



Backend Developer - My Roadmap @dnielpy

Github: <https://github.com/dnielpy>

Este roadmap esta inspirado en: <https://roadmap.sh/backend>

Roadmap Backend Developer 2024

- Java ✓
- Version Control System ✓
- Repo hosting service ✓
- Relational database
- Spring
- APIS
- Testing
- CI/CD
- Containerization vs Virtualization
- Web Servers

Java

Fuentes:

- https://www.youtube.com/@programando_en_java
- <https://github.com/practical-tutorials/project-based-learning?tab=readme-ov-file#c>

Temas Principales:

- *Los primeros temas del lenguaje los estudié antes de crear este Roadmap. Puedes checkearlo de todas formas en mi github: <https://github.com/dnielpy/CodePractice>*
- **Aspectos fundamentales de la Sintaxis de Java:**
 - **Static**

El modificador **static** en Java se utiliza para declarar miembros (variables, métodos y bloques) que pertenecen a la clase en sí, en lugar de a instancias individuales de la clase. Los miembros static se cargan cuando se carga la clase y se comparten entre todas las instancias de la clase.

- *Ejemplo en código*

```
class Contador{
    public static int contador_estatico;
    public int contador_no_estatico;

    public void Suma(){
        contador_estatico++;
        contador_no_estatico++;
    }
}
public class Main {
    public static void main(String[] args) {
        Contador a = new Contador();
        Contador b = new Contador();

        a.Suma();
        b.Suma();

        //El contador_estatico es 2, ya que ambos objetos, a y b (que heredan de Contador)
        System.out.println(Contador.contador_estatico);
        //El contador_no_estatico se mantendra con 1
        System.out.println(a.contador_no_estatico);
    }
}
```

- **ArrayList<>**

- Es el equivalente a List<> en C#

- **Genéricos <T>**

- Nos permiten crear una interfaz, clase o método que se pueden usar con diferentes tipos de datos
- Ejemplo, digamos que quieres hacer una clase que reciba en su constructor un número. En un principio, no sabes como va a ser número, ya que 2 ; 20000; 0.1212 o Int.MaxValue son del tipo numérico, pero de diferente tipo. Entonces lo que harías sería hacer la clase genérica pero extendiendo de Number, y así no tendrás problemas con el tipo de dato que te pasen siempre que sea un número
- *ejemplo en código :*

```
//Clase generica que acepta cualquier parametro
class Generico<T>{
    T dato_generico;

    public Generico(T objeto){
        this.dato_generico = objeto;
    }
    //Devuelve el tipo de dato
    public void Mostrar(){
        System.out.println(dato_generico.getClass().getName());
    }
}
//Clase generica que acepta solo numeros
class NumerosGenericos<A extends Number> {
    A dato_generico_2;

    public NumerosGenericos(A objeto){
        this.dato_generico_2 = objeto;
    }
    //Devuelve el tipo de dato
```



```

        public void Mostrar(){
            System.out.println(dato_generico_2.getClass().getName());
        }
    }
    public class Main {
        public static void main(String[] args) {
            Generico<String> a = new Generico<>("Hola");
            a.Mostrar();

            NumerosGenericos b = new NumerosGenericos<>(3);
            b.Mostrar();
        }
    }
}

```

- **Enum**

- Son clases que puedes definir para elementos estáticos
- *Ejemplo en código:*

```

enum DiasDeLaSemana{
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES;
}

public class Main {
    public static void main(String[] args) {
        DiasDeLaSemana hoy = DiasDeLaSemana.JUEVES;
        System.out.println(hoy);
    }
}

```

- **Haschcode y Equals**

HashCode:

hashCode:

Imagina que tienes una caja llena de objetos. Para encontrar un objeto específico rápidamente, puedes asignarle un número único, como un código de barras. El código hash es como ese número único para los objetos en Java. Ayuda a identificar objetos rápidamente en estructuras de datos especiales llamadas "tablas hash".

Equals:

El método equals te permite comparar dos objetos para saber si son el mismo objeto o si son diferentes. Devuelve true si son el mismo objeto y false si son diferentes. Por ejemplo, si tienes dos objetos Persona con el mismo nombre y edad, el método equals devolverá true porque son la misma persona.

Juntos, hashCode y equals permiten a Java identificar y comparar objetos de manera eficiente.

Imagina que tienes una caja llena de objetos. Para encontrar un objeto específico rápidamente, puedes asignarle un número único, como un código de barras. El código hash es como ese número único para los objetos en Java. Ayuda a identificar objetos rápidamente en estructuras de datos especiales llamadas "tablas hash".

Juntos, hashCode y equals permiten a Java identificar y comparar objetos de manera eficiente.

- **Clases Abstractas**

- Son clases que no se pueden instanciar, su cuerpo se usa a través de una clase hija de esta clase
- *Ejemplo en código*

```

abstract class Example1{
    public void exampleMethod(){
        System.out.println("Metoddo de la clase Example 1");
    }
}

```

```

}
class Example2 extends Example1{
}

public class Main {
    public static void main(String[] args) {
        Example2 a = new Example2();
        a.exampleMethod();
    }
}

```

- **Multihilo y concurrencia: Manejo de múltiples tareas simultáneamente**

- <https://github.com/dnielpy/CodePractice/tree/main/08 - Threads>

- **Gestión de excepciones**

- Sirve cuando esperamos que pueda suceder un error. La forma de usarlo es: "Si ocurre este error" → "haz esto"
- *Ejemplo en código*

```

public class Main {
    public static void main(String[] args) {
        //Capturar excepciones
        int[] numeros = {1,2,3};

        //Ejemplo cuando no se rompe
        try {
            int mynumero = numeros[2];
            System.out.println("El numero es " + mynumero);
        }
        catch (ArrayIndexOutOfBoundsException ex){
            System.out.println("Se fue de rango");
        }

        //Ejemplo cuando se rompe
        try {
            int mynumero = numeros[3];
        }
        catch (ArrayIndexOutOfBoundsException ex){
            System.out.println("Se fue de rango");
        }

        //Ejemplo cuando se rompe de cualquier forma
        try {
            int mynumero = numeros[3];
        }
        catch (Exception ex){
            System.out.println("Se fue de rango");
        }
        //Tenga o no tenga errores, esto se ejecuta
        finally {
            System.out.println("Mensaje final");
        }
    }
}

```

Database - PostgreSQL

Fuente: https://www.youtube.com/watch?v=XQ_6G0iCyMQ&list=PLwvrYc43l1MxAEOL_KwGe8l42uJxMoKeS

Spring Boot

Fuentes:

-

Temas Secundarios

- **Patrones de diseño:** Patrones de diseño comunes en desarrollo de software
 - **Factory:** <https://github.com/dnielpy/CodePractice/tree/main/07 - Design Patterns/01 - Factory>
 - **Abstract Factory:** <https://github.com/dnielpy/CodePractice/tree/main/07 - Design Patterns/02 - Abstract Factory>
- **Principios SOLID:**
 - Fuente: *Katrib*
 - **S**imple Responsibility
 - **O**pen-closed
 - **L**iskov Solution
 - **I**nterface Segregation
 - **D**ependency Inversion
- Single Responsibility:
 - Plantea que cada clase debería recibir una única responsabilidad, es decir. Sus métodos deben estar enfocados hacia un mismo objetivo de resolver un problema bien definido.
 - Si el problema se divide en subproblemas, deberían crearse clases para resolver dichos subproblemas. De esta forma logramos una mejor escalabilidad y flexibilidad a los cambios
- Open - Closed:
 - Si por introducir cambios en una clase, surgen cambios inesperados en otra parte de la solución, se dice que esta es *fragil*. Un diseño robusto en sus clases, métodos y propiedades, plantea que por hacer cambios en una clase, no haya que hacer cambios en otras.
 - Las clases deben estar abiertas para su extensión, adaptación, ampliación y cerradas para su modificación.
 - Aquí es donde se ve el uso de las interfaces. Digamos que tenemos una clase que depende de un método Ordenar. Bajo este principio, podríamos ser capaces de cambiar el algoritmo de Ordenar sin afectar a la clase. Esto lo hacemos mediante interfaces. Definimos una interfaz que requiera un método Ordenar y luego extendemos nuestra clase de esta interfaz. De esta forma podríamos modificar como queramos el algoritmo Ordenar sin afectar a la clase
- **Liskov Solution o Principio de Sustitución**
 - Plantea que una instancia de una subclase debería poder sustituirse por una instancia de la superclase.
 - Para lograr esto, la superclase debería tener igual o menores exigencias
 - Un rectángulo es un polígono?
-
- Pruebas unitarias y de integración: Pruebas de código para garantizar la calidad
- Diseño orientado a objetos (OOP): Modelar soluciones utilizando principios OOP

