¿QUÉ ES UN PATRÓN?

Veremos más adelante de forma más formal que es un patrón, por el momento, basta con quedarse con la idea de que para muchos "tipos de problemas" que se repiten una y otra vez hay publicadas "plantillas" que son una estructura de diseño de clases para enfrentarse a la solución de dicho problema.

Para que valores la importancia del concepto de patrón puedes
Leer está página
https://openwebinars.net/blog/que-son-los-patrones-de-diseno/

O ver este video
https://youtu.be/6BHOeDL8vls
Fíjate en la primera parte del video donde explica que es un patrón y la importancia de los patrones. Cuando comience a explicar patrones concretos puedes saltar a la conclusión del video en el minuto 11:45.

EL PROBLEMA DE LA CREACIÓN DE OBJETOS.

La creación de un objeto puede ser algo complicado, no siempre se soluciona con un simple new clase(). Si es muy complejo, lo mejor es que ese trabajo pesado lo haga un método o una clase especializada. A medida que avance el curso observarás que cada vez te encontrarás con más clases de las que **no** creamos sus objetos directamente con new, un ejemplo de una clase que ya utilizamos es Console

```
Console miConsola=System.console();
```

El método System.console() no hace simplemente un new, hace más cosas como por ejemplo consultar que tipo de consola hay en el sistema y crear un objeto acorde con ella, por eso "mágicamente" no tenemos problemas de codificación en la consola windows cuando usamos System.console()

Hay varios patrones que se dedican a resolver problemas que surgen al crear objetos. Son los patrones "creacionales", 3 de ellos llevan la palabra "factory" en su nombre:

- Simple Factory Pattern
- Factory Method Pattern
- Abstract Factory Pattern

PATRÓN SIMPLE FACTORY

De todos los patrones anteriores el más sencillo es SIMPLE FACTORY cuyo objetivo es que exista un método, que llamaremos método factoría que facilite la creación de objetos.

Simply factory tiene varias variantes y de todas ellas la más sencilla consiste simplemente en añadir un método static a la cabeza de una jerarquía. Este método static es el que en función de los parámetros que recibe u otras cuestiones decide qué tipo de objeto crear. A estos métodos que se encargan de la creación de objetos se llaman métodos factoría ya se comportan como una factoría(fábrica) de objetos. Al método podemos llamarle por ejemplo típicamente *getInstance()* o create() o bien con cualquier nombre que consideremos apropiado.

¿Cuándo usar Simple Factory?
Un contexto fácil de observar es en una jerarquía que al usarla desde un cliente observamos la creación de objetos obliga al cliente a conocer a fondo dicha jerarquía o que se produce código duplicado.

Veamos un ejemplo, con una una jerarquía Triangulo. La clase cliente App quiere crear un triángulo

```
abstract class Triangulo{
    private int ladoA;
    private int ladoB;
    private int ladoC;


    protected Triangulo(int ladoA, int ladoB, int ladoC) {
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }


    abstract String getDescripcion();
    //otros metods abstactos como getSuperficie(), dibujate()...
}
class Equilatero extends Triangulo{
    public Equilatero(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }


    @Override
    public String getDescripcion(){
        return "soy Equilatero";
    }
}
class Escaleno extends Triangulo{
     public Escaleno(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
    public String getDescripcion(){
        return "soy Escaleno";


    }
}
class Isosceles extends Triangulo{
     public Isosceles(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
```

```
    public String getDescripcion(){
        return "soy Isosceles";

    }
}

public class App {
    public static void main(String[] args) {
        //antes de hacer nada creamos un triangulo
        int lado1=3;
        int lado2=3;
        int lado3=3;
        //creo un triangulo
        if(lado1==lado2 && lado1==lado3)
            new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
             new Escaleno(lado1,lado2,lado3);
        else
             new Isosceles(lado1,lado2,lado3);
    }
}
```

Cada vez que App necesita crear un Triangulo tiene que decidirse que tipo de triángulo crear y en base a eso, ejecutar una serie de if.
Si App quiere crear otro triangulo aparece código duplicado. Es evidente que “algo va mal”.

```
public class App {
    public static void main(String[] args) {
        //antes de hacer nada creamos un triangulo
        int lado1=3;
        int lado2=3;
        int lado3=3;
        //creo un triangulo
       Triangulo t= null;
        if(lado1==lado2 && lado1==lado3)
            t=new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
             t=new Escaleno(lado1,lado2,lado3);
        else
             t=new Isosceles(lado1,lado2,lado3);
        //creo otro  triangulo
        lado1=3;
        lado2=6;
        lado3=3;
        if(lado1==lado2 && lado1==lado3)
            t=new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
             t=new Escaleno(lado1,lado2,lado3);
        else
            t=new Isosceles(lado1,lado2,lado3);
    }
}
```

como programador de App se me ocurre escribir un método que centralice todo el código duplicado relativo a la creación de triángulos:

```
public class App {
    public static Triangulo crearTriangulo(int lado1, int lado2, int lado3){
```

```
        Triangulo t= null;
        if(lado1==lado2 && lado1==lado3)
           t=new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
            t= new Escaleno(lado1,lado2,lado3);
        else
            t=new Isosceles(lado1,lado2,lado3);
        return t;
    }
    public static void main(String[] args) {
        //antes de hacer nada creamos un triangulo
        int lado1=3;
        int lado2=3;
        int lado3=3;
        //creo un triangulo
        Triangulo t1=crearTriangulo(lado1, lado2, lado3);
        //creo otro  triangulo
        lado1=3;
        lado2=6;
        lado3=3;
        Triangulo t2=crearTriangulo(lado1, lado2, lado3);

    }
}
```

Pero observa que todos los programadores que desean utilizar la jerarquía triángulo llegan a la misma conclusión, hacer un método que simplifique la creación de triángulos, Y POR TANTO,¿no debería ser este método una prestación de la propia jerarquía Triangulo? Además, ¿quien mejor que la jerarquía Triangulo puede tomar una decisión de qué tipo de triángulo se debe crear?. Observa que en el constructor de Triángulo esta decisión no puede tomarse ya que cuando se invoca a un constructor ¡ya se creó un triángulo!.

SOLUCIÓN: CREAR UN MÉTODO FACTORÍA

```
abstract class Triangulo{
    private int ladoA;
    private int ladoB;
    private int ladoC;

    static Triangulo getInstance(int lado1,int lado2, int lado3){
        if(lado1==lado2 && lado1==lado3)
            return new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
             return new Escaleno(lado1,lado2,lado3);
        else
             return new Isosceles(lado1,lado2,lado3);
    }
    protected  Triangulo(int ladoA, int ladoB, int ladoC) {
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;

    }

    abstract String getDescripcion();
    //otros metods abstractos como getSuperficie(), dibujate()...
```

```
}
class Equilatero extends Triangulo{
    public Equilatero(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }

    @Override
    public String getDescripcion(){
        return "soy Equilatero";
    }
}
class Escaleno extends Triangulo{
     public Escaleno(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
    public String getDescripcion(){
        return "soy Escaleno";

    }
}
class Isosceles extends Triangulo{
     public Isosceles(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
    public String getDescripcion(){
        return "soy Isosceles";

    }
}

public class App {
    public static void main(String[] args) {
        //antes de hacer nada creamos un triangulo
        int lado1=3;
        int lado2=3;
        int lado3=3;
        //creo un triangulo
        Triangulo t=Triangulo.getInstance(lado1, lado2, lado3);
        System.out.println(t.getDescripcion());
       //t= new Triangulo();//ERROR

    }
}
```

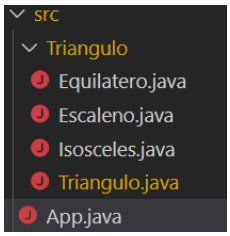
fíjate como se simplifica el código de la clase cliente(**Y DE TODAS LAS CLASES CLIENTES QUE UTILICEN LA CLASE TRIÁNGULO**) .

Los constructores de la clases abstractas a menudo son protected

Veamos por qué analizando el constructor de la clase Triángulo, el constructor de Triangulo debe ser protected ya que:

- Necesito un constructor aunque sea una clase Abstracta ya que es dicho constructor es invocado por super()el que se va encargar de las inicializaciones comunes a todo tipo de triángulos.
- El constructor no puede ser private ya que que si no super() falla. Prueba en el ejemplo anterior a poner private el constructor para comprobar esto.
- El constructor no puede ser public, no tiene sentido pues la clase es abstracta y genera confusión pero en cualquier caso no se podría usar ya que daría error el new Triangulo()
- **Por lo tanto, sólo puede ser protected.**

Ejercicio U5_B4B_E1: Retoca el ejemplo anterior, trabajando ahora la jerarquía y App en paquetes separados. Pon los public estrictamente necesarios.



SIMPLY FACTORY USANDO AHORA UNA CLASE FACTORÍA

Si en lugar de agregar el método factoría en la cabeza de la jerarquía Triangulo lo hacemos en una clase diferente que llamaremos clase factoría, mejoramos nuestro diseño. En una App grande al crear esta nueva clase mejorará el mantenimiento de toda la estructura. Recuerda el principio solid de responsabilidad única, me indica en este caso, que si es tan complicado crear objetos no se debe cargar a la clase Triangulo con esta nueva responsabilidad.

Los cambios son mínimos,simplemente movimos el getInstance() a una clase FactoriaTriangulos y el cliente ahora usa esta nueva clase para crear Triangulos. Ahora la clase Triangulo es más simple y legible y si queremos mantener la creación de objetos nos vamos directamente a la clase factoría donde de forma concreta nos ocupamos de ese problema.

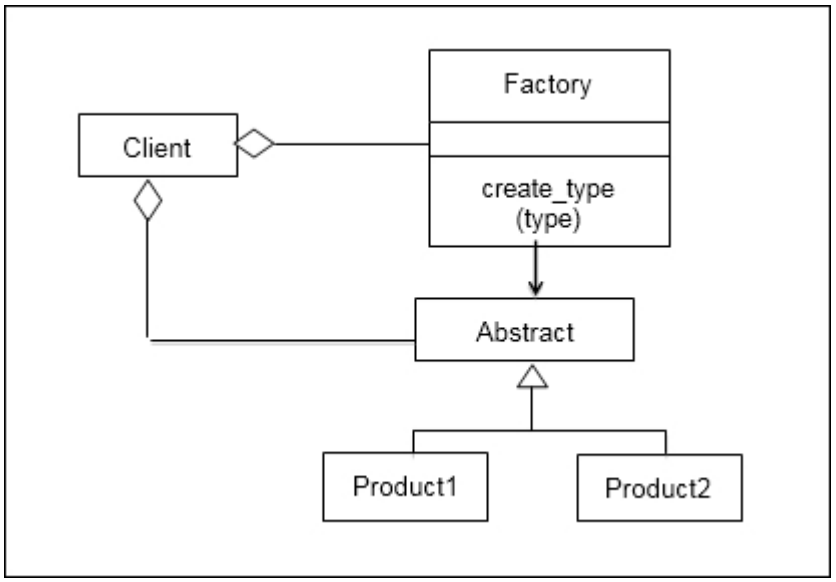
```
class FactoriaTriangulos{
    static Triangulo getInstance(int lado1,int lado2, int lado3){
        if(lado1==lado2 && lado1==lado3)
            return new Equilatero(lado1,lado2,lado3);
        else if (lado1!=lado2 && lado1!=lado3 && lado2!=lado3)
            return new Escaleno(lado1,lado2,lado3);
        else
            return new Isosceles(lado1,lado2,lado3);
    }
}
abstract class Triangulo{
    private int ladoA;
    private int ladoB;
```

```
    private int ladoC;


    protected  Triangulo(int ladoA, int ladoB, int ladoC) {
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;

    }

    abstract String getDescripcion();
    //otros metods abstactos como getSuperficie(), dibujate()...
}
class Equilatero extends Triangulo{
    public Equilatero(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }

    @Override
    public String getDescripcion(){
        return "soy Equilatero";
    }
}
class Escaleno extends Triangulo{
     public Escaleno(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
    public String getDescripcion(){
        return "soy Escaleno";

    }
}
class Isosceles extends Triangulo{
     public Isosceles(int ladoA, int ladoB, int ladoC) {
       super(ladoA,ladoB,ladoC);
    }
    @Override
    public String getDescripcion(){
        return "soy Isosceles";

    }
}

public class App {
    public static void main(String[] args) {
        //antes de hacer nada creamos un triangulo
        int lado1=3;
        int lado2=3;
        int lado3=3;
        //creo un triangulo
        Triangulo t=FactoriaTriangulos.getInstance(lado1, lado2, lado3);
        System.out.println(t.getDescripcion());
       //t= new Triangulo();//ERROR
```

```
    }
}
```

DESCRIPCIÓN DE SIMPLE FACTORY CON UML

El cliente para crear objetos o productos concretos de una jerarquía utiliza una clase Factoría que es la que contiene un método factoría llamado create_type() en el diagrama..



Ejercicio U5_B4B_E2:

Observa la siguiente jerarquía y la clase App que la usa. Falta por escribir la clase *AnimalFactory* que es la clase que implementa el patrón Simple Factory. Tiene un método *createAnimal()* que crea un nuevo objeto Animal basado en el string que se pasa como parámetro. Si es "cat", "dog" o "duck", crea una nueva instancia de la clase correspondiente.

```
abstract class Animal {
 public abstract void makeSound();
}

class Cat extends Animal {
 @Override
 public void makeSound() {
   System.out.println("Meow");
 }
}

class Dog extends Animal {
 @Override
 public void makeSound() {
   System.out.println("Woof");
 }
}

class Duck extends Animal {
 @Override
 public void makeSound() {
   System.out.println("Quack");
```

```
 }
}

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


   Animal cat = AnimalFactory.createAnimal("cat");
   cat.makeSound(); // Output: Meow

   Animal dog = AnimalFactory.createAnimal("dog");
   dog.makeSound(); // Output: Woof

   Animal duck = AnimalFactory.createAnimal("duck");
   duck.makeSound(); // Output: Quack

 }
}
```

Si la creación de objetos se complica

La clase `FactoriaTriangulos` anterior en un caso real se puede quedar corta. Necesitaremos no sólo una clase si no estructuras más complejas con subclases implicadas en la creación de objetos y debemos usar patrones más elaborados que aquí simplemente nombramos: Factory Method Pattern y Abstract Factory Pattern