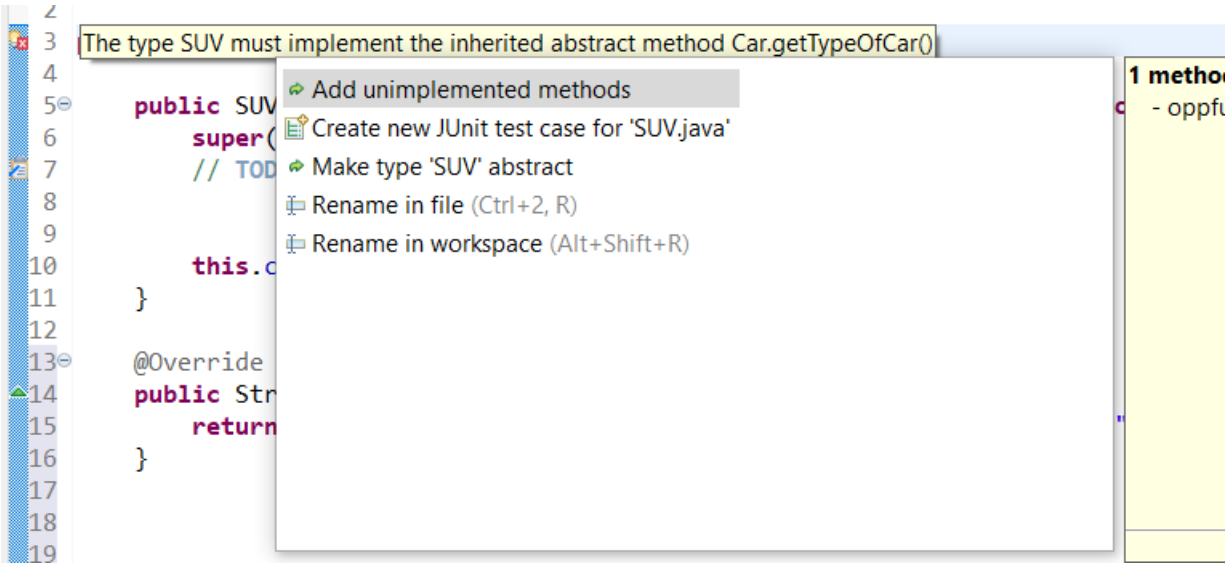
4. Los métodos abstractos son métodos sin cuerpo, sin implementación, solo ofrecemos la firma de la función. Su propósito es ser implementado en subclases que requieren polimorfismo.

Coche.java

```
package fundamentosorientacionaobjetos;
public abstract class  Coche {
	private String color;
	private String marca;
	private String modelo;
	private double precio;
	private double coste;

	private static int numeroDeCoches=0;

public abstract String getTipoCoche();



	public Coche (String color, String marca, String modelo,double precio,
double coste) {

			this.color=color;
			this.marca=marca;
			this.modelo = modelo;
			this.precio=precio;
			this.coste= coste;
			Coche.incrementaElNumeroDeCoches();

	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	public String getMarca() {
		return marca;
	}

	public void setMarca(String marca) {
		this.marca = marca;
	}

	public String getModelo() {
		return modelo;
	}

	public void setModelo(String modelo) {
		this.modelo = modelo;
	}
```

```
	public void repintar(String color) {


		this.color=color;
	}


public double beneficios(double iva) {

		return (precio-coste) -impuestos(iva);

	}



	private double impuestos( double iva) {

		return (precio-coste)* iva;
	}


	@Override
	public String toString() {
		return "Coche [color=" + color + ", marca=" + marca + ",
modelo=" + modelo + "]";
	}



public static int numeroDeCochesCreados() {


		return numeroDeCoches;
	}


public static void  incrementaElNumeroDeCoches() {

		numeroDeCoches = numeroDeCoches + 1;

	}



}
```

La cabeza a la clase SUV. También encontrará un error del compilador. No tengas miedo. Eclipse proporcionará una solución. Es necesario agregar los métodos no implementadas. Haga clic en Agregar métodos no implementadas, lo que resulta en la adición del nuevo método abstracto getTipoCoche() a su clase. Una vez agregado el método, debe comprobar que este método tiene un cuerpo. Debido al hecho de que no es abstracto, debe implementarlo.




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

Desafortunadamente, el método devuelve null. Se puede reemplazar esta declaración de devolución con "SUV";. De alguna manera, el uso de literales en la programación no es una buena práctica de codificación. Se sugiere que se agregue una variable constante en su lugar.

```
private static final String  TYPE_SUV="SUV";
```

Estas modificaciones en el programa deben ser similares al siguiente extracto de código:

```
package fundamentosorientacionaobjetos;

public class SUV extends Coche{

	private static final String  TYPE_SUV="SUV";

	public SUV(String color, String marca, String modelo, double precio,
double coste) {
		super(color, marca, modelo, precio, coste);
		// TODO Auto-generated constructor stub
	}

	@Override
```

```
	public String toString() {
		return "SUV [color=" + this.getColor() +   ", marca=" +
this.getMarca() + ", modelo=" + this.getModelo() + "]";
	}

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


}
```

Al final, la adición de métodos abstractos obliga a las subclases a proporcionarlos e implementarlos. Lo que es notable aquí es que una superclase lidera o impone el comportamiento de las subclases, debido a que si no implementan el método abstracto un error del compilador traería a colación.

3.4 Actividad independiente. Overriding

Siguiendo el último ejemplo, ¿podría hacer lo mismo para todas las clases?

3.5 Interfaces. Actividad guiada

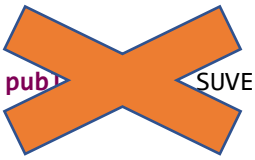
En algunos lenguajes de programación de objetos orientados, las subclases pueden heredar de más de una clase, es decir, C++, que etiquetamos como herencia múltiple. Desafortunadamente, en Java la herencia múltiple no está permitida, y eso da como resultado subclases que se extienden desde una sola superclase. No obstante, Java ofrece un mecanismo similar para simular parcialmente la herencia múltiple, interfaces.

Las interfaces son similares a las clases, pero en oposición a las clases, no ofrecen código para ser heredado por las clases. Las interfaces declaran métodos abstractos que deben ser codificados por las clases que implementan estas interfaces. Una interfaz es una colección de nombres de método, sin definiciones reales, que indican que una clase tiene un conjunto de comportamientos además de los comportamientos que la clase obtiene de sus superclases.

Podemos distinguir claramente en nuestro modelo a tipo de coches eléctricos y coches de combustibles fósiles. Entonces lo primero que deberíamos hacer es separar ambos. Por lo tanto, hacer USV eléctrico para heredar de coche, así.

```
package oppfundamentals;

public class SUVElectrico extends Car{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	public SUVElectrico(String color, String marca, String modelo, double
precio, double coste) {
		super(color, marca, modelo, precio, coste);

	}

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

}
```

En otros lenguajes POO, para mejorar la definición de eléctrico podríamos haber creado una clase llamada Electric y extenderla en SUVElectric. En Java no está permitido.



```
publ        SUVElectrico extends Coche, Electrico { PROHIBIDO
```

Para lograr este modelo deseado, podemos usar interfaces. Podemos definir una Interfaz para Electric, denominada ElectricInterface, ya que nos interesa la tienda, la capacidad de la batería y el consume del coche cada 100 km.

InterfaceElectrico.java

```
package fundamentosorientacionaobjetos;

public interface InterfaceElectrico {


	public abstract double capacidadBateria ();

	public double consumoParaCienKilometros ();


}
```

Una vez declarados los métodos, podemos explicar cómo funcionan las interfaces. Observa el método `capacidadBateria` y compruebe que se ha declarado como abstracto. Por el contrario, para la segunda, no lo hicimos, aunque no es necesario. Todos los métodos declarados pero no definidos en Interfaces son abstractos, yo no necesita usar el modificador abstract. Por lo tanto, `consumoParaCienKilometros()` es igualmente abstracto. No utilizamos el modificador abstract en las interfaces porque no es necesario. Los métodos son abstractos perse,a menos que definamos su cuerpo.

Posteriormente, necesitamos implementar esta interfaz en nuestra clase SUV. Además, es obligatorio añadir esta nueva información pero solo a los coches eléctricos.

En consecuencia, necesitamos:

1. Implementación de la interfaz

```
public class SUVElectrico extends Coche implements InterfaceElectrico
```

1. Añadir dos nuevas propiedades

```
private double capacidadBateria=0;
	private double consumo=0;
```

2. Implementando los métodos que se declararon en la interfaz

```
@Override
	public double capacidadBateria () {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}
```

```
	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}
```

```
public class SUVElectrico extends Coche implements InterfaceElectrico{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	private double capacidadBateria=0;
	private double consumo=0;


	public SUVElectrico(String color, String marca,
			String modelo, double precio, double coste,
			double capacidadBateria, double consumo) {
		super(color, marca, modelo, precio, coste);

		this.capacidadBateria= capacidadBateria;
		this.consumo= consumo;

	}




	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return TYPE_SUV_ELECTRICO;
	}



	@Override
	public double capacidadBateria() {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}



	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}

}
```

Se supone que cuando las clases implementan un Interface, firman un contrato con esta interfaz. Como resultado de este contrato, las clases deben implementar métodos de interfaz abstractos. Al igual que las clases abstractas, las interfaces imponen un comportamiento a las clases, borrándolas para definir

o implementar métodos. En este punto, los coches eléctricos deben proporcionar información sobre la capacidad de la batería y el consumo.

Podemos seguir detallando aún más nuestro modelo. Imagina que te gustaría gestionar la información sobre asientos de coche, para Berlines y SUV. Luego podemos crear una nueva interfaz para cumplir con su requisito, ISeats. Nos acostumbramos en java a comenzar los nombres de interfaz con I mayúscula, siguiendo una palabra o nombre compuesto que describen la función de interfaz.

```
package fundamentosorientacionaobjetos;

public interface IAsientos {


	public int numeroDeAsientos();

}
```

Comencemos a agregar cambios al SUVElectrico. Mientras que solo podemos extender desde una clase en Java, podemos implementar múltiples interfaces. marcado en amarillo, puede comprobar los cambios para este nuevo comportamiento, IAsientos.

```
package fundamentosorientacionaobjetos;


public class SUVElectrico extends Coche implements InterfaceElectrico,
IAsientos{
	private static final String  TYPE_SUV_ELECTRICO="SUVElectrico";
	private double capacidadBateria=0;
	private double consumo=0;
	private int numAsientos=0;


	public SUVElectrico(String color, String marca,
			String modelo, double precio, double coste,
			double capacidadBateria, double consumo, int numAsientos)
{
		super(color, marca, modelo, precio, coste);

		this.capacidadBateria= capacidadBateria;
		this.consumo= consumo;
		this.numAsientos= numAsientos;

	}
```

```
	public String getTipoCoche() {
		// TODO Auto-generated method stub
		return TYPE_SUV_ELECTRICO;
	}




	@Override
	public double capacidadBateria() {
		// TODO Auto-generated method stub
		return capacidadBateria;
	}




	@Override
	public double consumoParaCienKilometros() {
		// TODO Auto-generated method stub
		return consumo;
	}



	@Override
	public int numeroDeAsientos() {
		// TODO Auto-generated method stub
		return numAsientos;
	}

@Override
	public String toString() {
		return "SUV [color=" + this.getColor() +   ", brand=" +
this.getMarca() +
				", model=" + this.getModelo() + ",
capacidadBateria=" +
				capacidadBateria +  ", consumo=" +  consumo +
				", numAsientos=" + numAsientos + "]";
	}

}
```

Por último, podemos modificar nuestra clase AppHerencia para probar los cambios de modelo

```
package fundamentosorientacionaobjetos;


public class AppHerencia {
```

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		SUV miTeslaSUV = new SUV("red","Tesla","Model S High
Performace", 50000,30000);



		System.out.println("Mi nuevo SUV:" + miTeslaSUV.toString());

		System.out.println("Total de coches creados: "+
Coche.numeroDeCochesCreados());


		miTeslaSUV.repintar("blanco");


		System.out.println("Mi nuevo SUV repintado:" +
miTeslaSUV.toString());

		SUVElectrico coche1 = new SUVElectrico("rojo","Tesla",
				"Model S High Performace", 50000,30000,20,7,5);

		Sedan coche2 = new Sedan("azul","BMW",
				"320", 50000,30000);

		System.out.println("Mi nuevo SUV:" + coche1.toString());

		System.out.println("Mi nuevo Sedan:" + coche2.toString());

	}

}
```

El resultado de la ejecución del programa esperaba:

```
Mi nuevo SUV:SUV [color=red, marca=Tesla, modelo=Model S High Performace]
Total de coches creados: 1
Mi nuevo SUV repintado:SUV [color=blanco, marca=Tesla, modelo=Model S High
Performace]
Mi nuevo SUV:SUV [color=rojo, brand=Tesla, model=Model S High Performace,
capacidadBateria=20.0, consumo=7.0, numAsientos=5]
Mi nuevo Sedan:Sedan [color=azul, marca=BMW, modelo=320]
```

3.6 Actividad de refuerzo. Modelo de clases

1. Incluir IAsiento en la clase SUV y Sedan
2. Cree una nueva clase SedanElectrico, agregue el contrato IAsiento y InterfaceElectrico a esta clase.
3. Cree una nueva clase SedanElectrico, incluidas las interfaces mencionadas anteriormente.

4 Clases wrap para tipos primitivos. Conversión de tipos

Debido a que Java es un lenguaje de programación orientado a objetos, Java ofrece un futuro para envolver todos los tipos primitivos con clases. El propósito es poder trabajar en programas Java con clases. Estas clases wrap son similares a la clase String, ya que aunque son clases de alguna manera se comportan como tipos primitivos. Por lo tanto, una clase contenedora primitiva es una clase contenedora que encapsula, oculta o ajusta tipos de datos de los ocho tipos de datos primitivos, de modo que estos se pueden usar para crear objetos instanciados con métodos en otra clase o en otras clases. Las clases contenedoras primitivas se encuentran en la API de Java.

Las ocho clases primitivas básicas son las siguientes:

- Lang.Boolean.
- lang.Character.
- lang.Byte.
- lang.Short.
- lang.Integer.
- lang.Long.
- lang.Float.
- lang.Double.

Por ejemplo, tenemos una clase para envolver e incrustar el tipo primitivo int, llamado Integer. Un objeto de tipo Integer contiene un único atributo o propiedad cuyo tipo es int. Además, esta clase proporciona varios métodos para convertir un int en un String y un String en un int, así como otras constantes y métodos útiles cuando se trata de un int.

La clase Integer, por ejemplo, está definida de la siguiente manera:

```
Class Integer {

private int value;

public int intValue() {
 return value;
}
… mas métodos

}
```

En términos generales, estas clases primitivas similares, String incluida en la

opinión del autor, proporcionan métodos para realizar la conversión de tipos. Además, el constructor podría recibir uno de los dos parámetros de un tipo diferente. Recuerde la función de sobrecarga en Java. Permitió crear una versión diferente del mismo método.

Primitive type	Wrapper class	Constructor arguments (old versions of Java) valueOf arguments (from Java 13)
`byte`	`Byte`	`byte` or `String`
`short`	`Short`	`short` or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`float`	`Float`	`float`, `double` or `String`
`double`	`Double`	`double` or `String`
`char`	`Character`	`char`
`boolean`	`Boolean`	`boolean` or `String`

Se puede crear objetos de esta clase mediante el constructor, y también asignando literales, como se hizo con la clase String. Es más común crear estas clases con la instrucción asignar. De todos modos, siempre que se desee se puede realizar la conversión de tipos, puede usar el constructor del mismo modo, en versiones anteriores de Java. A partir de Java 13, los constructores para estas clases están en desuso, en otras palabras, se recomienda que no se

utilicen. Cuando Java u otros desarrolladores etiquetan en sus proyectos un método, campo o tipo como obsoleto, hay ciertos constructores, campos, tipos o métodos que ya no quieren que la gente use.

En consecuencia, para llevar a cabo la conversión de tipos se pueden utilizar algunos de los métodos proporcionados por estas clases, como valueOf. es decir, entero. valueOf("5"). Las clases de envoltura primitivas no son lo mismo que los tipos primitivos. Mientras que las variables, por ejemplo, se pueden declarar en Java como tipos de datos dobles, cortos, int, etc., las clases contenedoras primitivas crean objetos y métodos instanciados que heredan pero ocultan los tipos de datos primitivos, no como las variables a las que se asignan los valores de tipo de datos.

Pero primero intentaremos ilustrar cómo crear estos objetos wrap en las versiones modernas de java.

1. El constructor Integer está en desuso, no se recomienda su uso.

```
Integer iObj = new Integer("3");


Float flObj = new Float(dObje);
```

2. Podemos asignar literales a esta clase de variables, lo que da como resultado la creación del objeto automática.

```
Long lObje = 5L;
Double dObj =5.9;

Boolean blObj= true;
```

3. Podemos utilizar métodos de conversión.

```
Integer intVar =Integer.valueOf("6");

Byte bObj = Byte.valueOf("23");
```

4. Podemos obtener el valor primitivo original.

```
int iPrim = iObj.intValue();
```

5. Del mismo modo, puede asignar directamente estos objetos a variables de tipo primitivas. Como mencionamos anteriormente, estas clases solo tienen un atributo para almacenar el tipo primitivo. Por lo tanto, el compilador permite a los programadores trabajar con ellos como tipos primitivos.Se puede encontrar los posibles tipos de argumentos que se deben usar para valueOf en la tabla anterior

```
int iPrim = iObj;
long lPrim= lObje;

double dPrim =  dObj;
```

Finalmente, String ofrece un método para la conversión de String, valueOf, que puede recibir todos los tipos primitivos como argumentos a convertir. Este método está sobrecargado en la clase String, recibiendo diferentes parámetros. Preste atención al siguiente fragmento de código, de modo que estemos convirtiendo de tipos primitivos a String.

```
String convPrim = " int a String " + String.valueOf(iPrim) + "" +

                  " long a String " +       String.valueOf(lPrim) + "." +
                  " double a String " +     String.valueOf(dPrim) + "." +
                  " float a String " +      String.valueOf(flPrim) + "." +
                  " boolean a String " +    String.valueOf(blPrim) + "." ;

System.out.println("Conversión de tipos primitivos a String" + convPrim);
```

Asimismo, String.valueof() puede recibir todos los tipos de wrapper como argumentos a convertir, y eso resulta de la estructura de Wrappers, dado que almacenan el tipo primitivo en un atributo. Se ilustra en la siguiente secuencia de código:

```
            String convObj = " Integer a String " + String.valueOf(iObj) +
"." +

                  " Long a String " +       String.valueOf(lObje) + "." +
                  " Double a String " +     String.valueOf(dObj) + "." +
                  " Float a String " +      String.valueOf(flObj) + "." +
                  " Boolean a String " +    String.valueOf(blObj) + "."  ;


System.out.println("Conversión de clases wrapper primitivas a String " +
convObj);
```

En las siguientes unidades, veremos cómo java trata con los tipos primitivos y ofrecemos algunas clases de adaptador para usarlas en modelos Java más complejos y clases en la API de Java. Tampoco para decir, que la tendencia en la programación Java es usar clases wrapper en lugar de tipos primitivos.

5 Clases e interfaces con tipos genéricos en Java

En **esencia, el término genéricos significa tipos parametrizados.** Los **tipos parametrizados son importantes porque permiten crear clases, interfaces y métodos en los que se** especifica como parámetro el tipo de datos en los que *operan*. Una **clase, interfaz o método que funciona con un tipo de parámetro** se denomina **genérico,** como una clase o método genérico.

La principal ventaja del **código genérico es que funcionará automáticamente con el tipo de datos pasado a su parámetro de tipo.** Muchos algoritmos **son lógicamente iguales ,**independientemente del tipo de datos a los que se apliquen. Por ejemplo, un Quicksort (algoritmo de ordenación) es el mismo si está ordenando elementos de tipo Integer, String, Object o Thread. **Con los genéricos, puede definir un algoritmo una vez,** independientemente **de cualquier tipo específico** de datos, y luego aplicar ese algoritmo a una amplia variedad de tipos de datos sin ningún esfuerzo adicional.

A partir de Java SE 5, podemos crear clases cuyo tipo se indique en tiempo de compilación. Mira el siguiente ejemplo. Definimos una clase genérica que recibe un tipo T.class Generic<T>. El operador <> se conoce como operador Diamond. Nos permiten crear clases que manejan diferentes tipos.

Si nos fijamos en el ejemplo, tenemos crear a objeto de nuestra clase genérica. Uno para Doble, otro para Entero. Nos permite definir el tipo que utilizará nuestra clase genérica desde el momento de la compilación. Vamos a desglosar la siguiente clase:

Hemos declarado una clase genérica de tipo T genérico. La clase tiene dos propiedades Tipo T, operando1 y operando2.

```
public class EjemploClaseGenerica<T> {

	private T operando1;
	private T operando2;
```

Realizamos algunas operaciones a través de métodos para estos operandos genéricos, sin importar cuál sea el tipo. En el primero de ellos la clase Type se devuelve como String, mientras que en el segundo se comparan ambos operandos.

```
public String muestraTipo () {

		return operando1.getClass().toString();

	}
```

```
public boolean comparacion() {
		return  operando1.equals(operando2);
	}
```

En el tercero, comparamos operand1 con un valor pasado como parámetro.

```
public boolean comparationExternal(T op3) {

		return operand1.equals(op3);
	}
```

Para crear objetos para esta clase, necesitamos pasar el tipo como un paremeter usando el operador diamante. En el primer objeto, pasamos Double como parámetro. Por lo tanto, en la llamada al constructor necesitamos pasar valores de tipo doble como parámetro.

```
	EjemploClaseGenerica<Double> dobleGenerico = new
EjemploClaseGenerica<Double> (Double.valueOf(1),Double.valueOf(2));
```

Además, para cada método de la clase que recibe un parámetro T, ya que hemos definido la T parametrizada como Double, debemos pasar un Double como parámetro. Por ejemplo, puede echar un vistazo al método comparationExternal. Recibe un parámetro op3 tipado T.

```
public boolean comparacionExterna(T op3) {

		return operando1.equals(op3);
	}
```

Cuando llamamos a este método desde el objeto genericDouble, que T es Double en la creación del objeto, pasamos como parámetro un parámetro Double `Double param =1.0;`.

```
	System.out.println(" Operando 1 es el parámetro. La respuesta es " +
dobleGenerico.comparacionExterna(param));
```

Podríamos hacer lo mismo con los getters y setters. Intente introducir esta línea en el código y verifique si funciona.

```
        dobleGenerico.setOperando2(param);
```

EjemploClaseGenerica.java

```
package genericas;

public class EjemploClaseGenerica<T> {

        private T operando1;
        private T operando2;

        public EjemploClaseGenerica(T operando1, T operando2) {


                this.operando1=operando1;
                this.operando2=operando2;


        }

public T getOperando1() {
                return operando1;
        }



        public void setOperando1(T operando1) {
                this.operando1 = operando1;
        }



        public T getOperando2() {
                return operando2;
        }
```

```
	public void setOperando2(T operando2) {
		this.operando2 = operando2;
	}


public String muestraTipo () {

		return operando1.getClass().toString();

	}

	public boolean comparacion() {

		return  operando1.equals(operando2);

	}
	public boolean comparacionExterna(T op3) {


		return operando1.equals(op3);
	}


	@Override
	public String toString() {
		return "EjemploClaseGenerica [operando1=" + operando1 + ",
operando2=" + operando2 + "]";
	}




	public static void main(String[] args) {

		Double param =1.0;

		EjemploClaseGenerica<Double> dobleGenerico = new
EjemploClaseGenerica<Double> (Double.valueOf(1),Double.valueOf(2));


		System.out.println("El tipo de mi clase genérica es " +
dobleGenerico.muestraTipo());


		System.out.println("¿Son iguales los dos operandos? La respuesta
es " + dobleGenerico.comparacion());

	System.out.println(" Operando 1 es el parámetro. La respuesta es " +
dobleGenerico.comparacionExterna(param));

EjemploClaseGenerica<Integer> integerGenerico = new
EjemploClaseGenerica<Integer> (Integer.valueOf(1),Integer.valueOf(2));
```

```
            System.out.println("El tipo de mi clase genérica es " +
integerGenerico.muestraTipo());


            System.out.println("¿Son los dos operandos iguales? La respuesta
es " + integerGenerico.comparacion());




      }

}
```

Podemos extraer varias conclusiones de este ejemplo. La principal podría ser que las clases genéricas permiten crear clases que administran múltiples tipos, lo cual es una herramienta poderosa. Esta técnica está destinada a clases wrap y clases de colección. Las clases de colección son clases que gestionan una colección de objetos que estudiaremos en este curso.

Las clases genéricas nos permiten restringir el tipo T. Podemos hacer que el tipo T extienda una clase o implemente una Interfaz. Permítanme ilustrar la última declaración. Para ello, puede cambiar la declaración de clase en el ejemplo anterior.

```
public class EjemploClaseGenerica<T extends Number>
```

Number es la clase principal de clases numéricas contenedoras en Java, como Integer, Double, Long, etc. Siempre que T se extienda desde Número, estamos forzando a T a ser de Tipo de Number o una de sus subclases, Double, Long, etc. Este es un medio proporcionado por Java para restringir el Tipo T a un ámbito más concreto, en este caso Tipos numéricos Java. Teniendo en cuenta esta declaración, no podemos crear un objeto String con tipo de esta clase, como el nuevo EjemploClaseGenerica<String>.

Practica

Intenta sobreescribir el método equals para la clase genérica anterior. El método equals devolvería true si operando1, y el operando 2 son iguales para ambos objetos.

5.1 Interfaces genéricos

Las interfaces genéricas siguen los mismos principios de las clases genéricas. Sin embargo, son aún más útiles en Java por lo que solo declaramos métodos abstractos en Java, y podemos implementar cuando conocemos el tipo T que hemos pasado como parámetro para crear objetos de esta clase.

Abordemos el código de InterfacesGenericos como ejemplo. Hemos definido la interfaz GenericInterfaceExample<T> con genérico Tipo T

```
package genericas;

public interface GenericInterfaceExample<T> {

	T suma(T op1, T op2);
	T resta(T op1, T op2);
	T producto(T op1, T op2);
	T division(T op1, T op2);


}
```

Podemos hacer múltiples implementaciones de esta interfaz con diferentes tipos. Para ilustrar esto, considerad las siguientes dos clases:

La clase ClassDouble implementa la interfaz pasando como parámetro Double for T. Ahora, las implementaciones de métodos de la interfaz reciben Double como parámetros. Podemos hacer una implementación concreta de estos métodos basados en un tipo específico Doble. La respuesta para division(Double.valueOf(3), Double.valueOf(2)) debe ser 1.5.

```
package genericas;

public class ClassDouble implements GenericInterfaceExample<Double>{

	@Override
	public Double suma(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1+op2;
	}

	@Override
	public Double resta(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1-op2;
	}

	@Override
	public Double producto(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1*op2;
	}
```

```
	@Override
	public Double division(Double op1, Double op2) {
		// TODO Auto-generated method stub
		return op1/op2;
	}

}
```

Ahora observa la siguiente implementación. Es similar pero esta vez T se escribe Long. El resultado de llamar a division(Long.valueOf(3), Long.valueOf(2)) sería 1, ya que nuestra implementación se basa en un tipo entero.

```
package genericas;

public class ClassLong implements GenericInterfaceExample<Long> {

	@Override
	public Long suma(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1+op2;
	}

	@Override
	public Long resta(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1-op2;
	}

	@Override
	public Long producto(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1*op2;
	}

	@Override
	public Long division(Long op1, Long op2) {
		// TODO Auto-generated method stub
		return op1/op2;
	}


}
```

Resumiendo, los tipos genéricos llevan a nuestro código a simular comportamientos similares para tipos similares, aunque mantienen sus propias características. La división para los tipos enteros es ligeramente diferente a la división para los dobles.

6 Interfaces funcionales y lambdas. Tipos complejos en

Java

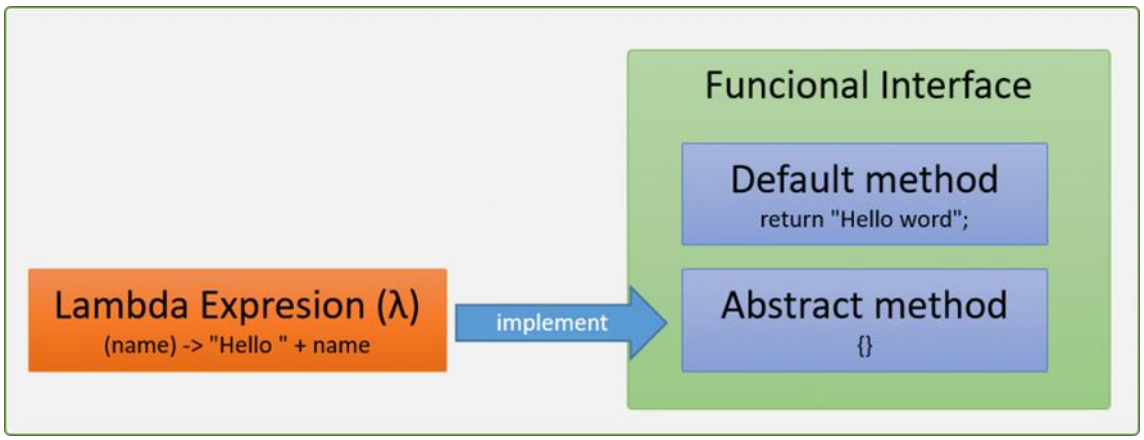
Recordamos de nuestro conocimiento en teoría del lenguaje **tres paradigmas de los lenguajes de programación: imperativo, funcional y declarativo.** Java ha sido tradicionalmente un lenguaje de programación imperativo o declarativo, basado en la mutabilidad, asignando valores a variables y transportando o transformando datos de una variable o estructura de datos a otra.

El paradigma funcional se basa en el concepto matemático de **función.** Los programas **escritos en lenguajes funcionales** consistirán en un conjunto de definiciones de funciones (entendiendo estas no como subprogramas clásicos de un lenguaje imperativo) junto con los **argumentos** sobre los que se aplican.

A partir de la versión 8 de Java, un nuevo paradigma de programación está disponible en Java. Es la **programación funcional.** Realizar código usando funciones y listas únicas o básicamente como si estuviéramos usando un lenguaje funcional como Lisp. Esta nueva forma de programar se basa en:

- **Interfaces funcionales:** interfaces que ofrecen sólo un método abstracto. Es decir, hacen un trabajo.
- **Lambda** expressions: Se **basan en el cálculo lambda.** La clave es la declaración de **funciones anónimas.** En el cálculo lambda, una función se puede declarar de forma anónima. **Por ejemplo, Square(x)x^2** se puede definir como $x.x^2$ o $(x) \rightarrow x^2$. De esta manera definiremos **nuestras funciones en** java. Y resolveremos nuestros algoritmos usando básicamente funciones.

El objetivo es **sobrescribir ese método abstracto proporcionado por la interfaz funcional** con una expresión **Lambda.** Si la interfaz no es funcional, no se le puede pasar una expresión lambda. Ahorramos instrucciones y líneas de código haciéndolo de esta manera. Se puede decir que la expresión lambda representa una función anónima que se contribuye a la interfaz funcional que define la función anónima. Aportan una acción real a la definición



6.1 Clases anónimas

Una **clase interna anónima** es una forma de clase interna que se declara e instancia con una **sola declaración.** Como resultado, **no hay ningún nombre para la clase que se pueda usar** en otra parte del programa; Quiero decir, es anónimo.

Las **clases anónimas se suelen usar en situaciones en las** que es necesario poder crear una **clase ligera** que se pase como parámetro **o argumento.** Esto generalmente se hace con una interfaz.

Por ejemplo, en el siguiente subapartado, crearemos una clase anónima a partir de otra clase y de una interfaz. Esta práctica está ampliamente extendida en Java hoy en día.

6.2 Clases anónimas a partir de una Clase. Constructores

La idea es lograr la herencia evitando la necesidad de agregar un archivo .java y escribir una nueva clase. A veces, necesitamos una nueva clase que modifique ligeramente la clase principal. En este escenario, no necesitamos escribir el código para una nueva clase, podemos crear la clase en tiempo de ejecución como se mostrará en el siguiente ejemplo. Creamos clases anónimas mediante la anulación de algunos de los métodos de clase principal en tiempo de ejecución.

En el siguiente programa, puede encontrar una clase denominada BaseClassParaAnonima. Contiene un constructor y el método metodoNombre. Lo que se intenta en el programa es crear un nuevo objeto de una nueva clase anónima. Para obtenerlo seguimos estos pasos.

1. Creamos una variable ltipo de la clase , BaseClassParaAnonima.

   ```
   BaseClassParaAnonima anon
   ```

2. Usamos el operador new para invocar el constructor BaseClassParaAnonima

   ```
   new BaseClassParaAnonima()
   ```

3. LLegado a este punto, Java permite sobreescribir dinámicamente el método de la clase padre metodoNombre.

```
        {
                        @Override
          public void metodoNombre(String name) {

                        System.out.println("Clase anonima creada sobreescrita:" +
name);
                }

            };
```

Teniendo en cuenta el `BaseClassParaAnonima` resulta que el nuevo objeto creado es diferente de uno creado a partir de la clase original. La razón es que los métodos `metodoNombre` son diferentes. Como resultado, tenemos una nueva clase de objeto, pero esta nueva clase no tiene nombre, una clase de anónimo.

El método para un objeto de la clase `BaseClassParaAnonima` es

```
public void metodoNombre(String nombre) {

                        System.out.println("Clase creada normal: " + nombre);
            }
```

El nuevo objeto creado tiene este método metodoNombre:

```
        @Override
          public void metodoNombre(String name) {

                        System.out.println("Clase anonima creada sobreescrita:" +
name);
                }
```

Como puedes comprobar son diferentes. Las clases anónimas están destinadas a crear clases a partir de interfaces como explicaremos en el siguiente ejemplo.

```
        BaseClassParaAnonima anon = new BaseClassParaAnonima() {
                @Override
                public void metodoNombre(String name) {

                        System.out.println("Clase anonima creada sobreescrita:" +
name);
                }


            };
```

BaseClassParaAnonima.java

```
package anonimas;
public class BaseClassParaAnonima {


	public BaseClassParaAnonima () {


	}

	public void metodoNombre(String nombre) {

		System.out.println("Clase creada normal: " + nombre);
	}


    public static void main(String[] args) {


	BaseClassParaAnonima objetoNormal = new BaseClassParaAnonima();

	objetoNormal.metodoNombre("Betty");

	BaseClassParaAnonima anon = new BaseClassParaAnonima() {
	  @Override
	  public void metodoNombre(String name) {

		System.out.println("Clase anonima creada sobreescrita:" +
name);
	  }


	};

	anon.metodoNombre("Mildred");

    }

}
```

6.2.1 Clases anónimas para interfaces.

Este caso es el más habitual en la programación Java. Las últimas incorporaciones del marco de API de Java, el marco del SDK de Android (para desarrolladores de Android), están diseñadas para crear clases anónimas a partir de interfaces. El procedimiento es similar al ejemplo anterior:

a. Se ha definido una interfaz `InterfaceAnonimo` en este ejemplo.

```
interface InterfaceAnonimo {

		public void metodoNombre(String name);

	}
```

b. Se declara una variable de tipo Interfaz. Este concepto ya ha sido introducido anteriormente

```
InterfaceAnonimo anon =
```

c. Java permite llamadas a una interfaz inexistente constructores (interfaces no tienen constructor) si anulamos los métodos abstractos de esta interfaz

```
new InterfaceAnonimo() {
```

d. In runtime, the abstract method methodName is implemented by the anonym class.

```
@Override
                    public void metodoNombre(String name) {

                            System.out.println("Sobreescribimos el método de
manera anónima:" + name);
                    }
```

e. Again, the outcome is an object from new class created from the interface. But this class does not exist in any file. Moreover, it has no given name, thus it is anonym.

```
package anonimas;
public class ClasesAnonimasDesdeInterfaz {

	interface InterfaceAnonimo {

		public void metodoNombre(String name);

	}

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

```
            InterfaceAnonimo anon = new InterfaceAnonimo() {
                  @Override
                  public void metodoNombre(String name) {

                        System.out.println("Sobreescribimos el método de
manera anónima:" + name);
                  }

            };

            anon.metodoNombre("Mildred");

      }

}
```

Las clases y objetos anónimos son un concepto clave para desarrollar expresiones lambda en Java. En el siguiente capítulo vamos a presentar interfaces funcionales y expresiones lambda. Las expresiones de Lambda son una nueva forma de invalidar métodos y crear clases anónimas.

6.3 Interfaces funcionales

Como se mencionó anteriormente, una interfaz funcional es una interfaz que especifica solo un método abstracto. Antes de continuar, recuerde que **no todos los métodos de interfaz son abstractos.** A partir de JDK 8, una interfaz puede tener uno o más métodos predeterminados. **Los métodos predeterminados no son** abstractos. Tampoco lo son los métodos de **interfaz estática o privada.** Por lo tanto, un método de interfaz es abstracto sólo si no especifica una implementación.

Esto significa que una interfaz funcional puede incluir métodos predeterminados, estáticos o privados, pero en todos los casos debe **tener un único método abstracto.** Debido a que los **métodos de interfaz no predeterminados, no estáticos y no privados son implícitamente abstractos,**no es necesario usar el **modificador** abstracto (aunque se puede especificarlo, si lo desea).

En definitiva, **son las herramientas para soportar expresiones lambda** con el fin de llevar a cabo este nuevo paradigma **en Java.** En el caso de las interfaces, el concepto es representar un tipo único de operación como veremos más adelante.

Mire las interfaces y clases que implementan aquellas interfaces que ya han existido en versiones anteriores de Java o con un solo **método como Listener, Runnable, Callable o Comparable.** ¿Podemos crear una clase anónimas a partir de estas interfaces? Claro.

```
Runnable thread = new Thread("Nuevo Hilo ") {
      public void run(){
        System.out.println("run by: " + getName());
      }
   };
```

```
new ActionListener() {
                public void actionPerformed(ActionEvent arg0) {

                        System.out.println("Botón pulsado:");

                }
        });
```

Todavía no hemos abordado las expresiones lambda. Una de sus funciones es anular los métodos de interfaz funcional. ¿Podemos declarar o sobrescribir **estas interfaces** y objetos de forma anónima con una expresión lambda? Sí.

Hilos Runnable con una expresión lambda.

```
Runnable threadLambda = () -> System.out.println("ejecutado por " +
getName());
```

Es exactamente lo mismo que el siguiente código

```
Runnable thread = new Thread() {
      @Override
      public void run(){
        System.out.println("ejecutado por: " + getName());
      }
   };
```

Y este código realiza la misma acción. Están sobreescribiendo el método run y creando una clase anónima Runnable tipada.

```
Runnable threadLambda = () -> System.out.println("ejecutado por: " +
getName());
```

Esas interfaces anteriores no son interfaces funcionales perse. Sin embargo, pueden comportarse como interfaces funcionales en Java. Su nombre, interfaces legacy.

Como ya lo hemos descrito, una interfaz funcional es una interfaz que solo

ofrece un método abstracto. Podemos etiquetarlos con la etiqueta de interfaz funcional @FunctionalInterface. Aunque es opcional, es recomendable.

Mostramos un ejemplo de una interfaz funcional. Incluso si la interfaz incluye tres métodos, solo uno de ellos es abstracto: `double operation(double a, double b);`

Los otros dos son predeterminados default:

```
default double identity(double num) {

                        return num;
                }
```

Y estático, static:

```
public static int transformaAInteger(double num) {

                        return (int) num;
                }
```

Se puede apreciar que tienen cuerpo. Podemos implementar métodos en interfaces de estos dos tipos.

```
@FunctionalInterface
        interface MiPrimerFunctionalInterface {

                double operacion(double a, double b);

                default double identity(double num) {

                        return num;
                }

                public static int transformaAInteger(double num) {

                        return (int) num;
                }

        }
```

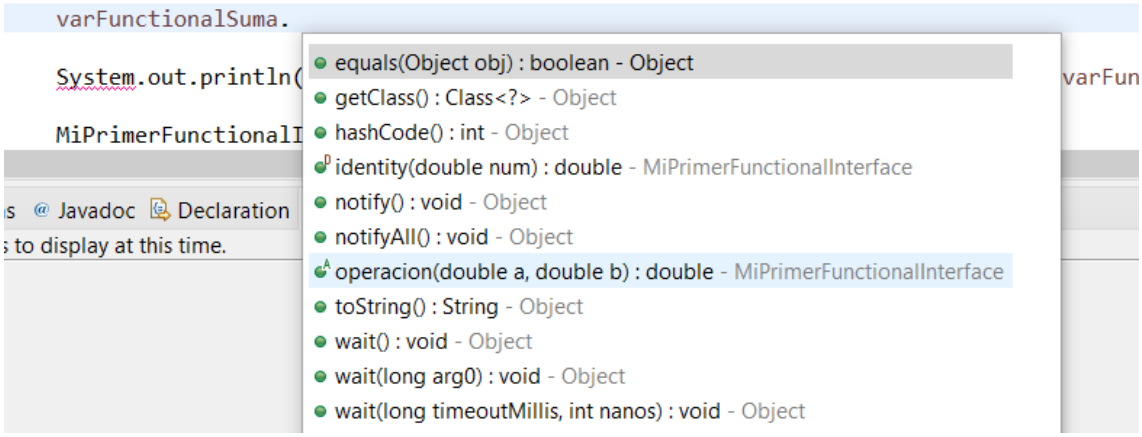
Para concluir esta exposición, cualquier clase que implemente esta interfaz solo tiene que invalidar un método, operacion.

Una vez explicada la interfaz funcional, mira cómo creamos la clase anonym, ya que seguimos el mismo patrón que anteriormente.

```
MiPrimerFunctionalInterface varFunctionalSuma = new
MiPrimerFunctionalInterface() {

                        @Override
```

```
            public double operacion(double a, double b) {
                  // TODO Auto-generated method stub
                  return a + b;
            }
};
```

Ahora tenemos un objeto Anónimo almacenado en el varFunctionalSuma. El objeto sigue la estructura definida por la interfaz. Hereda métodos de la clase Object y también del método predeterminado definido en MiPrimerFunctionalInterface:



Se puede llamar al método operacion sobreescrito, que realizará una suma:

```
System.out.println("La operacion definida en la clase anónima es la suma:"
 + varFunctionalSuma.operacion(5, 7));
```

Lo que es nuevo es el uso de la expresión lambda. Como se mencionó anteriormente, las expresiones lambda anulan el método abstracto de una interfaz. Por lo tanto, si llamamos a la operación del método varFunctionalProduct, se puede inferir que calculará el producto de estos dos números pasados como argumentos a la operación del método.

```
MiPrimerFunctionalInterface varFunctionalProduct = (x, y) -> x * y;

            System.out.println("La operación definida en la expresion lambda
producto da como resultado:"
                        + varFunctionalProduct.operacion(5, 7));
```

No te asustes, te lo explicaremos en el apartado siguiente.

El resultado de la consola de ejecución para este programa:

```
La operacion definida en la clase anónima es la suma:12.0
La operación definida en la expresion lambda producto da como resultado:35.0
```

FunctionalInterfaceEjemplo.java

```
package introduccionprogramacionfuncional;

public class FunctionalInterfaceEjemplo {

	@FunctionalInterface
	interface MiPrimerFunctionalInterface {

		double operacion(double a, double b);

		default double identity(double num) {

			return num;
		}

		public static int transformaAInteger(double num) {

			return (int) num;
		}

	}

	public static void main(String[] args) {

		MiPrimerFunctionalInterface varFunctionalSuma = new
MiPrimerFunctionalInterface() {

			@Override
			public double operacion(double a, double b) {
				// TODO Auto-generated method stub
				return a + b;
			}
		};

		System.out.println("La operacion definida en la clase anónima es
la suma:" + varFunctionalSuma.operacion(5, 7));

	MiPrimerFunctionalInterface varFunctionalProduct = (x, y) -> x * y;

		System.out.println("La operación definida en la expresion lambda
producto da como resultado:"
				+ varFunctionalProduct.operacion(5, 7));

	}

}
```

6.4 Expresiones Lambda.

La expresión lambda se basa en un elemento de sintaxis y un operador que difieren de lo que hemos visto en los temas anteriores. El operador, a veces llamado **operador lambda** u operador **de flecha**,es **->**.

Este operador divide una expresión lambda en dos partes:

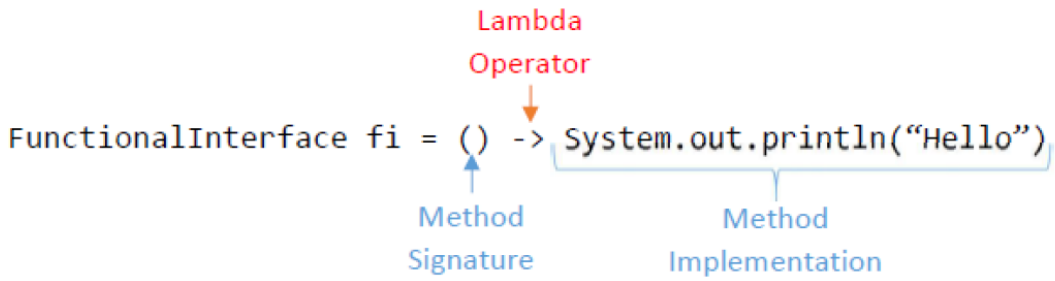
- **El lado izquierdo** especifica los argumentos requeridos por la expresión lambda.
- **En el lado derecho** está el cuerpo lambda, que especifica las acciones de la expresión lambda.

Java define dos tipos de cuerpos lambda. El primer tipo **consiste en una sola expresión, y el otro tipo consiste en** un bloque de **instrucciones** o bloque de **código.** Comenzaremos con lambdas **que definen la expresión única.**

Expresion Lambda simple

En la siguiente figura puede ver las diferentes partes de una expresión lambda:

1. La signatura o cabecera del método
2. El operador lambda
3. Implementación del método



Volvamos al ejemplo anterior:

```
interface MiPrimerFunctionalInterface {

            double operacion(double a, double b);

            default double identity(double num) {
```

```
            return num;
        }
```

```
c varFunctionalProduct = (x,y)-> x*y;
```

Repasando la expresión lambda que implementó la operación del método, podemos concluir que la firma de expresión lambda coincide con la firma de interfaz funcional. La aridad de expresión lambda difiere de la aridad del método de operación en la declaración de tipo. Mientras que las expresiones lambda no definen tipos, las interfaces sí los definen. Además, el método tiene un nombre, pero la expresión lambda no. Resolvamos este enigma.

Las expresiones lambda se basan en el cálculo lambda. Esta rama del cálculo se caracteriza por la definición de funciones anónimas. Tradicionalmente en matemáticas, las funciones se definen como f(x)= x+1;, name(variable)= expresión matemática. Por el contrario, lambda calculas permite funciones anónimas sin nombre. Para declararlos, el cálculo lambda proporciona dos notaciones:

a. λx.x+1
b. x->x+1

Algunos de los lenguajes de programación, incluido Java, utilizan la segunda notación para declarar sus propias funciones anónimas.

En el ejemplo anterior, la interfaz funcional `MiPrimerFunctionalInterface` actúa como una plantilla para las clases en general. Además, es una plantilla para todas las expresiones lambda que puede declarar siguiendo el patrón de interfaz. Al hacerlo, no es necesario declarar tipos en las expresiones lambda teniendo en cuenta que los tipos ya están declarados en la interfaz. El compilador infiere que los argumentos (x,y) son de doble tipo desde la operación del método abstracto, escríbalos como doble: `double operacion(double a, double b);`.

Siguiendo esta línea de pensamiento, x pares con doble a, y pares con doble b. Queda algo para resolver el dilema, y ese es el tipo de retorno. El tipo de retorno para la operación del método es double. Aunque no hay una instrucción return en las expresiones lambda, el compilador asume que la expresión lambda devuelve el resultado de la instrucción java `x*y.` La instrucción de retorno está implícita en el código. Esta coincidencia de patrones funciona para expresiones lambda de una sola línea. Como demostraremos más adelante, para las lambdas de instrucción de bloque, necesitamos agregar una instrucción

return. Vamos a mostrarlo con la siguiente lambda que puede agregar al final del Ejemplo `FunctionalInterfaceEjemplo`:

Expresiones Lambda de Bloque

La expresión lambda de bloque posterior derecha es una instrucción de bloque. En este caso, es obligatorio declarar explícitamente la frase de retorno al final de la instrucción block siempre que el método que anula defina un tipo de retorno.

```
MiPrimerFunctionalInterface varFunctionalbloque = (x,y)->  { x=2*x ; return x*y; };
```

El sintax para una expresión de bloque lambda es:

(param1, param2…., paramn) -> {

Instruccion 1;
Instruccion 2
…
Instruccion n;
Return Instruccion; Sólo si la interfaz funcional define un tipo de retorno.

}

La evidencia presentada nos enseña que una expresión lambda es similar a una función. Las diferencias surgen de la naturaleza anónima de lambdas, y la declaración implícita de tipos, que se infiere de la interfaz funcional relacionada con la lambda.

Se mostrará otro ejemplo para completar esta sección. Intentaremos escribir el código para una expresión lambda que muestre en la consola solo el subconjunto de los números pares de los n primeros números.

El primer paso a dar es declarar la interfaz funcional necesaria para crear una variable adecuada para la lambda. Como señalamos, la lambda no necesita devolver un valor, ya que su trabajo es mostrar números pares, por lo que el método abstracto en la interfaz devuelve void, `void displayEvenNumbers(int n);`.

Resulting from the fact that we need to print in the console the subset of the even numbers, from the n first numbers, the methods requires n as a parameter.

```
@FunctionalInterface
	interface NumerosImpares {

		void muestraNumerosImpares(int n);

	}
```

El siguiente paso es escribir el algoritmo en la expresión lambda. Declaramos una variable de interfaz NumerosImpares. Después de eso, asignamos la lambda. La lambda recibe n como argumento. El bucle for itera n veces y solo imprime el número si i%2==0, que significa el resto de dividir la variable I por 2 es cero. Recuerde que un número par es un número divisible por 2.

```
		NumerosImpares evenLambda = (n) -> {

			for (int i=0; i<=n ; i++) {

				if (i%2==0)
					System.out.println("El número " + i + " es
impar.");

			}


		};
```

Puede darse cuenta de que este lambda de bloque no tiene una instrucción return ya que el método displayEvenNumber devuelve void. Y luego, para probar este lambda, necesitamos hacer una llamada al método displayEvenNumber.

```
evenLambda.muestraNumerosImpares (10);
```

Ejecucion

```
El número 0 es impar.
El número 2 es impar.
El número 4 es impar.
El número 6 es impar.
```

```
El número 8 es impar.
El número 10 es impar.
```

EjemploNumerosImpares.java

```
package introduccionprogramacionfuncional;
public class EjemploNumerosImpares {

	@FunctionalInterface
	interface NumerosImpares {

		void muestraNumerosImpares(int n);

	}



	public static void main(String[] args) {


		NumerosImpares evenLambda = (n) -> {

			for (int i=0; i<=n ; i++) {

				if (i%2==0)
					System.out.println("El número " + i + " es
impar.");

			}


		};


		evenLambda.muestraNumerosImpares(10);

	}



}
```

Ejercicio

La idea de esta práctica es hacer que la interfaz funcional sea genérica y probar el algoritmo para Long y Float.

```
@FunctionalInterface
	interface NumerosImpares<T> {

		void muestraNumerosImpares(T n);
```

```
}
```

6.5 Interfaces predefinidos en Java

En nuestros programas anteriores, para definir una lambda necesitábamos declarar y definir una Interfaz Funcional. Para evitar eso, la API de Java ofrece interfaces funcionales predefinidas. Ahorra tiempo al desarrollador y reduce la cantidad de código para nuestros programas. En esta sección presentaremos algunas de las interfaces funcionales Java más útiles.

Revisaremos algunas de las interfaces funcionales predefinidas más útiles en Java. El paquete java.util.function contiene todas estas interfaces funcionales, esto necesitamos importarlas ya sea que las utilicemos en nuestros programas.

La documentación oficial de este paquete se encuentra en el siguiente enlace:

https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/function/package-summary.html

6.5.1 Composición de funciones

Uno de los puntos clave de la programación funcional o declarativa es la composición de funciones. Este concepto deriva del concepto matemático de composición. La mayoría de las interfaces funcionales predefinidas ofrecen la posibilidad de composición. Teniendo en cuenta eso, debemos explicar el concepto desde un punto de vista matemático.

Let us define the next two functions

F(x)= x+1
G(x)= 2x

F(2) podría evaluarse como 3 (2+1)
Los valores de G(2) son 4 (2x2)

La composición matemática permite que el matemathiciano combine funciones. Entonces, si componemos FoG(X) la evaluación matemática de FoG(2) debería funcionar como:

Primero evaluamos la función más interna G(x), G(2), el valor de retorno es 4. Usamos ese 4 como entrada para F(x) en la composición, luego debemos evaluar F(4) = 5. Como resultado, la composición de FoG(2) devuelve como valor a 5.

Ejercicio

¿Qué pasa si componemos las funciones en el orden opuesto GoF(x)?
¿Cuál sería el valor devuelto de GoF(3)?

6.5.2 Interfaz funcional de predicado

Esta interfaz se utiliza para aplicar operaciones de selección o filtrado. Los predicados en Java se implementan **con interfaces. Predicado<T>**es una interfaz funcional genérica que representa una función de argumento único que devuelve un valor booleano. Se encuentra en el paquete java.util.function. Contiene **un método test(T t)** que evalúa el predicado **dado** el argumento. Este es el **método para sobrescribir por la expresión lambda.**

Representa el tipo de operación que definiríamos como condicional a los parámetros que recibe. **Es deber del programador definir esa operación utilizando expresiones lambda** o funciones anónimas.

La definición de interfaz en la API de Java es Predicate<T>. Se puede inferir aquí que el predicado es una interfaz genérica. Funciona con diferentes tipos.

```
@FunctionalInterface
public interface Predicate<T>
```

It offers the subsequent methods:

default **Predicate**<**T**>	**and**(**Predicate**<? super **T**> other) Devuelve un predicado compuesto que representa un AND lógico de cortocircuito de este predicado y otro.
static <T> **Predicate**<T>	**isEqual**(**Object** targetRef) Devuelve un predicado que prueba si dos argumentos son iguales según**Objects.equals(Object, Object)**.
default **Predicate**<**T**>	**negate**()

	Devuelve un predicado que representa la negación lógica de este predicado.
default **Predicate**<**T**>	**or**(**Predicate**<? super **T**> other) Devuelve un predicado compuesto que representa un OR lógico de este predicado y el pasado como parámetro.
boolean	**test**(**T** t)

En la tabla, se señala que los únicos métodos abstractos a implementar es la prueba, debido a la naturaleza de la interfaz funcional del predicado. Los otros métodos son predeterminados y ofrecen funcionalidad adicional al predicado, como por ejemplo, la capacidad de combinar algunas variables de predicado. La combinación de interfaces funcionales es el resultado de aplicar el concepto matemático de composición de funciones, que detallaremos más adelante.

En el siguiente fragmento de código, estamos definiendo un predicado que verifica si un número es mayor que 10. También declaramos otro que comprueba si un número es menor que 20. Finalmente, combinamos ambos con los métodos ofrecidos por la interfaz de predicados para combinar interfaces de predicados.

Como se mencionó anteriormente, la interfaz Predicate pertenece al paquete java.util.function. Para usarlo en nuestro programa necesitamos importarlo.

```
import java.util.function.Predicate;
```

En segundo lugar, estamos definiendo una variable Predicate<Integer>mayorque10. El objetivo de la implementación es crear una lambda que pruebe si un número es mayor que 10. A partir de entonces, necesitamos declarar un predicado numérico, como Integer. En la declaración en línea del lado derecho tenemos la expresión lambda (n)-> n>10; . Recibe un número entero como parámetro y se devuelve el tipo booleano ya que estamos utilizando un operador de comparación.

Finalmente, estamos probando el predicado usando el método de prueba. El método de prueba recibe un Integer ya que hemos declarado el predicado como Integer Predicate<Integer>. Además, cuando llamamos al método de prueba lambda (n)-> n>10; porque el lambda anula el método de prueba en las interfaces funcionales de Predicates.

```
Predicate<Integer> mayorque10 = (n)-> n>10;
```

```
            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10? "+
                        mayorque10.test(numero));
```

Resultado de la ejecución:

```
Introduce un numero entero
13
¿Es el siguiente número  13 mayor que 10? true
```

Para 13, el método de prueba devuelve true, siguiendo la implementación de lambda

(13)->13>10 y 13>10 se evalúa como verdadero.

Cornell notes:

Comenta el siguiente código y explica cómo funciona

```
Predicate<Integer> menorque20 = (n)-> n<20;

            System.out.println("¿Es el siguiente número  " + numero + "
menor que 20? "+
                        menorque20.test(numero));
```

PredicateInterface.java

```
package introduccionprogramacionfuncional;

import java.util.Scanner;
import java.util.function.Predicate;

public class PredicateInterface {

      public static void main(String[] args) {


            Scanner sc = new Scanner(System.in);

            System.out.println("Introduce un numero entero");


            int numero = sc.nextInt();
```

```
            Predicate<Integer> mayorque10 = (n)-> n>10;


            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10? "+
                        mayorque10.test(numero));


            Predicate<Integer> menorque20 = (n)-> n<20;

            System.out.println("¿Es el siguiente número  " + numero + "
menor que 20? "+
                        menorque20.test(numero));



            Predicate<Integer> and = mayorque10.and(menorque20);

            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10 "+
                         " y " + " menorque20? " + and.test(numero));


            Predicate<Integer> or = mayorque10.or(menorque20);


            System.out.println("¿Es el siguiente número " + numero + " mayor
que 10 "+
                         " 0 " + " menor que 20? " + and.test(numero));


      }

}
```

Composición de Predicates

Para completar esta sección vamos a explicar la última parte del código. Estamos usando el método predeterminado y, para combinar a predicados y producir uno nuevo, Predicate<Integer>and = mayorque10. and(menorque20);. El método and devuelve un nuevo predicado que resulta de la composición lógica de los dos. Por lo tanto, si hacemos dos predicados mayorque10, que evalúa si el número es mayor que 10, y menorque20, que evalúa si un número es menor que 20, La composición and da como resultado un nuevo predicado que evalúa si el número es mayor que diez e inferior a 20.

```
Predicate<Integer> and = mayorque10.and(menorque20);

            System.out.println("¿Es el siguiente número  " + numero + "
mayor que 10 "+
                         " y " + " menor que 20? " + and.test(numero));
```

La ejecución quedaría como sigue

```
¿Es el siguiente número  13 mayor que 10  y  menor que 20? true
```

Cornell notes

Comenta este código

```
Predicate<Integer> or = mayorque10.or(menorque20);


            System.out.println("¿Es el siguiente número " + numero + " mayor
que 10 "+
                               " 0 " + " menor que 20? " + and.test(numero));


      }
```

Ejercicio

Escribe un predicado que compruebe si un número es un número primo.

El algoritmo para los números primos es:

Leer (n)
Integer i=1
boolean numeroPrimo = true

While (i<=n/2)

If (n%i==0)
        numeroPrimo =false

End While

6.5.3 El interface Consumer

El Consumidor <T> esta definido de representar o ser la función de plantilla **con** un argumento de tipo T **(parámetros)** que **devuelve nulo,** nada. Esta interfaz **consumirá básicamente algunos datos de entrada** y realizará una **acción,** es decir, la interfaz representa o define una **operación que realiza**

una acción y no devuelve ningún resultado. La interfaz ofrece el método abstracto aceptar ser reemplazado por la expresión lambda o las clases anónimas.

La declaración de la interfaz del consumidor tiene la siguiente forma:

Interface Consumer<T>

Ofrece el método de aceptación antes mencionado y algunos métodos predeterminados para la composición de funciones:

void	**accept**(T t) Realiza esta operación sobre el argumento dado.
default Consumer<T>	**andThen**(Consumer<? super T> after) Devuelve un consumidor compuesto que realiza, en secuencia, esta operación seguida de la operación posterior.

Esta interfaz es el modelo para lambdas que funciona como procedimientos. Reciben parámetros pero no devuelven ningún valor. Los programadores utilizan este tipo de interfaces funcionales para imprimir en la consola, la interfaz gráfica o para escribir en archivos o bases de datos.

Vamos a analizar el siguiente ejemplo:

```
Consumer<Integer> consumer = i -> System.out.println("Consumer 1 hace una
operación " + +i*i);
```

Este consumidor ejecutará lo que está en la lambda, el cuadrado de la variable i.

```
System.out.println("Ejecutamos el primer Consumer");
      consumer.accept(7);
```

Resultado de la ejecución:

```
Ejecutamos el primer Consumer
Consumer 1 hace una operación 49
```

Construimos el segundo consumidor a partir del primer consumidor y el método yEntonces. Este nuevo consumidor tiene ahora dos operaciones. Cuando llamamos al método accept de consumerWithAndThen,primero se ejecutará el lambda consumidor. Después de eso, ejecutará la segunda lambda, la nueva que

hemos agregado. Como se ilustra en la ejecución a continuación, la nueva marca consumerWithAndThen está compuesta por los consumidores. El original más la expresión que hemos añadido como parámetro para el método andThen. Una vez más, estamos componiendo interfaces, dos funciones.

```
Consumer<Integer> consumerWithAndThen =
                    consumer
                    .andThen(i -> System.out.println("Consumer With and Then:
Explica la operación anterior, el cuadrado de :" + i ));
```

Ejecución:

```
Ejecutamos el consumer compuesto que ejecutará los dos uno detras de otro
Consumer 1 hace una operación 36
Consumer With and Then: Explica la operación anterior, el cuadrado de :6
```

Debemos tener en cuenta que la segunda lambda no se ejecutará hasta que la primera haya cumplido con su deber. En programación, conocemos este comportamiento como una devolución de llamada, una función que se ejecuta tan pronto como la función con la que está relacionada ha terminado sus ejecuciones.

Finalmente, para demostrar que podemos crear clases anónimas a partir de Interfaces Funcionales, tenemos este fragmento de código. Aquí, estamos anulados el Método del Consumidor aceptar.

```
Consumer<Integer> consumer3 = new Consumer<Integer> () {

            @Override
            public void accept(Integer t) {
                  // TODO Auto-generated method stub

                  System.out.println("Consumer anónimo");

            }


      };
```

ConsumerInterface.java

```
package introduccionprogramacionfuncional;
import java.util.function.Consumer;

public class ConsumerInterface {
      public static void main(String[] args) {

      Consumer<Integer> consumer = i -> System.out.println("Consumer 1 hace
una operación " + +i*i);
      Consumer<Integer> consumerWithAndThen =
                  consumer
```

```
                    .andThen(i -> System.out.println("Consumer With and Then:
Explica la operación anterior, el cuadrado de :" + i ));

        System.out.println("Ejecutamos el primer Consumer");
        consumer.accept(7);
        System.out.println("Ejecutamos el consumer compuesto que ejecutará los
dos uno detras de otro");
        consumerWithAndThen.accept(6);



        Consumer<Integer> consumer3 = new Consumer<Integer> () {

                @Override
                public void accept(Integer t) {
                        // TODO Auto-generated method stub

                        System.out.println("Consumer anónimo");

                }



        };

        }

}
```

Ejercicio

A. Crear un consumer que implemente un menú que ofrezca estas opciones:

1. Convertir de Farehheit a Celsius
2. Convertir de Celsius a Fahrenheit
3. Convertir de libras a kilogramos
4. Convertir de Kilogramas to libras
5. Convertir de Kw to caballos
6. Convertir de caballos a Kw

Cada opción leerá un valor de la consola y lo convertirá en la medida de destino. Los resultados se mostrarán en la consola.

B. Implementando un bucle anidado, crear un consumer que muestre la lista de los n primeros números primos, n pasado como parámetro.

6.5.4 Supplier Interface

La interfaz Supplier realiza la función opuesta a la de los consumidores. Permite anular con una lambda que no recibe ningún parámetro y devuelve un valor. La definición de proveedor en Java es:

```
@FunctionalInterface
public interface Supplier<T>
```

El método a anular es obtener en el estado en la siguiente tabla:

Modificador y Tipo	Método y descripción
T	get() Obtiene un resultado.

Presta atención al método get. Devuelve un tipo genérico T y no declara ningún argumento. Java no ofrece composición de interfaces funcionales para Supplier. El siguiente ejemplo aborda la descripción del proveedor. Es una interfaz funcional notablemente simple. Los proveedores tienen como objetivo ofrecer valores a los programadores, siguiendo diferentes esquemas. Estos valores pueden ser números en una serie, pueden ser datos de un archivo o una base de datos.

Hemos creado dos objetos de proveedores en el ejemplo.

El primero lee un nombre de la consola y devuelve este nombre:

```
Supplier<String> supNombre = ()->{

                        i=5;
                        System.out.println("¿Puedes introducir tu nombre,
por favor?");
                        return sc.nextLine();


            };
```

Observa cómo el lado izquierdo de la lambda no tiene ningún parámetro en su definición()->

El segundo proveedor devuelve un número aleatorio del 0 al veinte. Utilizamos la clase Random de java.util para generar el número aleatorio. El método que llamamos es nextInt(20) que genera un int aleatorio desde el rango de 0 y 20.

Si revisamos el ejemplo, hemos creado el objeto pero no lo hemos asignado a una variable. En su lugar, estamos llamando al método object directamente. Es una práctica común en Java funcional.

```
new Random().nextInt(20);
```

Este programa ilustra una característica bastante importante de las lambdas. Repasemos la variable Scanner sc. Siempre que desee hacer uso de la variable dentro de la lambda, sc debe declararse final. Esto resulta de la especificación lambda, y lambdas es un objeto, dado que a los lambdas no se les permite modificar variables que no están en su ámbito. Como puede ver, sc está fuera del bloque lambda, fuera de su alcance.

El compilador no dejará usar la variable sc dentro de la lambda a menos que se defina utilizando el modificador final, lo que la hace inmutable, lo que significa que no se puede modificar. El propósito de esta característica es evitar efectos secundarios e introducir inmutabilidad en nuestras lambdas. Estas dos características son características deseadas de la programación funcional, que presentaremos en el próximo capítulo. Por el contrario, puede modificar las propiedades del objeto contenedor lambda siguiendo la especificación orientada a objetos de Java. Resaltado en amarillo, la propiedad que se puede modificar. En azul la variable local que debe declararse final para ser utilizada en el bloque lambda.

Por otro lado estamos usando la propiedad i como estática. Es la única manera de poder usarla en el cuerpo de la lambda y poder modificarla. Con el modificador final no la podríamos modificar.

```
private static  int i=0;
```

```
Supplier<String> supNombre = ()->{

                        i=5;
                        System.out.println("¿Puedes introducir tu nombre,
por favor?");
                        return sc.nextLine();


            };
```

SupplierInterface.java

```
package introduccionprogramacionfuncional;
import java.util.Random;
import java.util.Scanner;
import java.util.function.Supplier;

public class SupplierInterface {

private static  int i=0;
	public static void main(String[] args) {

		final Scanner sc = new Scanner(System.in);

		Supplier<String> supNombre = ()->{

				i=5;
				System.out.println("¿Puedes introducir tu nombre,
por favor?");
				return sc.nextLine();



		};


		System.out.println("Nombre leido por la consola: "
+supNombre.get());


		Supplier<Integer> supRandom = () -> new Random().nextInt(20);


			System.out.println("Genera un número aleatorio: "
+supRandom.get());
	}

}
```

Ejercicio

C. Según el ejemplo, puede crear un proveedor que devuelva un valor aleatorio entero de 0 a 100. Utilice el proveedor en un bucle para mostrar 50 números aleatorios.

¿Podrías hacerlo con el tipo Double?

Nota: las siguientes interfaces son similares a las funciones debido al hecho de que reciben parámetros y devuelven resultados. Estudiaremos tres de ellos aunque Java ofrece más de ellos: UnaryOperator, Function, BiFunction.

6.5.5 El interface funcional Function

Este propósito predefinido de la interfaz de función es definir una plantilla para lambdas que reciben un solo parámetro y devuelven un valor. Junto con el BiFunction y el UnaryOperator muestran un gran uso ya que la API de Java sigue este patrón para muchas de sus clases, y los programadores también codifican este tipo de lambdas.

La declaración Java para la interfaz de función tiene esta estructura:

Interface Function<T,R>

La interfaz proporciona el siguiente método, teniendo en cuenta que el método que sobreescribe la lambda es apply.

default <V> Function<T, V>	andThen(Function<? super R,? extends V> after)	Devuelve una función compuesta que primero aplica esta función a su entrada y, a continuación, aplica la función after al resultado.
R	apply(T t)	Aplica esta función al argumento dado.
default <V> Function<V, R>	compose(Function<? super V,? extends T> before)	Devuelve una función compuesta que primero aplica la función before a su entrada y, a continuación, aplica esta función al resultado.
static <T> Function<T, T>	identity()	Devuelve la función identidad f(x)=x

El método que debemos sobreescribir con la expresión lambda es R apply (T t). Si miramos más de cerca la declaración de interfaz, define dos tipos genéricos <T, R>. T es el parámetro de entrada Type. R es el valor devuelto. En este caso, para declarar una variable Function Typed es necesario introducir a los tipos incluidos en el operador diamante, es decir, Function<Integer, Double>; En este escenario, el método apply recibiría un parámetro Integer y devolvería un valor Double.

Demos un ejemplo.

```
Function<Integer,Double> raizCuadrada = (n)-> Math.sqrt(n);
```

Esta variable `raizCuadrada` almacena una lambda que sobreescribe el método Function apply. Siguiendo la declaración de interfaz Function<Integer,Double>, podemos decir que la lambda recibe un Integer y devuelve un Double. Además, el código lambda devuelve la raíz cuadrada del número pasado.

To ejecutar este código lambda que llamamos el método apply pasando un entero, 25.

```
System.out.println( " El resultado de la raiz cuadrada es " +
raizCuadrada.apply(25));
```

The result is a Double, 5.0

```
El resultado de la raiz cuadrada es 5.0
```

Esta es la versión más simple ya que hemos hecho uso del método apply. Hay otros dos métodos predeterminados interesantes para la interfaz de función, andThen, y compose. Están destinados a llevar a cabo la composición de funciones para interfaces de funciones.

La variable function1 recibe un String y devuelve la versión en mayúsculas de esta cadena.

```
Function <String,String> function1 = (s-> s.toUpperCase() + "  Function 1 to
uppercase.");

            System.out.println( " valor de function1  " +
function1.apply(stringEjemplo));
```

En la ejecución:

```
valor de function1   MI STRING COMO PARAMETRO   Function 1 pasa a mayusculas.
```

La variable function2 recibe un String y devuelve la misma cadena, pero los espacios en blanco exteriores se han borrado.

```
            Function <String,String> function2 = (s->s.trim() + " Function 2
elimina blancos.");
            System.out.println( " valor de function2  " +
function2.apply(stringEjemplo));
```

En la ejecución:

```
valor de function2  Mi String como parametro Function 2 elimina blancos.
```

La función de interfaz3 combina las dos interfaces anteriores. Ya hemos introducido el método andThen para otras interfaces. Se sabe que el andThen funciona como una devolución de llamada. Una vez que la función1 devuelve un resultado, este resultado va como parámetro a la función2. Hasta que la primera función no termina, la segunda no se ejecuta. Además, el resultado, el resultado de la función1 es la entrada de la función2. Este es el resultado que la function3.apply(ejemploString) **devolverá, en este orden.**

```
Function <String,String> function3 = function1.andThen(function2);
            System.out.println(      "      function3      value      "      +
function3.apply(stringExample));
```

En la salida del ejemplo puede ver cómo se ha evaluado primero la Función 1.

```
valor de function3  MI STRING COMO PARAMETRO   Function 1 pasa a mayusculas.
Function 2 elimina blancos.
```

Y por último, el método de composición. Ya hemos descrito la composición de la función. En la composición de dos funciones matemáticas, FoG(x), la primera en ser evaluada es G(x), la función más interna. La salida de G(x) será la entrada de F(x) la función externa. El método compose funciona siguiendo este patrón. Como resultado, en el siguiente ejemplo, function2 se ejecutará primero. El resultado de function2, será la entrada de function1. Este es el resultado que la functionn4.apply(stringEjemplo) **devolverá, en este orden.**

```
      Function <String,String> function4= function1.compose(function2);
            System.out.println( " valor de function4 " +
function4.apply(stringEjemplo) );
```

En la salida de ejemplo puede ver cómo se ha evaluado primero la Función 2.

```
valor de function4 MI STRING COMO PARAMETRO FUNCTION 2 ELIMINA BLANCOS.
Function 1 pasa a mayusculas.
```

FunctionInterface.java

```
package introduccionprogramacionfuncional;
import java.util.Scanner;
import java.util.function.Function;

public class FunctionInterface {


	public static void main(String[] args) {

		String stringEjemplo=" Mi String como parametro ";

		Scanner sc = new Scanner(System.in);


		Function<Integer,Double> raizCuadrada = (n)-> Math.sqrt(n);

		System.out.println( " El resultado de la raiz cuadrada es " +
raizCuadrada.apply(25));


		Function <String,String> function1 = (s-> s.toUpperCase() + "
Function 1 pasa a mayusculas.");

		System.out.println( " valor de function1  " +
function1.apply(stringEjemplo));

		Function <String,String> function2 = (s->s.trim() + " Function 2
elimina blancos.");
		System.out.println( " valor de function2  " +
function2.apply(stringEjemplo));

		Function <String,String> function3 =
function1.andThen(function2);
		System.out.println( " valor de function3  " +
function3.apply(stringEjemplo));

			Function <String,String> function4=
function1.compose(function2);
		System.out.println( " valor de function4 " +
function4.apply(stringEjemplo) + "function 1 pasa a mayusculas");



	}

}
```

6.5.6 The Unary Operator Interface

El operador unario es casi idéntico a la interfaz de función. La única diferencia destacable es que devuelve el mismo Tipo que recibe como parámetro dado que se extiende desde la interfaz Función. En consecuencia, no es necesario definir el Tipo dos veces.

La estructura del operador unario es:

```
public interface UnaryOperator<T>
extends Function<T,T>
```

Y los métodos ofrecidos por esta interfaz son los mismos que la interfaz de función proporciona, apply, andThen, compose e identity, . Se podría afirmar que es una especialización de la primera.

En el siguiente ejemplo, se define el interfaz factorial como Long, donde Long es el Tipo recibido como parámetro y también el tipo devuelto.

```
UnaryOperator<Long> factorial
```

UnaryOperatorExample.java

```
package PredefinedInterfaces;

import java.util.function.UnaryOperator;

package introduccionprogramacionfuncional;

import java.util.function.UnaryOperator;

public class UnaryOperatorEjemplo {


public static void main(String[] args) {

            UnaryOperator<Long> factorial = (n) -> {

                  Long res= 1L;
                  for (int i = 1; i<=n ;i++) {


                        res= res*i;
                  }

                  return res;
            };
```

```
            System.out.println("The factorial de 5 es: " +
factorial.apply(5L));

}

}
```

Activity

Agregue al ejemplo anterior un operador unario Long potenciaDeDos que calcula la potencia del número. Después de eso, combínalo con el factorial para obtener un nuevo UnaryOperator:

a. Usando el método andThen
b. Con el método compose

Mostrar los diferentes resultados en la consola.

6.5.7 Interfaz funcional Bifunction

This interface is a function alike interface that receives two generic types as an input, and the third one would be the returned value.

La estructura del interface Bifunction es:

```
@FunctionalInterface
public interface BiFunction<T,U,R>
```

T y U son los tipos de los parámetros de entrada. R es el valor retornado

El método sobreescrito por la expression lambda es R apply(T t. U u). En total, esta interfaz ofrece estos dos métodos.: andThen y apply.

default <V> BiFunction<T, U,V>	andThen (Function<? super R,? extends V> after)	Returns a composed function that first applies this function to its input, and then applies the after function to the result.
R	apply (T t, U u)	Applies this function to the given arguments.

Un ejemplo de Interface Bifunction

The siguiente interfaz recibe dos enteros como parámetro y devuelve un Doble.

```
BiFunction<Integer, Integer, Double> floatDiv
```

```
package PredefinedInterfaces;

package introduccionprogramacionfuncional;
import java.util.function.BiFunction;

public class BifunctionInterfaceEjemplo {


	public static void main(String[] args) {

		BiFunction<Integer, Integer, Double> floatDiv = (a,b) ->
(double) a/b;



		BiFunction <String, String, String > concatMayusculas = (s1,s2)-
> (s1 +s2).toUpperCase();


		System.out.println ("Division decimal de 5 y 7:" +
floatDiv.apply(5, 7));


		System.out.println ("concatMayusculas interface:" +
concatMayusculas.apply("String 1", "String 2"));




	}

}
```

6.5.8 Interfaces funcionales para tipos primitivos

Veamos los diferentes tipos de especialización de interfaces funcionales predefinidas que se ven en el tema anterior. Para casi todos los tipos primitivos en Java tenemos su especialización en Interfaz Funcional. Si presta atención a los ejemplos anteriores de interfaces funcionales, todos ellos reciben tipos genéricos. Los tipos genéricos no pueden ser tipos primitivos. Para solucionar este problema, Java ofrece interfaces funcionales que devuelven o reciben

tipos primitivos, evitando la necesidad de pasarlos como tipos genéricos, lo que no está permitido.

Dado que un tipo primitivo no puede ser un argumento de **tipo genérico,** existen versiones de la interfaz Function para los tipos primitivos más utilizados **double, int, long y sus** combinaciones en tipos de argumento y tipo devuelto:

1. ***IntFunction, LongFunction, DoubleFunction- Los argumentos* son del tipo especificado,**el tipo devuelto está parametrizado.
1. ***ToIntFunction, ToLongFunction, ToDoubleFunction:*** Los tipos de retorno son de tipo **especificado,** los argumentos están parametrizados.
1. ***DoubleToIntFunction, DoubleToLongFunction, IntToDoubleFunction, IntToLongFunction, LongToIntFunction, LongToDoubleFunction*** - Tienen un **argumento** y un tipo de retorno definidos como tipos primitivos, según lo especificado por sus nombres.

En el siguiente ejemplo utilizamos algunas de estas especializaciones.

La interfaz IntFunction, recibe un tipo primitivo int y devuelve un tipo Genérico, String en este caso.

```
IntFunction<String> convertdeIntaString = (i)-> String.valueOf(i);
```

La interfaz toIntFunction, recibe un Generic Type, String, y devuelve un tipo primitivo int.

```
ToIntFunction<String> convertdeStringaInt = (s -> Integer.valueOf(s));
```

Observe cómo siempre definimos el tipo Genérico en el diamante operador. El tipo primitivo se infiere del nombre de la interfaz funcional.

En el último ejemplo no hay ningún tipo genérico pasado como parámetro. Tanto int como doble son tipos primitivos.

```
IntToDoubleFunction convertdeIntaDouble = (i -> Double.valueOf(i));
```

Con todo, Java incluye estas especializaciones para permitir al programador utilizar tipos primitivos en sus expresiones lambda.

PrimitiveTypeSpecialization.java

```
package PredefinedInterfaces;

package introduccionprogramacionfuncional;
import java.util.Scanner;
import java.util.function.IntFunction;
import java.util.function.IntToDoubleFunction;
import java.util.function.ToIntFunction;

public class PrimitiveTypeSpecialization  {



	public static void main(String[] args) {



		Scanner miScanner = new Scanner(System.in);
		System.out.println("Introduce un número entero");
		int numero1 = miScanner.nextInt();

		IntFunction<String> convertdeIntaString = (i)->
String.valueOf(i);

		ToIntFunction<String> convertdeStringaInt = (s ->
Integer.valueOf(s));

		IntToDoubleFunction convertdeIntaDouble = (i ->
Double.valueOf(i));

		String cadena = convertdeIntaString.apply(numero1);

		System.out.println("Convertir de tipo primitivo int  " + numero1
+ " a String " + cadena);

		int numero2 = convertdeStringaInt.applyAsInt(cadena);

			System.out.println("Convertir de String  " + cadena + " a
tipo primitivo in " + numero2);

		Double numerodecimal =
convertdeIntaDouble.applyAsDouble(numero1);


		System.out.println("Convertir de primitivo int " + numero1 + " a
primitivo double " + numerodecimal);

	}

}
```

7 Programación funcional

7.1 Categorización de paradigmas de programación

Por supuesto, la programación funcional no es el único estilo de programación en la práctica. En términos generales, los estilos de programación se pueden clasificar en paradigmas de programación imperativos y declarativos:

El **enfoque imperativo** define un programa como una secuencia de declaraciones que cambian el estado del programa hasta que alcanza el estado final. La programación procedimental es un tipo de programación imperativa en la que construimos programas utilizando procedimientos o subrutinas. Uno de los paradigmas de programación populares conocidos como programación orientada a objetos (POO) extiende los conceptos de programación procedimental.

En contraste, el **enfoque declarativo** expresa la lógica de un cálculo sin describir su flujo de control en términos de una secuencia de declaraciones. En pocas palabras, el enfoque del enfoque declarativo es definir lo que el programa tiene que lograr en lugar de cómo debería lograrlo. La programación funcional es un subconjunto de los lenguajes de programación declarativos.

7.2 Programación funcional

La programación funcional es un paradigma de programación en el que tratamos de unir todo en el estilo de funciones matemáticas puras. Es un tipo declarativo de estilo de programación. Su enfoque principal está en "qué resolver" en contraste con un estilo imperativo donde el enfoque principal es "cómo resolver". Utiliza expresiones en lugar de declaraciones. Una expresión se evalúa para producir un valor, mientras que una instrucción se ejecuta para asignar variables. Esas funciones tienen algunas características especiales que se analizan a continuación.

La programación funcional se basa en el cálculo lambda:

El cálculo lambda es un marco desarrollado por Alonzo Church para estudiar cálculos con funciones. Se puede llamar como el lenguaje de programación más pequeño del mundo. Da la definición de lo que es computable. Cualquier cosa que pueda ser calculada por el cálculo lambda es computable. Es equivalente a la máquina de Turing en su capacidad de computar. Proporciona un marco teórico para describir las funciones y su evaluación. Forma la base de casi todos los lenguajes de programación funcionales actuales.

Nota: Alan Turing fue un estudiante de la Alonzo Church y creó la máquina Turing que sentó las bases del estilo de programación imperativo.

Lenguajes de programación que soportan programación funcional: Haskell, JavaScript, Scala, Erlang, Lisp, ML, Clojure, OCaml, Common Lisp, Racket.

Los lenguajes de programación funcionales se clasifican en dos grupos, es decir,

- Lenguajes funcionales puros: estos tipos de lenguajes funcionales solo admiten los paradigmas funcionales. Por ejemplo − Haskell.

- Lenguajes funcionales impuros: estos tipos de lenguajes funcionales admiten los paradigmas funcionales y la programación de estilo imperativo. Por ejemplo − LISP.

7.3 Functional Programming Advantages

- Código libre de errores − La programación funcional no admite el estado, por lo que no hay resultados de efectos secundarios y podemos escribir códigos sin errores.

- Programación paralela eficiente: los lenguajes de programación funcionales NO tienen estado mutable, por lo que no hay problemas de cambio de estado. Se puede programar "Funciones" para que funcionen en paralelo como "instrucciones". Tales códigos admiten una fácil reutilización y estabilidad.

- Eficiencia: los programas funcionales consisten en unidades independientes que pueden ejecutarse simultáneamente. Como resultado, tales programas son más eficientes.

- Soporta funciones anidadas: la programación funcional admite funciones anidadas.

- Evaluación Lazy − La programación funcional admite construcciones funcionales perezosas como listas perezosas, mapas perezosos, etc. La evaluación Lazy o por necesidad es una estrategia de evaluación que

retrasa el cálculo de una expresión hasta que su valor sea necesario, y que también evita repetir la evaluación en caso de ser necesaria en posteriores ocasiones. Esta compartición del cálculo puede reducir el tiempo de ejecución de ciertas funciones de forma exponencial, comparado con otros tipos de evaluación.

Basándose en la idea de la programación declarativa, la programación funcional busca resolver problemas de algoritmos utilizando llamadas a funciones. El siguiente ejemplo tratará de ilustrar las diferencias sobre la programación imperativa y declarativa:

Una vez que hemos leído el número al que queremos calcular la tercera potencia, las soluciones imperativas se basan en la declaración de variables y la asignación para resolver el problema.

```
//solución imperativa
double result = numero*numero*numero;
System.out.println("El cubo de "+ numero + " es " + result);
```

Por otro lado, la programación funcional trata de evitar la declaración y asignación de variables, que es la principal causa de bugs y errores en nuestros programas. Como se muestra en el ejemplo, en una línea resolvemos el problema con llamadas a funciones o métodos.

```
//Solución funcional

System.out.println("El cubo de "+ numero + " es " +
ProgramacionFuncionalEjemplo.potenciaCubo(numero));
```

Al mismo tiempo, los métodos en programación funcional intentan omitir la asignación de valores a la variable tanto como sea posible.

```
public static double potenciaCubo(double x) {

        return x*x*x;
    }
```

ProgramacionFuncionalEjemplo.java

```
package paradigmaprogramacionfuncional;
import java.util.Scanner;
```

```
public class ProgramacionFuncionalEjemplo {

	public static double potenciaCubo(double x) {

		return x*x*x;
	}

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);


		System.out.println("Escribir un número para calcular el cubo");

		double numero = sc.nextInt();

		//Imperative solution
		double result = numero*numero*numero;
		System.out.println("El cubo de "+ numero + " es " + result);


		//Declarative and functional solution
System.out.println("El cubo de "+ numero + " es " +
ProgramacionFuncionalEjemplo.potenciaCubo(numero));

	}

}
```

7.4 Programación funcional Java 8

La programación funcional en Java no ha sido fácil históricamente, e incluso había **varios aspectos de la programación funcional que** ni siquiera eran **realmente posibles** en Java. En **Java 8, Oracle hizo un esfuerzo para facilitar la programación funcional,** y este esfuerzo tuvo éxito hasta cierto **punto.** En estas **notas de programación** funcional **Java** repasaremos **los conceptos básicos de la programación funcional, y** qué partes de **ella son factibles en Java.**

7.4.1 Conceptos básicos de la programación funcional

La programación funcional incluye los siguientes conceptos clave:

1. Las funciones son objetos de primera clase
2. Funciones puras
3. Funciones de orden superior

La programación funcional pura también tiene **un conjunto de reglas** a seguir:

1. Sin estado
2. Sin efectos secundarios.
3. Variables inmutables
4. Favorecer la recursión sobre el bucle

Estos conceptos y reglas se explicarán en el resto de esta unidad.

Incluso si no sigue todas estas reglas todo el **tiempo, podemos beneficiarnos de las ideas de programación funcional** en nuestros **programas.** Desafortunadamente, **la programación funcional no es la herramienta adecuada** para todos los **problemas.** Especialmente la idea de "sin efectos secundarios" **hace que sea difícil,** por ejemplo, **escribir en una base de datos** (que es un efecto secundario). Pero **la java Stream API y otras cualidades de programación funcional** son para uso diario de **los programadores de Java.**

7.5 Las funciones son objetos de primera clase / miembros de primera clase del lenguaje

Una de las principales ventajas que Java 8 y la programación funcional aporta a la mesa es que las funciones se han convertido en objetos de primera clase o miembros del lenguaje a través de las interfaces funcionales. Algo que es bastante común en algunos de los lenguajes de programación como JavaScript, que es que las funciones se pueden almacenar en variables o se pueden pasar como parámetros, era imposible en versiones anteriores de Java.

En este extracto de Javascript estamos pasando la función myCallback como parámetro a la función addContact.

```
function addContact(id, callback) {
  callback();
}

function  myCallback() {
  alert('Hello World');
```

}

addContact (myCallback);

En el paradigma de programación funcional, las funciones son **objetos de primera clase en el lenguaje.** Esto significa que puede crear una **"instancia" de** una función, al igual que una referencia variable a esa instancia de función, así como una **referencia** a un String, un HashMap o cualquier otro objeto. Las funciones también se **pueden pasar como parámetros a otras funciones. En Java,** los métodos no son objetos de primera **clase.** Lo más cerca **que llega Java son las** interfaces funcionales.

Vamos a explicarlo con un ejemplo sencillo. En nuestra clase FunctionPrimeraClaseEjemplo, **tenemos una** formula de parámetro de interfaz funcional que aplica la **fórmula recibida, llamando al método apply. Observe que para ejecutar la función necesitamos llamar al método apply,** formula.apply(x); .

```
public Double funcionFormula(Double x, Function<Double,Double> formula) {


            return formula.apply(x);
      }
```

También tenemos dos funciones formulaCuadrado y formulaCubo. Estas dos funciones de firma Doble x, y tipo devuelto Doble, coinciden con la firma de la Interfaz Funcional `Function<Double,Double> formula`.

```
public Double formulaCuadrado(Double x) {
            return x*x;
      }
      public Double formulaCubo(Double x) {
            return x*x*x;
      }
```

Podemos crear un objeto de tipo `FuncionPrimeraClaseEjemplo`.

```
FuncionPrimeraClaseEjemplo objeto = new FuncionPrimeraClaseEjemplo();
```

Después de eso, podemos pasar al método functionFormula una función de este objeto u otro objeto de una clase diferente, `object::formulaCuadrado`. Para ello, utilizamos el operador :: , que permite a los programadores hacer referencia a las funciones.

En este caso, hacemos referencia a la función a través del objeto, y el sintax para esta expresión es:

object_variable_name::nombreMetodo

```
Double result = objeto.funcionFormula(numero, objeto::formulaCuadrado);
```

Por lo tanto, los interfaces funcionales proporcionan la oportunidad de pasar funciones como parámetro y también almacenarlas en una variable.

```
Function<Double,Double> formulaVar = objeto::formulaCuadrado;
```

Además, podemos pasar como parámetro una expresión lambda, lo que finalmente es una función anónima.

```
result = object.functionFormula(numero, x->x*x*x*x);
```

Finalmente, podemos pasar como parámetro un método estático, un método de clase. El sintax para los métodos estáticos es ligeramente diferente ya que usamos el nombre de la clase para hacer referencia a un método estático:

nombreClase::nombreMetodo

```
FunctionFirstClassExample:: formulaRaizCuadrada
```

```
result = objeto.funcionFormula(numero,
FuncionPrimeraClaseEjemplo::formulaRaizCuadrada);
```

FunctionFirstClassExample.java

```
package paradigmaprogramacionfuncional;

import java.util.Scanner;
import java.util.function.Function;


public class FuncionPrimeraClaseEjemplo{


	public Double funcionFormula(Double x, Function<Double,Double>
formula) {



		return formula.apply(x);
	}

	public Double formulaCuadrado(Double x) {

		return x*x;

	}

	public Double formulaCubo(Double x) {

		return x*x*x;

	}


public static Double formulaRaizCuadrada(Double x) {

		return Math.sqrt(x);


	}

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);

		System.out.println("Introduzca un número en la consola");

		Double numero = sc.nextDouble();
		FuncionPrimeraClaseEjemplo objeto = new
FuncionPrimeraClaseEjemplo();
```

```
            Double result = objeto.funcionFormula(numero,
objeto::formulaCuadrado);

            System.out.println("Calculamos el cuadrado del número  " +
result);

            Function<Double,Double> formulaVar = objeto::formulaCuadrado;

            result = objeto.funcionFormula(numero, objeto::formulaCubo);


            System.out.println("Calculamos el cubo del número: " + result);

            result = objeto.funcionFormula(numero, x->x*x*x*x);

            System.out.println("Calculamos la cuarta potencia del número: "
+ result);

            result = objeto.funcionFormula(numero,
FuncionPrimeraClaseEjemplo::formulaRaizCuadrada);

      System.out.println("Calculamos la raiz cuadrada del número: " +
result);

      }

}
```

Un ejemplo de ejecución:

```
25
Calculamos el cuadrado del número  625.0
Calculamos el cubo del número: 15625.0
Calculamos la cuarta potencia del número: 390625.0
Calculamos la raiz cuadrada del número: 5.0
```

7.6 Funciones puras, sin estado.

En programación, una función pura es una función que tiene las siguientes propiedades:

1. La función siempre devuelve el mismo valor para las mismas entradas.
2. La evaluación de la función no tiene efectos secundarios. Los efectos secundarios se refieren al cambio de otros atributos del programa no contenidos en la función, como el cambio de valores de variables globales o el uso de flujos de E/S (sin mutación de variables estáticas locales, variables no locales, argumentos de referencia mutables o flujos de entrada/salida).

Efectivamente, el valor devuelto de una función pura se basa solo en sus entradas y no tiene otras dependencias o efectos en el programa general.

Las funciones puras son conceptualmente similares a las funciones matemáticas. Para cualquier entrada dada, una función pura debe devolver exactamente un valor posible.

Sin embargo, al igual que una función matemática, se le permite devolver ese mismo valor para otras entradas. Además, al igual que una función matemática, su salida está determinada únicamente por sus entradas y no por ningún valor almacenado en algún otro estado global.

Un ejemplo

```
public class PuraFuncionClass{

   public int sum(int a, int b) {
      return a + b;
   }
}
```

Cada vez que llame a la función de suma devolverá el mismo valor, es decir, la evaluación de suma (5,7) será siempre 12, sin importar cuántas veces llame a esta función.

Observe cómo el valor devuelto de la función sum() **depende únicamente de los parámetros de entrada.** También tenga en cuenta **que el sum() no tiene** efectos secundarios, lo que significa **que no modifica ningún estado (variables) fuera de la función** o clase.

Lo contrario se puede encontrar en el siguiente ejemplo **de una función no pura:**

```
public class NoPuraFuncionExample{
   private int value = 1;

   public int sum(int nextValue) {
      this.value += nextValue;
      return this.value;
   }
```

Aquí, el efecto es el contrario. La primera vez que llame a esta función sum(3) devolvería 4. Siempre que después de esta llamada el valor de la propiedad cambie a 4, la segunda vez que llame a sum(3), el resultado devuelto de esta llamada de método sería 7. Al final, el valor devuelto de la función depende del valor de la propiedad del objeto, que cambia en cada llamada.

7.6.1 Función de orden superior

La idea detrás de las funciones de alto orden representa el empleo de funciones como argumentos. Para que una función sea una función de alto orden, debe cumplir la siguiente condición: Cualquier parámetro en la firma de la función debe ser una función, y el tipo de retorno debe ser una función del mismo modo. Ergo, el producto final es una función que manipula funciones y genera una función como resultado.

En Java, **lo más cerca que podemos llegar a una función de orden superior** es una función (método) que toma una o más interfaces **funcionales como** parámetros y **devuelve** una **interfaz funcional.** Aquí hay **un ejemplo de una función de orden superior en Java:**

In this **simpler example you can try a higher order function.**

```
public Supplier<Double> funcionOrdenSuperior(Supplier<Double> numero, Function<Double,Double> funcion)
```

HighOrderFunction toma como parámetro una **interface Consumer** que nos da un **número, una interfaz de función** y devuelve una **interfaz Supplier.** Lo que estamos haciendo básicamente es permitir que las expresiones o funciones lambda **se pasen como parámetro** y devuelvan una función o expresión lambda como **parámetro.**

```
public Supplier<Double> funcionOrdenSuperior(Supplier<Double> number, Function<Double,Double> function) {

                return ( ()->function.apply(number.get()));
```

```
        }
```

Los parámetros de la firma funcionOrdenSuperior son interfaces funcionales, lambdas o funciones. Asimismo, el tipo devuelto es un Supplier<Double> otra función. Se muestra en la instrucción return de este ejemplo, cómo estamos devolviendo una función, que es una lambda en este caso.

```
package introductionfunctionalprogrammin;

package paradigmaprogramacionfuncional;
import java.util.Scanner;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;

public class FuncionOrdenSuperiorEjemplo {


        public Supplier<Double> funcionOrdenSuperior(Supplier<Double>
number, Function<Double,Double> function) {


                return ( ()->function.apply(number.get()));

        }


          public static void main(String[] args) {


                Scanner miScanner = new Scanner(System.in);

                System.out.println("Introduce un número decimal");

                Double numero1 = miScanner.nextDouble();

                FuncionOrdenSuperiorEjemplo highOrder = new
FuncionOrdenSuperiorEjemplo() ;


                Supplier<Double> supplierNumerico=
highOrder.funcionOrdenSuperior(()-> numero1 ,(x)->x*x);

                System.out.println(" Función de orden superior que
retorna un supplier " + supplierNumerico.get());


                System.out.println(" Función de orden superior que
retorna un supplier directamente"
                +  highOrder.funcionOrdenSuperior(()-> numero1 ,(x)-
>x*x).get());

          }
```

```
	}
```

Debido a que este es un concepto absolutamente importante en la programación funcional, vamos a introducir otro ejemplo. Además, emplearemos algunos de los conceptos que hemos aprendido, como la composición de funciones y el :: operador.

El método componeFormulaSeleccionadaYRedondea recibe un argumento Function<Double,Double>. También devuelve una Función<Double,Doble>

```
public Function<Double,Double>
componeFormulaSeleccionadayRedondea(Function<Double,Double> function) {

		return function.andThen(FuncionOrdenSuperiorEjemplo2::redondea);


	}
```

El cálculo que realiza consiste en combinar la fórmula pasada como parámetro, una función, con la función redondear, mostrada en amarillo resaltado. Finalmente, llamamos a esta función en el programa principal.

```
Function<Double,Double> miFormula =
objectHighOrder.componeFormulaSeleccionadayRedondea(FuncionOrdenSuperiorEjemplo2::cuadrado);
```

La función cuadrado es un método estático que se adapta a la definición función<doble, doble>; Por lo tanto, podemos pasarlo como un parámetro

```
public static Double cuadrado(Double a) {


		return a*a;
	}
```

Como resultado, tenemos una función que calcula el cuadrado y redondea el resultado en la variable de tipo interfaz funcional Function<Double,Double> miFormula.

Podemos emplear esta función en cualquier lugar de nuestro código, e infinitas

veces. Es una función pura también; devolverá el mismo valor en cualquier llamada. Para invocar la función, utilizamos el método apply myFormula.apply(24.2).

```
System.out.println("La función obtenida de componer finalmente calcula el
cuadrado" +
                " y redondea el resultado del cuadrado: " +
miFormula.apply(24.2));


        }
```

El resultado de la ejecución es

```
función obtenida de componer finalmente calcula el cuadrado y redondea el
resultado del cuadrado: 586.0
```

FuncionOrdenSuperiorEjemplo2.java

```
package paradigmaprogramacionfuncional;
import java.util.function.Function;

public class FuncionOrdenSuperiorEjemplo2 {


      public Function<Double,Double>
componeFormulaSeleccionadayRedondea(Function<Double,Double> function) {

            return function.andThen(FuncionOrdenSuperiorEjemplo2::redondea);


      }

      public static Double  raizCuadrada(Double a) {

            return Math.sqrt(a);
      }
public static Double cuadrado(Double a) {

            return a*a;
      }

public static double redondea(Double a) {
```

```
            return (double) Math.round(a);

}

        public static void main(String[] args) {

            FuncionOrdenSuperiorEjemplo2 objectHighOrder = new
FuncionOrdenSuperiorEjemplo2();

            Function<Double,Double> miFormula =
objectHighOrder.componeFormulaSeleccionadayRedondea(FuncionOrdenSuperiorEjemp
lo2::cuadrado);

            System.out.println("La función obtenida de componer finalmente
calcula el cuadrado" +
            " y redondea el resultado del cuadrado: " +
miFormula.apply(24.2));

        }
}
```

Ejercicio

Añadir al ejemplo anterior miFormula2 de manera que el método resultado realice la raíz cuadrada y luego redondee, usando el método raizCuadrada.

```
Function<Double,Double> miFormula2
```

Añadir al ejemplo anterior miFormula3 de manera que el método resultado realice el cubo y luego redondee, pasando una función lambda que calcule el cubo.

```
Function<Double,Double> miFormula3
```

7.6.2 Sin estado

Como **se mencionó al principio de estos capítulos,** una regla del paradigma de programación funcional es no tener **estado.** "Ningún estado" o "apátrida" generalmente es entendido por cualquier estado fuera de la función. Una

función **puede tener variables locales** que contienen el estado temporal **internamente,** pero la función no debe hacer referencia a **ninguna variable miembro de la clase u objeto** al que pertenece la función. No **debe tener** acceso a **valores externos.**

Example of a function that does not use any external state:

```
public class Calculator{
   public int sum(int a, int b) {
      return a + b;
   }
}
```

Por el contrario, en la **siguiente versión** del método sum **el problema** es que el **método** depende del estado de **la calculadora** en un momento determinado, **ya que su cálculo radicaba en el valor initVal.**

```
public class Calculator {
   private int initVal = 5;

public setInitVal(int val) {

initVal= val;
}
   public int sum(int a) {
      return initVal + a;
   }
}
```

Este tipo de prácticas de programación pueden conducir a errores y errores en sucódigo. Sin embargo, dado que Java es un lenguaje de programación orientado a objetos, tenemos que confiar en el estado de un objeto para hacer algunos cálculos. El camino a seguir debe ser tratar de encontrar un equilibrio o equilibrio entre la programación funcional y la orientada a objetos. Por lo tanto, en algún contexto, cuando es necesario, aplicamos conceptos del Diseño Orientado a Objetos. Debería ser obligatorio en cualquier escenario de programación donde necesitemos representar datos o comportamiento, es decir, hacer abstracciones.

7.6.3 Sin efectos secundarios

Otra regla en el paradigma de la programación funcional es la de no tener efectos secundarios. Esto significa que una función no puede cambiar ningún estado fuera de la función. Cambiar el estado fuera de una función se conoce como un *efecto secundario*.

Se deduce del ejemplo anterior. Siempre que mi **estado** no cambie, es poco probable que ocurra **un resultado no deseado** en las ejecuciones de funciones. Es una regla en el paradigma de la programación funcional. Esto significa **que una función no puede cambiar ningún estado** fuera de la función. **Cambiar el estado del objeto externo en una función se** conoce como efecto **secundario.**

El estado fuera de una **función se refiere** tanto a las variables miembro de la clase u objeto **de función,** como **a las variables miembro dentro de los parámetros de las funciones o** el estado en sistemas externos, como sistemas de archivos o bases de **datos.**

```
public class Calculator{
   public int sum(int a, int b) {
      return a + b;
   }
}
```

El valor devuelto de la suma de llamadas (5,7) sería siempre 12. En consecuencia, es un método de programación funcional correcto. No reproduce ni causa errores.

```
public class Calculador {
   private int initVal = 5;

public setInitVal(int val) {
initVal= val;
}
   public int sum(int a) {
      return initVal + a;
   }
}
```

En este segundo escenario, dado que las funciones de suma se basan en el valor de la propiedad initVal, el estado del objeto, podría llevar al programa a errores indeseables.

7.6.4 Objetos inmutables

La definición que puedes ver en Wikipedia establece que un objeto inmutable es aquel cuyo estado no se puede modificar después de que se crea. O en otras palabras, alguna variable que no puede cambiar su valor.

Un objeto inmutable es un objeto cuyo estado no se puede modificar después

de crearlo. En aplicaciones multihilo eso es una gran cosa. Permite que un hilo actúe sobre los datos representados por objetos inmutables sin preocuparse por lo que otros hilos están haciendo.

Es muy común en aplicaciones de tamaño grande y mediano que múltiples programas hagan referencia a la misma variable. Este es el concepto introducido anteriormente, multithreading o multiprocesamiento. Hasta cierto punto, estamos tratando de evitar el efecto secundario de algunos procesos o hilos que podrían modificar el mismo objeto en un período de tiempo.

En Java, es bastante simple crear un objeto inmutable. La forma de proceder es declarar una clase con propiedades privadas, getters y no setters. Los valores de propiedad se fijan o se asignan en el constructor. Una vez creado el objeto, es imposible modificar su estado.

En el siguiente programa, la clase `ObjetoInmutablePersona` posee propiedades privadas. Además, no se declara ningún método set para estas propiedades. Una vez que el programa crea el objeto, queda claro que no es posible modificar sus propiedades.

```
            ObjetoInmutablePersona person = new
ObjetoInmutablePersona("John","Doe", 40);
```

Sin embargo, las propiedades se pueden evaluar desde getters, y eso da como resultado que el programa tenga acceso completo al estado del objeto.

```
System.out.println("Nombre completo de la persona:"+ person.getNombre() + " "
+ person.getApellidos());
```

InmutableObjectPerson.java

```
package paradigmaprogramacionfuncional;

public class ObjetoInmutablePersona {

	private String nombre="";
	private String apellidos="";
	private int edad=0;


	public ObjetoInmutablePersona() {

	}

	public ObjetoInmutablePersona(String nombre, String apellidos, int
edad) {
		super();
		this.nombre = nombre;
```

```
		this.apellidos = apellidos;
		this.edad = edad;
	}

	public String getNombre() {
		return nombre;
	}

	public String getApellidos() {
		return apellidos;
	}

	public int getEdad() {
		return edad;
	}




	@Override
	public String toString() {
		return "ObjetoInmutablePersona [nombre=" + nombre + ",
lastName=" + apellidos + ", age=" + edad + "]";
	}

	public static void main(String[] args) {


		ObjetoInmutablePersona person = new
ObjetoInmutablePersona("John","Doe", 40);

		System.out.println("Nombre completo de la persona:"+
person.getNombre() + " " + person.getApellidos());



	}
}
```

7.6.5 Favorecer la recursividad sobre los bucles

Una cuarta regla en el paradigma de programación funcional es favorecer la recursividad sobre el bucle. La recursión utiliza llamadas a funciones para lograr bucles, por lo que el código se vuelve más funcional.

Otra alternativa a los bucles es la API de Java Streams. Esta API está inspirada funcionalmente. Nos vemos más adelante en el curso.

El principio de recursividad es la capacidad de resolver problemas de computación con una función que se llama a sí misma. En la siguiente sección, analizaremos ampliamente la recursividad.

8 Recursividad

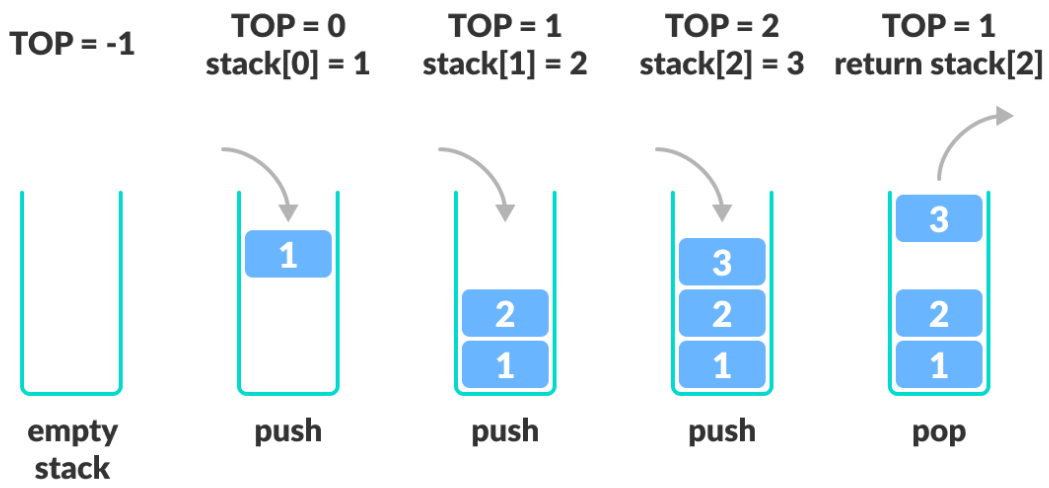
Una de las **herramientas de programación que se hace necesaria en la programación funcional** es el concepto de **recursividad**. En general, la recursividad es el proceso de **definir algo en términos de sí mismo y es algo similar a una definición circular**. El componente clave de un método recursivo es una declaración que ejecuta una llamada a sí mismo. La recursividad es un poderoso mecanismo de control.

El **proceso en el que una función se llama directa o indirectamente** se denomina **recursividad y la función correspondiente se llama función recursiva**. Usando un **algoritmo recursivo**, ciertos problemas **se pueden resolver con bastante facilidad.** Ejemplos de tales problemas son **Torres de Hanoi (TOH), recorridos** de **árboles**, y **algoritmos de búsqueda como quickshort**.

8.1 La pila de ejecución

8.1.1 ¿Qué es una pila?

Una **pila es una estructura de datos en java** que va colocando elementos uno encima de otros. Cuando hacemos **una operación push(), colocamos un elemento arriba**. Cuando **hacemos un pop(),** sacamos el elemento que **está arriba del todo.**



8.2 La Pila de ejecución en java

Cada vez **que llamamos a una función o método en uno de nuestros programas** desde una **clase o programa principal ese método va a tomar el control de la ejecución** en nuestro programa. Para **guardar su ejecución y el estado de todas sus variables y parámetros locale**s, **Java implementa**, al igual que otros compiladores, **una pila de ejecución o Runtime Stack** de tamaño limitado, para cada programa.
Para cada **proceso la máquina virtual java, JVM (Java Virtual Machine**) **creará una pila en tiempo de ejecución**. Todos y **cada uno de los métodos se insertarán en la pila**. Cada entrada **se denomina Registro de activación de marco de pila**.
Cada vez **que llamamos a un método o función, ese método o función Java crea una entrada en la pila**. Este mecanismo **se usa para saber en que ejecución** y **quien tiene el control de la ejecución de nuestro programa** en cada momento. Para **cada llamada** se **guarda en memoria una copia** de **sus variables locales y parámetros.**

En el ejemplo siguiente se ve de manera bastante clara. El primer elemento en ser colocado en la pila de ejecución es mi función main() del programa principal, es quien toma el control el (1**). Main() llama a first_func(), se hace una operación push en la pila** de **first_func()** y se coloca en la parte superior de la pila. El **método first_func() llama a second_func()** que toma el **control de la ejecución** y se coloca en la parte de arriba de la pila. Como veis el método que está en **la posicion superior de la pila es quien se esta ejecutando en cada momento,** tiene el control de la ejecución. **Cuando second_func() termina se hace un pop de second_func(),** se saca de la pila. El control vuelve a first_func(). La misma operación se hace cuando first_func() termina, se saca de la pila. Finalmente main() recupera el control. Termina de ejecutarse, y sale de la pila.

Una vez finalizada, **Stack Frame se vaciará y la pila vacía será destruida** por la máquina virtual java, justo antes de la terminación del programa o hilo de ejecución.

(1)	(2)	(3)	(4)	(5)	(6)	(7)
*(Runtime Stack for Main Thread)						(*Empty Stack)
			second_func()			
		first_func()	first_func()	first_func()		
	main()	main()	main()	main()	main()	

```
/**
 *
 */
package com.test.java;

/**

 */
public class MecanismoEjecucionPilaJava {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		first_func();
	}

	public static void first_func() {
		second_func();
	}

	public static void second_func() {
		System.out.println("Hello World");
	}

}
```

8.3 Recursividad

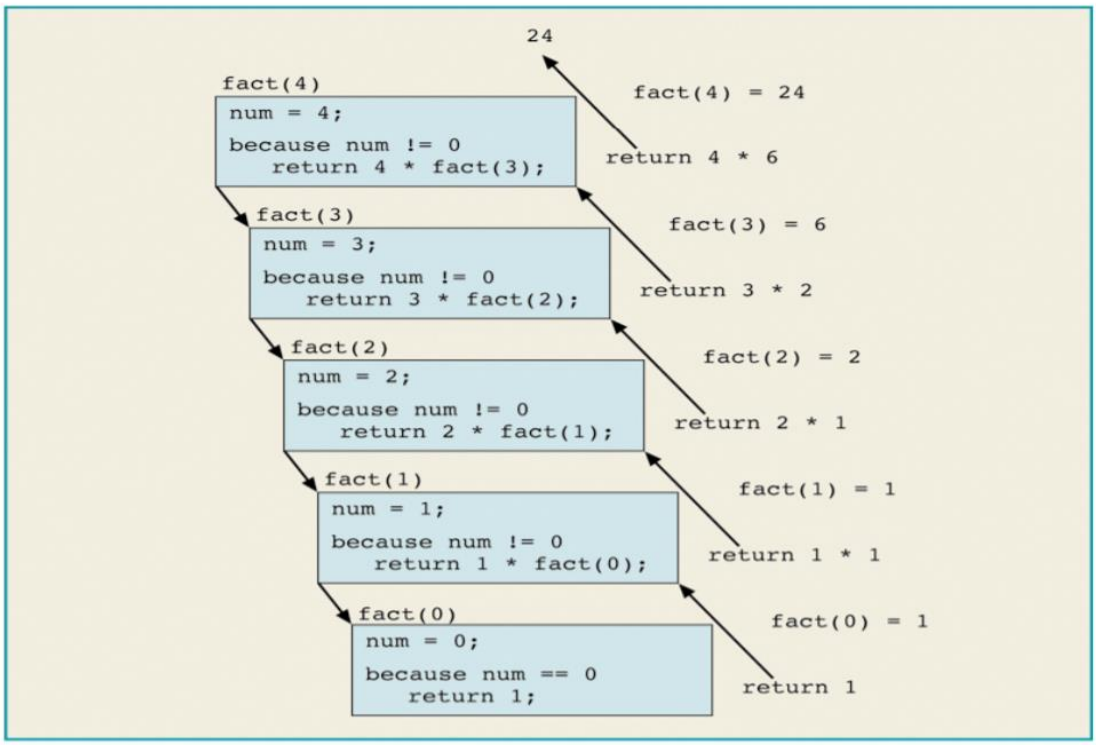
¿Cuál es condición base en recursividad?

En el **programa recursivo**, se proporciona **la solución al caso base** y la **solución del problema más grande** se expresa en **términos de problemas más pequeños**.

```
public static Long factorial(long n)
	{
	   if (n <= 1) // base case
	      return 1;
	   else
	      return n*factorial(n-1);
	}
```

En el ejemplo anterior, **se define el caso base para n <= 1** y se puede resolver el problema **para un número mayor** con la **división del problema a números más pequeños** hasta que se alcance el caso base, factorial(1) que devueve 1.

En la solución anterior si **la aplicamos para el factorial(4**) la función nos devolvería **4*factoria(3).** Ahora **factorial(4) espera a que factorial(3) se resuelva**. Estamos en la **ejecución de factorial(3)**. La función **factorial(3) devuelve 3*factorial(2**). En este punto, factorial(3) espera a que se resuelva factorial(2), **factorial(2)** nos devolvería **2*factorial(1).** Es el mismo caso que el anterior, **factorial(2) debe esperar** a que **factorial(1) termine para devolver el resultado**.
En este punto hemos llegado **al caso base, factorial(1).** Cuando la llamada a factorial(1) termine, automáticamente **devolverá un 1**. Ahora tenemos que hacer el **recorrido inverso**. Factorial(2) esta esperando a que factorial(1) termine, cuando termina, **factorial(2) devolverá 2*1**. Ahora, **factorial(3) puede resolverse** recibe el resultado de factorial 2, un 2 y puede continuar, devolviendo el resultado **3*2**. Y finalmente, **factorial(4) ya tiene le resultado de factorial 3**, que es 6 y **termina su ejecución devolviendo 4*6, 24** Es un **enfoque diferente para resolver el problema**, que tradicionalmente resolveríamos con un bucle.



Factorial.java

```
package paradigmaprogramacionfuncional;

import java.util.Scanner;

public class Factorial {

	public static Long factorial(long n)
	{
	    if (n <= 1) // caso base
```

```
            return Long.valueOf(1);
        else
            return n*factorial(n-1);
    }


    public static void main(String[] args) {


        Scanner miScanner = new Scanner(System.in);


        System.out.println("Escriba un número entero");
        Long numero1 = miScanner.nextLong();

        Long resultado = Factorial.factorial(numero1);



        System.out.println("El resultado del factorial de " + numero1 + "
es: "+ resultado);


    }

}
```

¿Por qué se produce un error de desbordamiento de pila en la recursividad? Si **el caso base no se alcanza o no está definido**, puede surgir el problema de **desbordamiento de pila.** Tomemos un ejemplo para entender esto. Si no hay caso base, no podemos **hacer la vuelta o marcha atrás**. Volver de **factorial(1)** y la **función** se quedaría **infinitamente llamándose a sí misma**, hasta que el **desborde el tamaño de la pila de ejecución java**. Si la **pila tiene un millón de entradas disponibles**, en la **llamada un millon y uno a factorial**, la pila se desborda. Se produce **la excepción stackoverflow** en Java.

```
Long factorial(Long n)
{
   // Puede causar overflow
   //
   if (n == 100)
      return Long.valueOf(1);

   else
      return n*fact(n-1);
}
```

Si se **llama a fact(10), llamará a fact(9), fact(8), fact(7**) y así sucesivamente, pero el **número nunca llegará a 100**. Por lo tanto, **el caso base no se alcanza.** Si estas funciones agotan la memoria en la pila, **se producirá un error de desbordamiento de pila**.

¿Cuál es la diferencia entre la recursividad directa e indirecta?

Una función se denomina recursiva directa si llama a la misma función. Una función se denomina recursiva indirecta indirectaRecFun1 si llama **a otra función nueva** indirectaRecFun2. La función indirectaRecFun2 volverá a llamar a indirectaRecFun1. Y así **hasta que alguna de ellas alcance** el **caso base** como en una partida de ping pong, **ida y vuelta de una función a otra**. La diferencia entre recursividad directa e indirecta se ha ilustrado en el Cuadro 1.

Recursividad directa

```
void directaRecFun()
{
   // Some code....

   directaRecFun();

   // Some code...
}
```

Recursividad Indirecta

```
void indirectaRecFun1()
{
   // codigo...

   indirectaRecFun2();

   // codigo...
}

void indirectaRecFun2()
{
   // código...

   indirectaRecFun1();

   // código
}
```

Como veis en la recursividad indirecta, tenemos dos funciones. Una se llama a la otra a la vez que se hace el problema más pequeño hasta llegar al caso base en una de ellas.

¿Cuál es la diferencia entre la recursividad de cola y la de cabecera?

Una función recursiva es recursiva de cola cuando la **llamada recursiva es la**

última instrucción ejecutada por la función.

Por ejemplo, en el ejemplo **anterior lo último que hace factorial es llamada recursiva:**

return n*factorial(n-1). De cola

El ejemplo siguiente la recursividad es de cabecera. Porque **se hace la llamada recursiva**, y después **se ejecutan más instrucciones en nuestro código.**

```
// Instruccion 2
printFuncion(test - 1);

System.out.printf("%d ", test);
return;
```

¿Cómo se asigna la memoria a diferentes llamadas de función en recursividad?

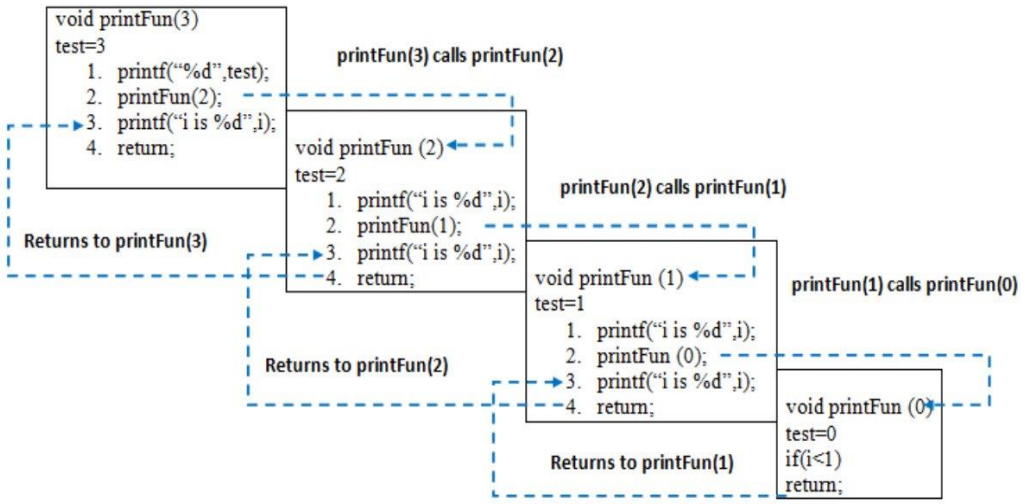
Cuando se llama a cualquier función desde main(), se le **asigna memoria en la pila. Una función recursiva se llama a sí misma**, la **memoria de la función llamada se asigna encima de la memoria asignada a la función de llamada** y **se crea una copia diferente de las variables locales para cada llamada de función.** Cuando se alcanza el caso base, **la función devuelve su valor a la función por la que se llama** y se desea asignar memoria y el proceso continúa.

Tomemos el ejemplo de cómo funciona la recursividad tomando una función simple.

```
package paradigmaprogramacionfuncional;
public class ImprimirRecursivo {
    static void printFun(int test)
    {
        if (test < 1)
            return;

        else {
            System.out.printf("%d ", test);


            printFun(test - 1);

            System.out.printf("%d ", test);
            return;
        }
    }

    public static void main(String[] args)
    {
        int test = 3;
        printFun(test);
    }
}
```

Cuando se llama a **printFun(3)** desde main(), la memoria se asigna a **printFun(3)** y una la variable local test se inicializa en 3 y la instrucción 1 a 4 se inserta en la pila como se muestra en el diagrama siguiente. Primero imprime '3'. En la instrucción 2, se llama a **printFun(2)** y se asigna memoria a **printFun(2)** y variable local test se inicializa en 2 y la instrucción 1 a 4 se inserta en la pila. De forma similar,

printFun(2) llama a **printFun(1)** y **printFun(1)** llama a **printFun(0)**. **printFun(0)** va a la instrucción if y vuelve a **printFun(1)**. Las instrucciones restantes de **printFun(1)** se ejecutan y vuelve a **printFun(2)** y así sucesivamente. En la salida, se imprime el valor de 3 a 1 y, a continuación, se imprimen de 1 a 3. La pila de memoria se muestra en la imagen siguiente.



¿Cuáles son las desventajas de la programación recursiva sobre la programación iterativa?

Tenga en cuenta que **tanto los programas recursivos como los iterativos tienen la misma capacidad de resolución de problemas**, es decir, **cada programa recursivo se puede escribir de forma iterativa y viceversa** también es cierto. **El programa recursivo tiene mayores requisitos de espacio** que **el programa iterativo**, ya que **todas las funciones permanecerán en la pila** hasta que **se alcance el caso base**. También **tiene mayores requisitos de tiempo** debido a las llamadas de función se meten y sacan de la pila. **Se pierde tiempo** en esto, y **se ocupa más memoria**.

¿Cuáles son las ventajas de la programación recursiva sobre la programación iterativa?

La **recursividad proporciona una forma limpia y sencilla de escribir código**. Algunos problemas **son inherentemente recursivos** como **recorridos de árboles, Torre de Hanoi, etc.** Para estos problemas, **se prefiere escribir**

código recursivo. Podemos escribir **estos códigos también iterativamente** con la ayuda de **una estructura de datos de pila**.

8.4 Ejercicios.

1. Calcular la serie de fibonnaci de manera recursiva.

   La sucesión de Fibonacci es conocida desde hace miles de años, pero fue Fibonacci (Leonardo de Pisa) quien la dio a conocer al utilizarla para resolver un problema.
   El **primer** y **segundo** término de la sucesión son
   $$a_0 = 0$$
   $$a_1 = 1$$
   Los siguientes términos se obtienen sumando los dos términos que les preceden:
   El **tercer término** de la sucesión es
   $$a_2 = a_0 + a_1 = \\ = 0 + 1 = 1$$
   El **cuarto término** es
   $$a_3 = a_1 + a_2 = \\ = 1 + 1 = 2$$
   El **quinto término** es
   $$a_4 = a_2 + a_3 = \\ = 1 + 2 = 3$$
   El **sexto término** es
   $$a_5 = a_3 + a_4 = \\ = 2 + 3 = 5$$

   El $(n+1)$**-ésimo término** es.
   $$a_n = a_{n-2} + a_{n-1}$$

2. **Para calcular el máximo común divisor de dos números** enteros puedo aplicar **el algoritmo de Euclides**, que consiste **en ir restando el más pequeño del más grande** hasta que **queden dos números iguales**, que serán el máximo común divisor de los dos números. Si comenzamos con el par de números 412 y 184, tendríamos:

412	228	44	44	44	44	44	36	28	20	12	8	**4**
184	184	184	140	96	52	8	8	8	8	8	4	**4**

   Es decir, m.c.d.(412, 184)=4

Nota: Todavía es pronto para introducir **el tema de recursividad con expresiones lambda** lo **haremos en temas posteriores**, porque es un poco más complejo.

9 Bibliografía y referencias web

Referencias web

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021

Entornos de Desarrollo, Maria Jesus Ramos Martín, Garceta 2ª Edición, 2020

PROGRAMACIÓN EN LENGUAJES ESTRUCTURADOS, Enrique Quero Catalinas y José López Herranz, Paraninfo, 1997.