**SOBRE EL LIBRO DE QUE ES SPRING BOOT 2 (Apress -** Copyright © 2017)

Beginning Spring Boot 2

Applications and Microservices with the Spring Framework

The Spring framework is a very popular and widely used Java framework for building web and enterprise applications. Spring at its core is a dependency injection container that provides flexibility to configure beans in multiple ways, such as XML, Annotations, and JavaConfig.

Overview of the Spring Framework

If you are a Java developer, then there is a good chance that you have heard about the Spring framework and have used it in your projects. The Spring framework was created primarily as a dependency injection container, but it is much more than that.

Along with the Spring framework, there are many other Spring sub-projects that help build applications that address modern business needs:

• **Spring Data:** Simplifies data access from relational and NoSQL datastores.

• **Spring Batch:** Provides a powerful batch-processing framework.

• **Spring Security:** Robust security framework to secure applications.

Se puede configurar en xml, pero lo que es mas normal es configurarlo con annotation. **Annotation-Based Configuration:**

**@Service**

public class **UserService** {

private UserDao userDao;

**@Autowired**

public UserService(UserDao dao){

this.userDao = dao;

}

...

...

}

Y Java Based Configuration es cuando defines los Beans.Example of a **JavaConfig-Based Configuration:**

**@Configuration**

public class **AppConfig** {

**@Bean**

public **UserService** userService(UserDao dao){

return new UserService(dao);

}

**@Bean**

public **UserDao** userDao(DataSource dataSource){

return new JdbcUserDao(dataSource);

}

}

…

\*\*\*\*\* **PARA PONER UNA VISTA EN SPRING BOOT PROJECT** es: **src/main/webapp/WEB-INF/views/index.html**

Y la URL es: **//localhost:8080/springmvcjpa-demo**

This is a simple SpringMVC controller with one request handler method for URL /, which returns the view named index.html.

-------> Create a HTML view called **index.html**. By default, Spring Boot serves the static content from the **src/main/public/** and **src/main/static/** directories. So create **index.html** in **src/main/public/**, as shown:**src/main/public/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8"/>

<title>Home</title>

</head>

<body>

<h2>Hello World!!</h2>

</body>

</html>

<http://localhost:8080/>

**mvn spring-boot:run**

What Is Spring Boot?

Spring Boot is an opinionated framework that helps developers build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again. The key Spring Boot features include:

• Spring Boot starters

• Spring Boot autoconfiguration

• Elegant configuration management

• Spring Boot actuator

• Easy-to-use embedded servlet container support

Spring Boot Autoconfiguration

Spring Boot addresses the problem that Spring applications need complex configuration by eliminating the need to manually set up the boilerplate configuration. Spring Boot takes an opinionated view of the application and configures various components automatically, by registering beans based on various criteria. The criteria can be:

• Availability of a particular class in a classpath

• Presence or absence of a Spring bean

• Presence of a system property

• Absence of a configuration file

For example, if you have the spring-webmvc dependency in your classpath, Spring Boot assumes you are trying to build a SpringMVC-based web application and automatically tries to register DispatcherServlet if it is not already registered. If you have any embedded database drivers in the classpath, such as H2 or HSQL, and if you haven’t configured a DataSource bean explicitly, then Spring Boot will automatically register a DataSource bean using in-memory database settings.

Elegant Configuration Management

Spring supports externalizing configurable properties using the @PropertySource configuration.

Easy-to-Use Embedded Servlet Container Support

Traditionally, while building web applications, you need to create WAR type modules and then deploy them on external servers like Tomcat, WildFly, etc. But by using Spring Boot, you can create a JAR type module and embed the servlet container in the application very easily so that the application will be a self-contained deployment unit. Also, during development, you can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command-line using a build tool like Maven or Gradle.

public **@interface** **SpringBootApplication** {

....

....

}

The @**SpringBootConfiguration** is another composed annotation with the **@Configuration** annotation. Here are the meanings of these annotations:

• @**Configuration** indicates that this class is a Spring configuration class.

• @**ComponentScan** enables component scanning for Spring beans in the package in which the current class is defined.

• @**EnableAutoConfiguration** triggers Spring Boot’s autoconfiguration mechanisms.

Tengo 1 main package: package **com.mycompany.myproject**;

It is highly recommended that you put the main entry point class in the root package, say in **com.mycompany.myproject**, so that the **@EnableAutoConfiguration** and **@ComponentScan** annotations will scan for Spring beans, JPA entities, etc., in the root and all of its sub-packages automatically. If you have an entry point class in a nested package, you might need to specify the basePackages to scan for Spring components explicitly

Main Class **com.mycompany.myproject.config.Application.java** in a Non-Root Package **package com.mycompany.myproject.config**;

@Configuration

@EnableAutoConfiguration

@ComponentScan(**basePackages = "com.mycompany.myproject"**)

@EntityScan(**basePackageClasses=Person.class**)

public class **Application** {

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

**SpringApplication**.**run**(**Application.class**, args);

}

}

Here, the Application.java main class is in the **com.mycompany.myproject.config** package, which is not the root package. So, you need to specify **@ComponentScan(basePackages = "com.mycompany.myproject")** so that Spring Boot will scan **com.mycompany.myproject** and all of its sub-packages for Spring components. Also, we specified **@EntityScan**(**basePackageClasses=Person.class**) so that Spring Boot will scan for JPA entities under the package where **Person.class** exists.

Fat JAR Using the Spring Boot Maven Plugin

You can run your application directly from the IDE or use Maven spring-boot:run during development, but ultimately you need to create a deployment unit that can be run in the production environment without any IDE support. You can use spring-boot-maven-plugin to create a single deployment unit (a fat JAR) by executing the following Maven goals.

**mvn clean package**

Now there are two interesting files in the target directory—springboot-basic-1.0-SNAPSHOT.jar and springboot-basic-1.0-SNAPSHOT.jar.original. The springboot-basic-1.0-SNAPSHOT.jar.original file will contain only the compiled classes and classpath resources.

But if you look at springboot-basic-1.0-SNAPSHOT.jar, you find the following:

• Compiled classes of your own source code in src/main/java and static resources from src/main/resources will be in the BOOT-INF/classes directory

• All the dependent JARs in the BOOT-INF/lib directory

• Classes in the org.springframework.boot.loader package that do the Spring Boot magic of running the Spring Boot application

You can run the application using the following command:

**java -jar springboot-basic-1.0-SNAPSHOT.jar**

**GRADLE**: Now you can run the application by using the gradle bootRun command. You can also use the gradle build command, which generates the fat JAR in the build/libs directory.

**Maven or Gradle?**

In the Java world, Maven and Gradle are the two most popular build tools. Maven was released in 2004 and is used widely by many developers. Gradle was released in 2012 and it’s more powerful and easy to customize.

**THE PROFILE FEATURE**

**@Configuration**

public class **AppConfig** {

@Bean

**@Profile("DEV")**

public **DataSource** devDataSource() {

...

}

@Bean

**@Profile("PROD")**

public **DataSource** prodDataSource() {

...

}

}

With this configuration, you can specify the active profile using the **-Dspring.profiles.active=DEV** system property.

Logging

Logging is a very important part of any application and it helps with debugging issues. Spring Boot, by default, includes spring-boot-starter-logging as a transitive dependency for the spring-boot-starter module. By default, Spring Boot includes SLF4J along with Logback implementations. Spring Boot has a

LoggingSystem abstraction that automatically configures logging based on the logging configuration files available in the classpath. If Logback is available, Spring Boot will choose it as the logging handler. You can easily configure logging levels within the application.properties file without having to create logging provider specific configuration files such as logback.xml or log4j.properties.

logging.level.org.springframework.web=INFO

logging.level.org.hibernate=ERROR

logging.level.com.apress=DEBUG

If you want to log the data into a file in addition to the console, specify the filename as follows:

logging.path=/var/logs/app.log

or

logging.file=myapp.log

If you want to have more control over the logging configuration, create the logging provider specific configuration files in their default locations, which Spring Boot will automatically use. For example, if you place the logback.xml file in the root classpath, Spring Boot will automatically use it

to configure the logging system.

If you want to use other logging libraries, such as Log4J or Log4j2, instead of Logback, you can exclude spring-boot-starter-logging and include the respective logging starter, as follows:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-log4j</artifactId>

</dependency>

Now you can add the log4j.properties file to the root classpath, which Spring Boot will automatically use for logging.

Externalizing Configuration Properties

Typically you will want to externalize configuration parameters into separate properties or XML files instead of burying them inside code so that you can easily change them based on the environment of the application. Spring provides the **@PropertySource** annotation to specify the list of configuration files.

Spring Boot takes it one step further by automatically registering a **PropertyPlaceHolderConfigurer** bean using the **application.properties** file in the root classpath by default. You can also create profile specific configuration files using the filename as application-{**profile**}.properties.

For example, you can have application.properties, which contains the default properties values, **application-dev.properties**, which contains the dev profile configuration, and application-**prod**.properties, which contains the production profile configuration values. If you want to configure properties that are common for all the profiles, you can configure them in **application-default.properties**.

Type-Safe Configuration Properties

Spring provides the **@Value** annotation to bind any property value to a bean property. Suppose you had the following **application.properties** file:

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/test

jdbc.username=root

jdbc.password=secret

You can bind these property values into bean properties using **@Value** as follows:

**@Configuration**

public class **AppConfig** {

**@Value("${jdbc.driver}")**

private String driver;

**@Value("${jdbc.url}")**

private String url;

**@Value("${jdbc.username}")**

private String username;

**@Value("${jdbc.password}")**

private String password;

…

}

But binding each property using **@Value** is a tedious process. So, Spring Boot introduced a mechanism to bind a set of properties to a bean's properties automatically in a type-safe manner. Suppose you have the previous JDBC parameters and a **DataSourceConfig** class as follows:

public class **DataSourceConfig** {

private String driver;

private String url;

private String username;

private String password;

//setters and getters

}

Now you can simply annotate **DataSourceConfig** with **@ConfigurationProperties(prefix="jdbc")** to automatically bind the properties that start with jdbc.\*.

**@Component**

**@ConfigurationProperties(prefix="jdbc")**

public class DataSourceConfig {

...

...

**Validating Properties with the Bean Validation API**

You can use Bean Validation API annotations such as **@NotNull,@Min, @Max**, etc., to validate the property’s values.

**@Component**

**@ConfigurationProperties(prefix="support")**

public class **Support** {

**@NotNull**

private String applicationName;

**@NotNull**

**@Email**

private String email;

}

Customizing the Banner

The banner that is printed on start up can be changed by adding a banner.txt file to your classpath or by setting the **spring.banner.location property** to the location of such a file. If the file has an encoding other than UTF-8, you can set **spring.banner.charset**. In addition to a text file, you can also add a banner.gif, banner.jpg, or banner.png image file to your classpath or set the spring.banner.image.location property. Images are converted into an ASCII art representation and printed above any text banner.

Inside your banner.txt file, you can use any of the following placeholders:

| **Table 4. Banner variables** | |
| --- | --- |
| **Variable** | **Description** |
| ${application.version} | The version number of your application, as declared in MANIFEST.MF. For example, Implementation-Version: 1.0 is printed as 1.0. |
| ${application.formatted-version} | The version number of your application, as declared in MANIFEST.MF and formatted for display (surrounded with brackets and prefixed with v). For example (v1.0). |
| ${spring-boot.version} | The Spring Boot version that you are using. For example 2.2.5.RELEASE. |
| ${spring-boot.formatted-version} | The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with v). For example (v2.2.5.RELEASE). |
| ${Ansi.NAME} (or ${AnsiColor.NAME}, ${AnsiBackground.NAME}, ${AnsiStyle.NAME}) | Where NAME is the name of an ANSI escape code. See [AnsiPropertySource](https://github.com/spring-projects/spring-boot/tree/v2.2.5.RELEASE/spring-boot-project/spring-boot/src/main/java/org/springframework/boot/ansi/AnsiPropertySource.java) for details. |
| ${application.title} | The title of your application, as declared in MANIFEST.MF. For example Implementation-Title: MyApp is printed as MyApp. |

|  |  |
| --- | --- |
|  | The SpringApplication.setBanner(…​) method can be used if you want to generate a banner programmatically. Use the org.springframework.boot.Banner interface and implement your own printBanner() method. |

You can also use the spring.main.banner-mode property to determine if the banner has to be printed on System.out (console), sent to the configured logger (log), or not produced at all (off).

The printed banner is registered as a singleton bean under the following name: springBootBanner.

#### Customizing SpringApplication

If the SpringApplication defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

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

SpringApplication app = **new** SpringApplication(MySpringConfiguration.class);

app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

The constructor arguments passed to SpringApplication are configuration sources for Spring beans. In most cases, these are references to **@Configuration** classes, but they could also be references to XML configuration or to packages that should be scanned.

It is also possible to configure the SpringApplication by using an application.properties file. See [Externalized Configuration](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) for details.

For a complete list of the configuration options, see the [SpringApplication Javadoc](https://docs.spring.io/spring-boot/docs/2.2.5.RELEASE/api/org/springframework/boot/SpringApplication.html).

**SPRING BOOT Y LAS BASES DE DATOS**

To add the H2 database driver to the pom.xml file, you use the following:

<dependency>

<groupId>**com.h2database**</groupId>

<artifactId>**h2**</artifactId>

</dependency>

Create schema.sql in src/main/resources, as follows:

**CREATE TABLE users (**

**id int(11) NOT NULL AUTO\_INCREMENT,**

**name varchar(100) NOT NULL,**

**email varchar(100) DEFAULT NULL,**

**PRIMARY KEY (id)**

**);**

Create data.sql in src/main/resources, as follows:

insert into users(id, name, email) values(1,'John','john@gmail.com');

insert into users(id, name, email) values(2,'Rod','rod@gmail.com');

insert into users(id, name, email) values(3,'Mike','mike@gmail.com');

Using Spring Data JPA with Spring Boot

Now that you’ve had a glimpse of what Spring Data JPA is and what kind of features it provides, this section shows you how to put it into action.

1. Create a Spring Boot Maven project and add the following dependencies.

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-data-jpa**</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>**h2**</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-test**</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

2. Create a JPA entity called User and a JPA repository interface called **UserRepository**.

3. Create a user JPA entity. JPA Entity User.java:

**@Entity**

**@Table(name="USERS")**

public class **User** {

**@Id**

**@GeneratedValue(strategy=GenerationType.AUTO)**

private Integer id;

**@Column(nullable=false)**

private String name;

**@Column(nullable=false, unique=true)**

private String email;

private boolean disabled;

//setters and getters

}

4. Create the UserRepository interface by extending the JpaRepository interface,

JPA Repository Interface **UserRepository.java**

public interface **UserRepository** extends **JpaRepository**<User, Integer> { }

5. You can now populate some sample data using the SQL script src/main/resources/data.sql:

**insert into users(id, name, email,disabled) values (1,'John','john@gmail.com', false);**

Since you configured the in-memory database (**H2**) driver, Spring Boot automatically registers a DataSource. Since you added the spring-boot-starter-data-jpa dependency, Spring Boot autoconfiguration takes care of creating the JPA related beans like LocalContainerEntityManagerFactoryBean, TransactionManager, etc., automatically with sensible defaults.

6. Create a Spring Boot entry point class called **SpringbootJPADemoApplication.java. SpringbootJPADemoApplication.java**

**@SpringBootApplication**

public class **SpringbootJPADemoApplication** {

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

**SpringApplication**.run(SpringbootJPADemoApplication.class, args);

}

}

7. Create a JUnit test class for testing the **UserRepository** methods. **SpringbootJPADemoApplicationTests.java**

**@RunWith(SpringRunner.class)**

**@SpringBootTest**

public class **SpringbootJPADemoApplicationTests** {

**@Autowired**

private UserRepository userRepository;

**@Test**

public void findAllUsers() {

List<User> users = userRepository.findAll();

assertNotNull(users);

assertTrue(!users.isEmpty());

}

Add Dynamic Query Methods

Now you’ll add some finder methods to see how dynamic query generation based on method names works. To get a user by name, use this:

User findByName(String name)

To search for users by name, use this: List<User> findByNameLike(String name)

The preceding method generates a where clause like where u.name like ?1.

Suppose you want to do a wildcard search, such as where u.name like %?1%. You can use @Query as follows:

@Query("select u from User u where u.name like %?1%")

List<User> searchByName(String name)

**CHAPTER 10: Web Applications with Spring Boot**

Spring MVC is the most popular Java web framework based on the **Model-View-Controller (MVC)** design pattern. Since the Spring 3.0 version, Spring MVC has provided annotation based request mapping capabilities using **@Controller** and **@RequestMapping**.

Introducing SpringMVC

Spring MVC is a powerful web framework built on MVC and front controller design patterns. Spring MVC provides DispatcherServlet, which acts as a front controller by receiving all the requests and delegates the processing to request handlers (controllers). Once the processing is done, ViewResolver will render a view based on the view name.

Spring MVC provides annotation-based mapping support to map request URL patterns to handler classes using @Controller and @RequestMapping annotations

SpringMVC Annotation-Based Controller

**@Controller**

public class HomeController {

**@RequestMapping(value="/home", method=RequestMethod.GET)**

public String home(Model model) {

model.addAttribute("message", "Hello Spring MVC!!");

return "home";

}

}

The **@Controller** annotation on the HomeController class marks it as a request handler Spring component and the home() method will handle the GET requests to the /home URL. The ViewResolver will resolve the logical view name "home" to a view template, say **/WEB-INF/views/home.html**, and then render the view.

Developing Web Application Using Spring Boot

Spring Boot provides the Web starter spring-boot-starter-web for developing web applications using Spring MVC. Spring Boot autoconfiguration registers the SpringMVC beans like DispatcherServlet, ViewResolvers, ExceptionHandlers, etc. You can develop a Spring Boot web application as a JAR type

module using an embedded servlet container or as a WAR type module, which can be deployed on any external servlet container.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-web**</artifactId>

</dependency>

The spring-boot-starter-web starter by default configures DispatcherServlet to the URL pattern "/" and adds Tomcat as the embedded servlet container, which runs on port 8080. Spring Boot by default serves the static resources (HTML, CSS, JS, images, etc.) from the following CLASSPATH locations:

• /static

• /public

• /resources

• /META-INF/resources

**SPRING BOOT Y BOOTSTRAP**

1. Create a Spring Boot Maven project with the Web starter and add the Bootstrap WebJars dependency.

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-web**</artifactId>

</dependency>

<dependency>

<groupId>org.webjars.bower</groupId>

<artifactId>**bootstrap**</artifactId>

<version>3.3.7</version>

</dependency>

</dependencies>

2. Create the styles.css stylesheet in the src/main/resources/static/css folder.

body {

background-color: #A7A5A4;

padding-top: 50px;

}

3. Copy an image, such as spring-boot.png, into the src/main/resources/static/images folder.

4. Create the index.html file in the src/main/resources/public folder and add bootstrap.css to the index.html file. Use the Bootstrap navigation bar component.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Home</title>

<link rel="stylesheet" href="webjars/bootstrap/3.3.7 /css/bootstrap.css" />

<link rel="stylesheet" href="css/styles.css" />

</head>

<body>

<nav class="navbar navbar-inverse navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle collapsed"

data-toggle="collapse" data-target="#navbar"

aria-expanded="false" aria-controls="navbar">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" href="#">Project name</a>

</div>

<div id="navbar" class="collapse navbar-collapse">

<ul class="nav navbar-nav">

<li class="active">

<a href="#">Home</a>

</li>

<li>

<a href="#about">About</a>

</li>

<li>

<a href="#contact">Contact</a>

</li>

</ul>

</div>

</div>

</nav>

<div class="container">

<h2>Hello World!!</h2>

<img alt="SpringBoot" src="images/spring-boot.png" />

</div>

</body>

</html>

5. Create an application entry point class.

@SpringBootApplication

public class SpringbootWebDemoApplication {

public static void main(String[] args) {

SpringApplication.run(SpringbootWebDemoApplication.class, args);

}

}

Now run the SpringbootWebDemoApplication and navigate to http://localhost:8080/

**COMO ALTERAR LA URL**

By default, the Spring Boot Web starter uses Tomcat as the embedded servlet container and runs on port 8080. However, you can customize the server properties using server.\* in application.properties.

server.port=9090

server.servlet.context-path=/demo

server.servlet.path=/app

With these customizations, DispatcherServlet is configured to handle the URL pattern /app, the root contextPath will be /demo, and Tomcat now runs on port 9090. So, you would access the index.html file at http://localhost:9090/demo/app/.

Using the Tomcat, Jetty, and Undertow Embedded Servlet Containers

As mentioned, the Spring Boot Web starter includes Tomcat as the embedded servlet container by default.

Instead of Tomcat, though, you can use other servlet containers like **Jetty** or **Undertow**.

To use Jetty as the embedded container, you simply need to exclude spring-boot-starter-tomcat and add spring-boot-starter-jetty.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

**<exclusions>**

**<exclusion>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-tomcat</artifactId>**

**</exclusion>**

**</exclusions>**

</dependency>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-jetty</artifactId>**

**</dependency>**

Undertow (http://undertow.io/) is a web server written in Java. It provides blocking and non-blocking APIs based on NIO. Spring Boot provides autoconfiguration support for the Undertow server as well. Similar to what you saw with Jetty, you can configure Spring Boot to use the Undertow embedded server instead of Tomcat as follows:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

**<exclusions>**

**<exclusion>**

**<artifactId>spring-boot-starter-tomcat</artifactId>**

**<groupId>org.springframework.boot</groupId>**

**</exclusion>**

**</exclusions>**

</dependency>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-undertow</artifactId>**

**</dependency>**

You can customize various properties of the Tomcat, Jetty, and Undertow servlet containers using the **server.tomcat**.\* , server.jetty.\*, and server.undertow.\* properties, respectively.

server.tomcat.accesslog.directory=logs # Directory in which log files are created.

server.tomcat.accesslog.enabled=false # Enable access log.

server.tomcat.accesslog.file-date-format=.yyyy-MM-dd # Date format to place in log file name.

server.tomcat.basedir= # Tomcat base directory. If not specified a temporary directory will be used.

server.tomcat.max-connections= # Maximum number of connections that the server will accept and process at any given time.

server.tomcat.max-http-header-size=0 # Maximum size in bytes of the HTTP message header.

server.tomcat.max-http-post-size=0 # Maximum size in bytes of the HTTP post content.

server.tomcat.max-threads=0 # Maximum amount of worker threads.

server.tomcat.min-spare-threads=0 # Minimum amount of worker threads.

server.tomcat.port-header=X-Forwarded-Port # Name of the HTTP header used to override the original port value.

Spring Boot Web Application as a Deployable WAR

The Spring Boot web application can be developed using WAR type packaging also. The first thing you do if you want to build a deployable WAR file is change the packaging type. If you are using Maven, then in pom.xml, change the packaging type to war.

**<packaging>war</packaging>**

If you are using Gradle, you need to apply the WAR plugin.

**apply plugin: 'war'**

When you add the **spring-boot-starter-web** dependency. It will transitively add the **spring-bootstarter-tomcat** dependency as well. So you need to add **spring-boot-starter-tomcat** as the provided scope so that it won’t get packaged inside the **WAR** file.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-tomcat**</artifactId>

<scope>provided</scope>

</dependency>

If you are using Gradle, add spring-boot-starter-tomcat with the providedRuntime scope as follows:

dependencies {

...

providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

...

}

Finally, you need to provide a SpringBootServletInitializer sub-class and override its configure() method. You can simply make your application’s entry point class extend SpringBootServletInitializer Implementing SpringBootServletInitializer

**@SpringBootApplication**

public class **SpringbootWebDemoApplication** extends **SpringBootServletInitializer** {

**@Override**

protected SpringApplicationBuilder **configure**(SpringApplicationBuilder application) {

return application.sources(SpringbootWebDemoApplication.class);

}

public static void **main**(String[] args) throws **Exception** {

SpringApplication.run(SpringbootWebDemoApplication.class, args);

}

}

Now running the Maven/Gradle build tool will produce a WAR file that can be deployed on an external server.

**CHAPTER 11**

**Building REST APIs Using Spring Boot**

REST (REpresentational State Transfer) is an architectural style for building distributed systems that provide interoperability between heterogeneous systems.

SpringMVC provides first-class support for building RESTful web services. As Spring’s REST support is built on top of SpringMVC, you can leverage the knowledge of SpringMVC for building REST APIs. Spring Data REST is a spring portfolio project that can be used to expose Spring Data repositories

as REST endpoints.

\*\*\* The most commonly used data exchange formats (ContentTypes) are JSON and XML. The typical practice of determining the input request content and output response types in web based systems are based on the **ContentType** and **Accept** header values.

REST API Using SpringMVC

SpringMVC provides support for building RESTful web services and Spring Boot makes it much easier with its autoconfiguration mechanism: SpringMVC-based REST endpoint.

@Controller

public class PostController {

@Autowired

PostRepository postRepository;

**@ResponseBody**

@GetMapping("/posts")

public List<Post> listPosts() {

return postRepository.findAll();

}

}

If all of your handler methods are REST endpoint handler methods, you can have a @**ResponseBody** at the class level instead of adding it to each method. Even better, you can use @**RestController**, which is a composed annotation of @**Controller** and @**ResponseBody**.

**PROJECT: REST API for a simple blog application using Spring Data JPA, SpringMVC, and, Spring Boot**

**1. Create a Spring Boot project and configure the Web and JPA starters. Create a Spring Boot Maven project and add the following starters:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

**2. Create these resources as JPA entities**

**@Entity**

**@Table(name = "USERS")**

public class **User** {

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

private Integer id;

**@Column(name = "name", nullable = false, length = 150)**

private String name;

…

//setters & getters

}

**@Entity**

**@Table(name = "POSTS")**

public class **Post** {

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

private Integer id;

**@Column(name = "title", nullable = false, length = 150)**

private String title;

**@Lob // que es Lob???**

**@Column(name = "content", nullable = false, columnDefinition="TEXT")**

private String content;

**@Temporal(TemporalType.TIMESTAMP)**

**@Column(name="created\_on")**

private Date createdOn = new Date();

**@OneToMany // UN POST TIENE VARIOS COMENTARIOS**

**@JoinColumn(name="post\_id")**

private List<Comment> comments;

//setters & getters

}

**@Entity**

**@Table(name = "COMMENTS")**

public class **Comment** {

**@Id**

**@GeneratedValue(strategy = GenerationType.IDENTITY)**

private Integer id;

**@Column(name = "name", nullable = false, length = 150)**

private String name;

**@Column(name = "email", nullable = false, length = 150)**

private String email;

**@Lob**

**@Column(name = "content", nullable = false, columnDefinition="TEXT")**

private String content;

**3. Now you create the Spring Data JPA repositories for the JPA entities you just created**

public class **UserRepository** extends **JpaRepository**<**User**, Integer> { }

public class **PostRepository** extends **JpaRepository**<**Post**, Integer> { }

public class **CommentRepository** extends **JpaRepository**<**Comment**, Integer> { }

**4. Now you create a SpringMVC controller to implement all the Post resource related REST endpoints:**

**@RestController**

**@RequestMapping(value="/posts")**

public class **PostController** {

**@Autowired**

PostRepository postRepository;

**@Autowired**

CommentRepository commentRepository;

...

...

}

**\*\*\*** you need to create a custom exception class, called **ResourceNotFoundException**

**@ResponseStatus(HttpStatus.NOT\_FOUND)**

public class **ResourceNotFoundException** extends **RuntimeException** {

public **ResourceNotFoundException**() {

this("Resource not found!");

}

public **ResourceNotFoundException**(String message) {

this(message, null);

}

public **ResourceNotFoundException**(String message, Throwable cause) {

super(message, cause);

}

}

Note that the example annotates the ResourceNotFoundException class with @ResponseStatus(HttpStatus.NOT\_FOUND) so that when the request handler

method throws ResourceNotFoundException.

**\*\*\*** implementing the endpoint for creating a new post > If the POST creation is successful, you return the appropriate HTTP status code (201 CREATED) along with newly created post as the Response body.

**@ResponseStatus(HttpStatus.CREATED)**

**@PostMapping("")**

public Post **createPost**(@**RequestBody** Post post) {

return postRepository.save(post);

}

**\*\*\*\*\*** you will implement the endpoint for fetching a post for the given ID for which the endpoint URL will be GET [http://localhost:8080/posts/{id}](http://localhost:8080/posts/%7bid%7d).

**@GetMapping(value="/{id}")**

public Post **getPost**(@**PathVariable**("id") Integer id) {

return postRepository.**findById**(id)

. **orElseThrow**(() -> new **ResourceNotFoundException**("No post found with id="+id));

}

This example is getting the Post object from the database for the given ID and throwing ResourceNotFoundException if the post is not found; otherwise, it

returns the POST object.

Next, you need to implement the endpoint for updating a post for the given ID for which the endpoint URL will be PUT http://localhost:8080/posts/{id}.

**@PutMapping("/{id}")**

public Post **updatePost**(@**PathVariable**("id") Integer id, @**RequestBody** Post post) {

postRepository.**findById**(id).**orElseThrow**(() -> new **ResourceNotFoundException**("No post found with id="+id));

return **postRepository**.save(post);

}

The example gets the Post object from the database for the given ID and throws the ResourceNotFoundException if the post not found, otherwise updating the post. Similarly, you can implement the endpoint by deleting a post using the HTTP DELETE method at the URI http://localhost:8080/posts/{id}, as follows:

**@DeleteMapping("/{id}")**

public void **deletePost**(@**PathVariable**("id") Integer id) {

Post post = postRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("No post found with id="+id));

postRepository.deleteById(post.getId());

}

Note that you are not returning any content to the client, so on successful deletion of the post, the HTTP 200 OK status code will be sent.

**\*\*\*\*\*** You can also add validation for the REST endpoint handler methods, similar to SpringMVC controllers for traditional web applications.

**@ResponseStatus(HttpStatus.CREATED)**

**@PostMapping(value="")**

public Post **createPost**(@**RequestBody** @Valid Post post, BindingResult result) {

if(result.**hasErrors**()){

//handle errors

//throw Exception with Invalid data details

}

return **postRepository**.save(post);

}

This example adds the @**Valid** annotation to the method parameter Post so that the post object data will be validated against the Java Bean Validation Constraints defined on the POST properties

**\*\*\*\*\*** In addition to returning a arbitrary object from request handler methods, you can also return ResponseEntity/HttpEntity, which provides an easy way to set response headers and the status code.

Using ResponseEntity for Fine-Grained Control Over Responses

**@PostMapping("")**

public **ResponseEntity**<Post> **createPost**(@**RequestBody** @**Valid** Post post, **BindingResult** result) {

if(result.**hasErrors**()){

return new ResponseEntity<>(post, HttpStatus.BAD\_REQUEST);

}

Post savedPost = postRepository.save(post);

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.set("MyResponseHeader1", "MyValue1");

responseHeaders.set("MyResponseHeader2", "MyValue2");

return new ResponseEntity<>(savedPost, responseHeaders, HttpStatus.CREATED);

}

You can also use Spring’s RestTemplate as a client to invoke RESTful services. Now you’ll test the REST endpoints you have implemented using RestTemplate. First, you need to populate some sample data using the SQL script

**src/test/resources/data.sql**

delete from users;

**INSERT INTO users (id, email, password, name) VALUES (1, 'admin@gmail.com', 'admin', 'Admin');**

insert into posts(id, title, content, created\_on, updated\_on) values (1, 'Introducing SpringBoot', 'SpringBoot is awesome', '2017-05-10', null);

insert into comments(id, post\_id, name, email, content, created\_on, updated\_on) values (1, 1, 'John','john@gmail.com', 'This is cool', '2017-05-10', null);

**\*\*\*\*\*** Now you’ll write a Spring Boot test that starts an embedded servlet container on a defined port (server.port value) and uses RestTemplate to invoke the REST API endpoints.

Testing REST Endpoints Using RestTemplate

**@RunWith(SpringRunner.class)**

**@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED\_PORT)**

public class **SpringbootMvcRestDemoApplicationTest** {

private static final String ROOT\_URL = "http://localhost:8080";

RestTemplate restTemplate = new RestTemplate();

**@Test**

public void **testGetAllPosts**() {

ResponseEntity<Post[]> responseEntity = restTemplate.getForEntity(ROOT\_URL+"/posts", Post[].class);

List<Post> posts = Arrays.asList(responseEntity.getBody());

assertNotNull(posts);

}

**@Test**

public void **testGetPostById**() {

Post post = restTemplate.getForObject(ROOT\_URL+"/posts/1", Post.class);

assertNotNull(post);

}

…

}

CORS (Cross-Origin Resource Sharing) Support

For security reasons, browsers don’t allow you to make AJAX requests to resources residing outside of the current origin. CORS specification (https://www.w3.org/TR/cors/) provides a way to specify which crossorigin requests are permitted. SpringMVC provides support for enabling CORS for REST API endpoints so that the API consumers, such as web clients and mobile devices, can make calls to REST APIs.

Class- and Method-Level CORS Configuration

You can enable CORS at the controller level or at the method level using the @CrossOrigin annotation. Now you’ll see how you can enable CORS support on a specific request handling method.

**@RestController**

public class **UserController** {

**@CrossOrigin**

**@GetMapping("/users/{id}")**

public User **getUser**(@**PathVariable** Long id) {

// ...

}

**@DeleteMapping("/users/{id}")**

public void **deleteUser**(@**PathVariable** Long id) {

// ...

}

}

**\*\*\*** You can customize these properties by providing options on the @CrossOrigin annotation.

@**CrossOrigin**(origins= {"http://domain1.com", "http://domain2.com"},

allowedHeaders="X-AUTH-TOKEN",

allowCredentials="false",

maxAge=15\*60,

methods={RequestMethod.GET, RequestMethod.POST } )

@**GetMapping**("/users/{id}")

public User **getUser**(@**PathVariable** Long id) {

// ...

}

**Using @JsonIgnore**

You can break the infinite recursion by adding the @JsonIgnore annotation on the back reference from the child object.

**@Entity**

**@Table(name = "COMMENTS")**

public class **Comment** {

...

**@JsonIgnore**

**@ManyToOne(optional=false)**

**@JoinColumn(name="post\_id")**

private Post post;

...

}

You can add @**JsonIgnore** on all the properties that you want to exclude from marshaling or you can use @**JsonIgnoreProperties** at the class level to list all the property names to ignore.

**WEBFLUX**

Reactive WebClient

Spring provides RestTemplate to invoke RESTful service endpoints, which support message converters so that HTTP requests can be made using Java objects instead of preparing an input request body with JSON or XML manually.

Spring WebFlux provides WebClient as a reactive alternative to RestTemplate that supports non-blocking. Instead of using InputStream and OutputStream for request processing, WebClient uses Flux<DataBuffer> as the request and the response body.

**\*\*\*\*\*** Invoking Reactive REST Endpoints Using WebClient

**WebClient** webClient = **WebClient.create**("http://localhost:"+port);

List<User> users = **webClient.get**().**uri**("/api/users").**accept**(MediaType.APPLICATION\_JSON).**exchange**()

.**flatMap**(response -> response.**bodyToFlux**(User.class).**collectList**()).block();

**COMO TESTEAR CON TESTRESTEMPLATE**

Spring REST Controller:

**@RestController**

public class **PingController** {

**@GetMapping("/ping")**

public String ping() {

return "OK";

}

}

Now if you run the application, you can invoke the REST endpoint [**http://localhost:8080/ping**](http://localhost:8080/ping), which gives the response "OK". Now you can write a test for the /ping endpoint.

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;

import org.springframework.boot.test.web.client.TestRestTemplate;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.test.context.junit4.SpringRunner;

**@RunWith(SpringRunner.class)**

**@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)**

public class **PingControllerTests** {

**@Autowired**

TestRestTemplate restTemplate;

**@Test**

public void testPing() {

ResponseEntity<String> respEntity = restTemplate.getForEntity("/ping", String.class);

assertThat(respEntity.getStatusCode()).isEqualTo(HttpStatus.OK);

assertThat(respEntity.getBody()).isEqualTo("OK");

}

}

As you need to test the REST endpoint, you start the embedded servlet container by specifying the webEnvironment attribute of @SpringBootTest.

The default webEnvironment value is WebEnvironment.MOCK, which doesn’t start an embedded servlet container.

You can use various webEnvironment values based on how you want to run the tests.

• MOCK (default)—Loads a WebApplicationContext and provides a mock servlet environment. It will not start an embedded servlet container. If servlet APIs are not on your classpath, this mode will fall back to creating a regular non-web ApplicationContext.

• RANDOM\_PORT— Loads a ServletWebServerApplicationContext and starts an embedded servlet container listening on a random available port.

• DEFINED\_PORT—Loads a ServletWebServerApplicationContext and starts an embedded servlet container listening on a defined port (server.port).

• NONE—Loads an ApplicationContext using SpringApplication but does not provide a servlet environment.

The TestRestTemplate bean will be registered automatically only when @SpringBootTest is started with an embedded servlet container.

**Testing SpringMVC Controllers Using @WebMvcTest**

Spring Boot provides the **@WebMvcTest** annotation, which will autoconfigure SpringMVC infrastructure components and load only **@Controller**, **@ControllerAdvice**, **@JsonComponent**, Filter, WebMvcConfigurer, and HandlerMethodArgumentResolver components. Other Spring beans (annotated with **@Component**, **@Service**, **@Repository**, etc.) will not be scanned when using this annotation.

**Deploying Spring Boot Applications**

Spring Boot supports embedded servlet containers, which makes deploying applications much easier, because you don’t need an external application server setup. You can simply package your Spring Boot application as a JAR module and run it using the java **-jar** command. However, you need to consider a few

things while running applications in a production environment. You can use profiles to externalize configuration properties per environment and run your application, activating desired profiles.

Once the application is packaged as a JAR, you can simply run the application as follows:

**java -jar app.jar**

Pero ojo con los profiles:

src/main/resources/**application-prod.properties**

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://prodmysqlsrv:3306/myapp

spring.datasource.username=appuser

spring.datasource.password=S3\*(Hi)@32vi

configuration is pointing to remote MySQL server properties for the prod profile. Now you can run the application in production by activating the prod profile, as follows:

java -jar **-Dspring.profiles.active=prod** app.jar

o si le paso mas de uno:

java -jar **-Dspring.profiles.active=prod,cloud** app.jar

Spring Boot allows you to override the configuration parameters in various ways. You can override the properties using system properties as follows:

java -jar -Dserver.port=8585 app.jar

*Spring in Action* **--- CRAIG WALLS (2019)**

***Working with configuration properties***

***5.1 Fine-tuning autoconfiguration***

Before we dive in too deeply with configuration properties, it’s important to establish that there are two different (but related) kinds of configurations in Spring:

 *Bean wiring*—Configuration that declares application components to be created as beans in the Spring application context and how they should be injected into each other.

 *Property injection*—Configuration that sets values on beans in the Spring application context.

***5.1.1 Understanding Spring’s environment abstraction***

The Spring environment abstraction is a one-stop shop for any configurable property. It abstracts the origins of properties so that beans needing those properties can consume them from Spring itself. The Spring environment pulls from several property sources, including:

 JVM system properties

 Operating system environment variables

 Command-line arguments

 Application property configuration files

Suppose that you would like the application’s underlying servlet container to listen for requests on some port other than the default port of 8080. To do that, specify a different port by setting the server.port property in src/main/resources/application.properties like this:

server.port=9090

If you’d prefer to configure that property externally, you could also specify the port when starting the application using a command-line argument:

$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090

If you want the application to always start on a specific port, you could set it one time as an operating system environment variable:

$ export SERVER\_PORT=9090

Spring is able to sort it out and interpret

SERVER\_PORT as server.port with no problems.

***5.1.4 Configuring logging***

Most applications provide some form of logging. And even if your application doesn’t log anything directly, the libraries that your application uses will certainly log their activity. **By default, Spring Boot configures logging via Logback**

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here’s an example of a simple logback.

xml file you might use:

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>

%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n

</pattern>

</encoder>

</appender>

<logger name="root" level="INFO"/>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

To set the logging levels, you create properties that are prefixed with logging.level, followed by the name of the logger for which you want to set the logging level. For instance, suppose you’d like to set the root logging level to WARN, but log Spring Security logs at a DEBUG level. The following entries in application.yml will take care of that for you:

logging:

level:

root: WARN

org:

springframework:

security: DEBUG

Now suppose that you want to write the log entries to the file TacoCloud.log **at /var/logs/.** The logging.path and logging.file properties can help achieve that:

logging:

path: /var/logs/

file: TacoCloud.log

level:

root: WARN

org:

springframework:

security: DEBUG

***5.1.5 Using special property values***

When setting properties, you aren’t limited to declaring their values as hard-coded String and numeric values. Instead, you can derive their values from other configuration properties.

For example, suppose (for whatever reason) you want to set a property named greeting.welcome to echo the value of another property named spring.application.name. To achieve this, you could use the ${} placeholder markers when setting

greeting.welcome:

greeting:

welcome: ${spring.application.name}

***Consuming REST services***

A Spring application can consume a REST API with

 RestTemplate—A straightforward, synchronous REST client provided by the core Spring Framework.

 *Traverson*—A hyperlink-aware, synchronous REST client provided by Spring HATEOAS. Inspired from a JavaScript library of the same name.

 WebClient—A reactive, asynchronous REST client introduced in Spring 5.

**RestTemplate** provides many methods for interacting with REST resources.

delete(…) Performs an HTTP DELETE request on a resource at a specified URL

exchange(…) Executes a specified HTTP method against a URL, returning a

ResponseEntity containing an object mapped from the response body

execute(…) Executes a specified HTTP method against a URL, returning an object mapped from the response body

getForEntity(…) Sends an HTTP GET request, returning a ResponseEntity containing an object mapped from the response body

getForObject(…) Sends an HTTP GET request, returning an object mapped from a response body

headForHeaders(…) Sends an HTTP HEAD request, returning the HTTP headers for the specified resource URL

optionsForAllow(…) Sends an HTTP OPTIONS request, returning the Allow header for the specified URL

patchForObject(…) Sends an HTTP PATCH request, returning the resulting object mapped from the response body

postForEntity(…) POSTs data to a URL, returning a ResponseEntity containing an object mapped from the response body

postForLocation(…) POSTs data to a URL, returning the URL of the newly created resource postForObject(…) POSTs data to a URL, returning an object mapped from the response body

put(…) PUTs resource data to the specified URL

To use RestTemplate, you’ll either need to create an instance at the point you need it

RestTemplate rest = new RestTemplate();

or you can declare it as a bean and inject it where you need it:

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

If the client needs more than the payload body, you may want to consider using getForEntity():

getForEntity() works in much the same way as getForObject(), but instead of returning a domain object that represents the response’s payload, it returns a Response-Entity object that wraps that domain object. The ResponseEntity gives access to

additional response details, such as the response headers.

For example, suppose that in addition to the ingredient data, you want to inspect the Date header from the response. With getForEntity() that becomes straightforward:

public Ingredient **getIngredientById**(String ingredientId) {

ResponseEntity<Ingredient> responseEntity =

rest.getForEntity("http://localhost:8080/ingredients/{id}",Ingredient.class, ingredientId);

log.info("Fetched time: " + responseEntity.getHeaders().getDate());

return responseEntity.getBody();

}

The getForEntity() method is overloaded with the same parameters as getFor-Object(), so you can provide the URL variables as a variable list parameter or call getForEntity() with a URI object.

***Deploying Spring***

***Weighing deployment options***

You can build and run Spring Boot applications in several ways. The appendix covers many of them, including these:

 Running the application in the IDE with either Spring Tool Suite or IntelliJ IDEA

 Running the application from the command line using the Maven springboot: run goal or Gradle bootRun task

 Using Maven or Gradle to produce an executable JAR file that can be run at the command line or deployed in the cloud

 Using Maven or Gradle to produce a WAR file that can be deployed to a traditional Java application server

Given the options of deploying a WAR file or a JAR file, how do you choose? In general, the choice comes down to whether you plan to deploy your application to a traditional Java application server or to a cloud platform:

 *Deploying to Java application servers*—If you must deploy your application to Tomcat, WebSphere, WebLogic, or any other traditional Java application server, you really have no choice but to build your application as a WAR file.

 *Deploying to the cloud*—If you’re planning to deploy your application to the cloud, whether it be Cloud Foundry, Amazon Web Services (AWS), Azure, Google Cloud Platform, or most any other cloud platform, then an executable JAR file is the best choice. Even if the cloud platform supports WAR deployment, the JAR file format is much simpler than the WAR format, which is designed for application server deployment.

**\*\*\*\*\*** Si quiero un .war

If you’re building with Maven, the change required is as simple as ensuring that the <packaging> element in pom.xml is set to war:

<packaging>war</packaging>

The changes required for a Gradle build are similarly straightforward. You must apply the war plugin in the build.gradle file:

apply plugin: 'war'

Now you’re ready to build the application. With Maven, you’ll use the Maven wrapper script that the Initializr used to execute the package goal:

$ mvnw package

If the build is successful, then the WAR file can be found in the target directory. On the other hand, if you were using Gradle to build the project, you’d use the Gradle wrapper to execute the build task:

$ gradlew build

Once the build completes, the WAR file will be in the build/libs directory. All that’s left is to deploy the application. The deployment procedure varies across application servers, so consult the documentation for your application server’s specific deployment procedure.

It may be interesting to note that although you’ve built a WAR file suitable for deployment to any Servlet 3.0 (or higher) servlet container, the WAR file can still be executed at the command line as if it were an executable JAR file:

$ java -jar target/ingredient-service-0.0.19-SNAPSHOT.war

**\*\*\*\*\*** COMO COMENZAR UN PROYECTO CON CURL (POR CONSOLA) APUNTANDO A Spring Initializr

***curl and the Initializr API***

The simplest way to bootstrap a Spring project with curl is to consume the API like this:

% curl https://start.spring.io/starter.zip -o demo.zip

In this case, you’re requesting the /starter.zip endpoint from the Initializr, which will generate a Spring project and download it as a zip file. The generated project will be Maven-built and will have no dependencies other than the base Spring Boot starter dependency. All project information in the project’s pom.xml file will be set to default values.

If you don’t specify otherwise, the name of the file will be starter.zip. But in this case, the -o option specifies that the downloaded file should be named demo.zip.

making a simple request to the base Initializr URL: % curl https://start.spring.io

(muestra todas las opciones de creacion por consola)

The dependencies parameter is the one you’ll probably find the most useful. For example, suppose that you want to create a simple web project with Spring. The following command-line use of curl will produce a project zip with the web starter as a

dependency:

**> curl** [**https://start.spring.io/starter.zip**](https://start.spring.io/starter.zip)

**-d dependencies=web**

**-o demo.zip**

As a more complex example, suppose you wanted to develop a web application that uses Spring Data JPA for data persistence. You also want to build it with Gradle and the project should be under a directory named my-dir within the zip file. And let’s suppose that rather than just download a zip file, you want the project unpacked into your filesystem upon download. In that case, the following command should do

the trick:

**> curl** [**https://start.spring.io/starter.tgz**](https://start.spring.io/starter.tgz)

**-d dependencies=web,data-jpa**

**-d type=gradle-project**

**-d baseDir=my-dir | tar -xzvf -**

Entonces, con eso crea un Project (LLAMADO “my-dir) descomprimido ya con las dependencias que le hemos pedido!!!

***Building and running projects***

No matter how you initialize your project, you can always run the application from the command line with the java -jar command:

% java -jar demo.jar

This will even work if you decide to create a WAR file distribution instead of a JAR file:

% java -jar demo.war

You can also take advantage of the Spring Boot Maven and Gradle plugins to run your application. For example, if your project is built with Maven, you can run it like this:

% mvn spring-boot:run

If, on the other hand, you’ve chosen to build your project with Gradle, you can run your project like this:

% gradle bootRun

In either case, whether using Maven or Gradle, the build tool will first build your project (if it hasn’t already been built) and run it.

***Administering Spring: Using the Spring Boot Admin***

***Creating an Admin server***

To enable the Admin server, you’ll first need to create a new Spring Boot application and add the Admin server dependency to the project’s build. The Admin server is generally used as a standalone application, separate from any other application. Therefore, the easiest way to get started is to use the Spring Boot Initializr to create a new Spring Boot project and select the checkbox labeled Spring Boot Admin (Server). This results in the following dependency being included in the <dependencies> block:

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>**spring-boot-admin-starter-server**</artifactId>

</dependency>

Next, you’ll need to enable the Admin server by annotating the main configuration class with **@EnableAdminServer**

Finally, because the Admin server won’t be the only application running locally as it’s developed, you should set it to listen in on a unique port, but one you can easily access (not port 0, for example). Here, I’ve chosen port 9090 as the port for the Spring Boot Admin server:

server:

port: 8888

***Registering Admin clients***

Because the Admin server is an application separate from other Spring Boot application(s) for which it presents Actuator data, you must somehow make the Admin server aware of the applications it should display. Two ways to register Spring Boot Admin clients with the Admin server follow:

 Each application explicitly registers itself with the Admin server.

 The Admin server discovers services through the Eureka service registry.

EXPLICITLY CONFIGURING ADMIN CLIENT APPLICATIONS

In order for a Spring Boot application to register itself as a client of the Admin server, you must include the Spring Boot Admin client starter in its build. You can easily add this dependency to your build by selecting the checkbox labeled Spring Boot Admin (Client) in the Initializr, or you can set the following <dependency> for a Maven-built Spring Boot application:

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>**spring-boot-admin-starter-client**</artifactId>

</dependency>

With the client-side library in place, you’ll also need to configure the location of the Admin server so that the client can register itself. To do that, you’ll set the spring.boot.admin.client.url property to the root URL of the Admin server:

spring:

application:

name: **ingredient-service**

boot:

admin:

client:

url: http://localhost:9090

Notice that the spring.application.name property is also set (in this case, for the ingredient service). You’ve already used this property to identify microservices

**EJEMPLO PRACTICO**

**1) Creo mi SERVER**

**a. pom.xml**

<?xml version=**"1.0"** encoding=**"UTF-8"**?>

<project xmlns=**"http://maven.apache.org/POM/4.0.0"** xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

xsi:schemaLocation=**"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>

<modelVersion>**4.0.0**</modelVersion>

<parent>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-starter-parent**</artifactId>

<version>**2.2.5.RELEASE**</version>

<relativePath/><!-- lookup parent from repository -->

</parent>

<groupId>**com.example**</groupId>

<artifactId>**admin-server**</artifactId>

<version>**0.0.1-SNAPSHOT**</version>

<name>**admin-server**</name>

<description>**Demo project for Spring Boot**</description>

<properties>

<java.version>**1.8**</java.version>

<spring-boot-admin.version>**2.2.1**</spring-boot-admin.version>

</properties>

<dependencies>

<dependency>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-starter-web**</artifactId>

</dependency>

<dependency>

<groupId>**de.codecentric**</groupId>

<artifactId>**spring-boot-admin-starter-server**</artifactId>

</dependency>

<dependency>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-starter-test**</artifactId>

<scope>**test**</scope>

<exclusions>

<exclusion>

<groupId>**org.junit.vintage**</groupId>

<artifactId>**junit-vintage-engine**</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>**de.codecentric**</groupId>

<artifactId>**spring-boot-admin-dependencies**</artifactId>

<version>**${spring-boot-admin.version}**</version>

<type>**pom**</type>

<scope>**import**</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-maven-plugin**</artifactId>

</plugin>

</plugins>

</build>

</project>

**b. App class**

**@SpringBootApplication**

**@EnableAdminServer**

public class **AdminServerApplication** **{**

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

SpringApplication**.**run**(**AdminServerApplication**.**class**,** args**);**

**}**

**}**

**c. application.properties**

spring.application.name=spring-boot-admin

server.port=8888

**2) Creo mi CLIENT**

**a. pom.xml**

<dependencies>

<dependency>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-starter-web**</artifactId>

</dependency>

<dependency>

<groupId>**de.codecentric**</groupId>

<artifactId>**spring-boot-admin-starter-client**</artifactId>

</dependency>

<dependency>

<groupId>**org.springframework.boot**</groupId>

<artifactId>**spring-boot-starter-security**</artifactId>

</dependency>

<dependencies>

**b. creo mi app class**

**@SpringBootApplication**

**@RestController**

public class **AdminClientApplication** **{**

Logger log **=** LoggerFactory**.**getLogger**(**AdminClientApplication**.**class**);**

**@GetMapping("/message")**

public String **getMessage()** **{**

log**.**info**(**"Logging for message() method...."**);**

**return** "Spring boot admin client example"**;**

**}**

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

SpringApplication**.**run**(**AdminClientApplication**.**class**,** args**);**

**}**

**}**

**c. configuro mi SECURE class**

**import** org**.**springframework**.**context**.**annotation**.**Configuration**;**

**import** org**.**springframework**.**security**.**config**.**annotation**.**web**.**builders**.**HttpSecurity**;**

**import** org**.**springframework**.**security**.**config**.**annotation**.**web**.**configuration**.**WebSecurityConfigurerAdapter**;**

**@Configuration**

public class **SecurityPermitAllConfig** **extends** **WebSecurityConfigurerAdapter** **{**

**@Override**

protected void **configure(**HttpSecurity http**)** **throws** **Exception** **{**

**http.authorizeRequests().anyRequest().permitAll().and().csrf().disable();**

**}**

**}**

**d. configuro mi application.properties**

spring.boot.admin.client.url=http://localhost:**8888**

spring.application.name=**myfirstapplicationtest**

management.endpoints.web.exposure.include=\*

**Java CookBook**

1.1. Compiling and Running Java: JDK

Problem

You need to compile and run your Java program.

JDK

C:\javasrc>**javac HelloWorld.java**

C:\javasrc>**java HelloWorld**

Hello, World

There is an optional setting called CLASSPATH, discussed in Recipe 1.4, that controls where Java looks for classes. CLASSPATH, if set, is used by both javac and java. In older versions of Java, you had to set your CLASSPATH to include “.”

1.4. Using CLASSPATH Effectively

Problem

You need to keep your class files in a common directory, or you’re wrestling with CLASSPATH.

Solution

The class path is the path that the Java runtime environment searches for classes and other resource files. The class search path (more commonly known by the shorter name, "class path") can be set using either the -classpath option when calling a JDK tool (the preferred method) or by setting the CLASSPATH environment variable. The -classpath option is preferred because you can set it individually for each application without affecting other applications and without other applications modifying its value.

C:> sdkTool -classpath classpath1;classpath2...

-or-

C:> set CLASSPATH=classpath1;classpath2...

Set CLASSPATH to the list of directories and/or JAR files that contain the classes you want.

The CLASSPATH is used by the Java runtime to find classes. You can specify the CLASSPATH for a given command on the command line:

**C:\> java -classpath c:\ian\classes MyProg**

use of java -classpath or java -cp

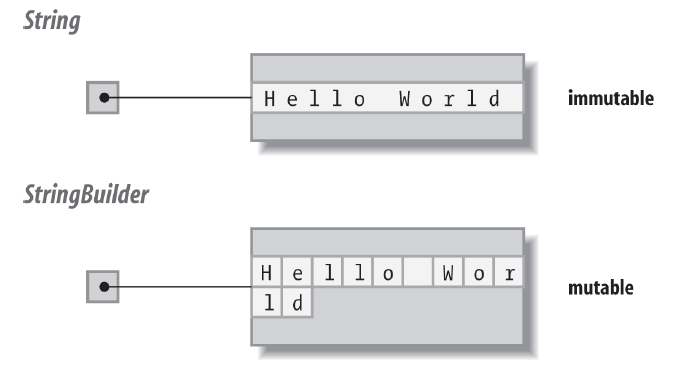
Example: your CLASSPATH were set to *C:* **\***classes*

Classpath in Java is the path to directory or list of the directory which is used by ClassLoaders to find and load class in Java program. Classpath can be specified using CLASSPATH environment variable which is case insensitive, -cp or -classpath command line option or Class-Path attribute in manifest.mf file inside JAR file in Java. 

Also don't confuse Classpath with PATH in Java, which is another environment variable used to find java binaries located in JDK installation directory, also known as JAVA\_HOME.

The main difference between PATH and CLASSPATH is that former is used to locate Java commands while later is used to locate Java class files. So let’s start with basic and then we will see some example and improvisation of Classpath in Java. In fact, CLASSPATH is an environment variable which is used by Java Virtual Machine to locate user defined classes.

**String vs StringBuilder**



Taking Strings Apart with Substrings

Problem

You want to break a string apart into substrings by position.

Solution

Use the String object’s substring() method.

The substring() method constructs a new String object made up of a run of characters

contained somewhere in the original string, the one whose substring() you called

String a **=** "Java is great."**;**

System**.**out**.**println**(**a**);**

String b **=** a**.**substring**(**5**);** // b is the String "is great."

System**.**out**.**println**(**b**);**

String c **=** a**.**substring**(**5**,**7**);** // c is the String "is"

System**.**out**.**println**(**c**);**

String d **=** a**.**substring**(**5**,**a**.**length**());** // d is "is great."

System**.**out**.**println**(**d**);**

**STRING TOKENIZER**

StringTokenizer st **=** **new** StringTokenizer**(**"Hello World"**);** // 2

System**.**out**.**println**(**st**.**countTokens**());**

**while** **(**st**.**hasMoreTokens**())**

System**.**out**.**println**(**"Token: " **+** st**.**nextToken**(** **));**

Token**:** Hello

Token**:** World

**STRING BUILDER**

String s1 **=** "Hello" **+** ", " **+** "World"**;**

System**.**out**.**println**(**s1**);**

// Build a StringBuilder, and append some things to it.

StringBuilder sb2 **=** **new** StringBuilder**();**

sb2**.**append**(**"Hello"**);**

sb2**.**append**(**','**);**

sb2**.**append**(**' '**);**

sb2**.**append**(**"World"**);**

// Get the StringBuilder's value as a String, and print it.

String s2 **=** sb2**.**toString**();**

System**.**out**.**println**(**s2**);**

// Now do the above all over again, but in a more concise (and typical "real-world" Java) fashion.

System**.**out**.**println**(**

**new** StringBuilder**()**

**.**append**(**"Hello"**)**

**.**append**(**','**)**

**.**append**(**' '**)**

**.**append**(**"World"**));**

**}**

…

Hello, World

Hello, World

Hello, World

===

private static final String SAMPLE\_STRING **=** "Hola hola esto es Carrusel Deportivo"**;**

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

// Method using regexp split

StringBuilder sb1 **=** **new** StringBuilder**();**

**for** **(**String word **:** SAMPLE\_STRING**.**split**(**" "**))** **{**

**if** **(**sb1**.**length**()** **>** 0**)** **{**

sb1**.**append**(**", "**);**

**}**

sb1**.**append**(**word**);**

**}**

System**.**out**.**println**(**sb1**);**

// Method using a StringTokenizer

StringTokenizer st **=** **new** StringTokenizer**(**SAMPLE\_STRING**);**

StringBuilder sb2 **=** **new** StringBuilder**();**

**while** **(**st**.**hasMoreElements**())** **{**

sb2**.**append**(**st**.**nextToken**());**

**if** **(**st**.**hasMoreElements**())** **{**

sb2**.**append**(**", "**);**

**}**

**}**

System**.**out**.**println**(**sb2**);**

**}**

…

Hola, hola, esto, es, Carrusel, Deportivo

Hola, hola, esto, es, Carrusel, Deportivo

**REGEX**

\^ Start of line/string

$ End of line/string

String pattern **=** "^Q[^u]\\d+\\."**;**

String**[]** input **=** **{**

"QA777. is the next flight. It is on time."**,**

"Quack, Quack, Quack!"

**};**

Pattern p **=** Pattern**.**compile**(**pattern**);**

**for** **(**String in **:** input**)** **{**

boolean found **=** p**.**matcher**(**in**).**lookingAt**();**

System**.**out**.**println**(**"'" **+** pattern **+** "'" **+**

**(**found **?** " matches '" **:** " doesn't match '"**)** **+** in **+** "'"**);**

**}**

**...**

'^Q[^u]\d+\.' matches 'QA777. is the next flight. It is on time.'

'^Q[^u]\d+\.' doesn't match 'Quack**,** Quack**,** Quack**!**'

The Matcher methods are:

match()

Used to compare the entire string against the pattern; this is the same as the routine

in java.lang.String. Because it matches the entire String, I had to put .\* before

and after the pattern.

lookingAt()

Used to match the pattern only at the beginning of the string.

find()

Used to match the pattern in the string (not necessarily at the first character of the

string), starting at the beginning of the string or, if the method was previously called

and succeeded, at the first character not matched by the previous match.

Each of these methods returns boolean, with true meaning a match and false meaning

no match. To check whether a given string matches a given pattern, you need only type

something like the following:

Matcher m = Pattern.compile(patt).matcher(line);

**if** (m.find( )) {

System.out.println(line + " matches " + patt)

}

But you may also want to extract the text that matched, which is the subject of the next

recipe.

Finding the Matching Text

Problem

You need to find the text that the regex matched.

ME VOY POR NUMBERS

**DEFENSIVE CODING**

**import** com.example.defense.model.Train;  
**import** com.example.defense.service.SearchRequest;  
**import** com.example.defense.service.TrainService;  
**import** org.springframework.web.bind.annotation.GetMapping;  
**import** org.springframework.web.bind.annotation.RequestBody;  
**import** org.springframework.web.bind.annotation.RestController;  
**import** java.util.Collection;  
**import** java.util.List;  
  
**@RestController**  
**public class DefenseController**{  
  
 **private** TrainService **trainService**;  
  
 **public** DefenseController(TrainService trainService){  
 **this**.**trainService** = trainService;  
 }  
  
 **@GetMapping("/trains")** **public** Collection<Train> **getTrains**(@**RequestBody** Train train){  
  
 SearchRequest searchRequest = **new** SearchRequest(train.getLine(), train.getDate(), train.getDestiny());  
  
 List<Train> found = **trainService**.search(searchRequest);  
  
 **return** *displayTrains*(found);  
 }  
  
 **private static** Collection<Train> **displayTrains**(List<Train> found){  
 **if**(found.isEmpty()){  
 **throw new** NullPointerException(**"There is no train match!"**);  
 } **else**{  
 **return** found;  
 }  
 }  
}

…

**import** com.example.defense.model.Train;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Service;  
**import** java.util.List;  
**import** java.util.stream.Collectors;  
  
**@Service  
public class TrainService** {  
  
 **@Autowired** **private** TrainStore **trainStore**;  
  
 **public TrainService**(TrainStore trainStore) { **this**.**trainStore** = trainStore; }  
  
 **public** List<Train> **search**(SearchRequest searchRequest){  
  
 List<Train> trains = **trainStore**.**getTrains**();  
  
 **return** trains.**stream**()  
 .filter(f -> f.getLine().equals(searchRequest.getLine()))  
 .collect(Collectors.*toList*());  
 }  
  
 **public** TrainService **trainSearch**(){  
 **return new** TrainService(**new** TrainStoreImpl());  
 }  
}

…

**import** com.example.defense.model.Train;  
**import** java.util.List;  
  
**public interface TrainStore** {  
 List<Train> getTrains();  
}

…

**import** com.example.defense.model.Train;  
**import** org.springframework.stereotype.Component;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**@Component  
public class TrainStoreImpl** **implements TrainStore**{  
  
 **private** List<Train> **trains**;  
  
 **@Override** **public** List<Train> **getTrains**(){  
  
 **trains** = **new** ArrayList<>();  
  
 Train train1 = **new** Train(10, **"15-06-2001"**, **"Mercado"**);  
 Train train2 = **new** Train(1, **"03-10-1999"**, **"Nazaret"**);  
 Train train3 = **new** Train(10, **"18-10-2006"**, **"San Antoni"**);  
 **trains**.add(train1);  
 **trains**.add(train2);  
 **trains**.add(train3);  
 **return trains**;  
  
 */\*return List.of (  
 new Train (10, "Mislata", "Mercado"),  
 new Train (1, "Mestalla", "Nazaret")  
 );\*/* }  
}

…

**import** lombok.Data;  
**import** lombok.NoArgsConstructor;  
**import** java.time.LocalDate;  
**import** java.time.format.DateTimeFormatter;  
**import** java.time.format.DateTimeParseException;  
**import** java.time.format.FormatStyle;  
  
**@Data  
@NoArgsConstructor  
public class SearchRequest**{  
  
 **private** Integer **line**;  
 **private** String **date**;  
 **private** String **destiny**;  
  
 **public SearchRequest**(Integer line, String date, String destiny){  
 **this**.**line** = validateLine(line);  
 **this**.**date** = validateInputDate(date);  
 **this**.**destiny** = validateString(destiny);  
 }  
  
 **private** Integer **validateLine**(Integer line){  
 **if**(line < 1 || line > 10){  
 **throw new IllegalArgumentException**(**"The line is between 1-10"**);  
 }  
 **return** line;  
 }  
  
 **private** String **validateInputDate**(String date){  
 LocalDate localDate;  
 **try**{  
 localDate = LocalDate.*parse*(date, DateTimeFormatter.*ofPattern*(**"dd-MM-yyyy"**));  
 } **catch**(DateTimeParseException ex) {  
 String msg = String.*format*(**"Could not parse this %s, porque es otro formato"**, date);  
 **throw new IllegalArgumentException**(msg, ex);  
 }  
 **return** localDate.format(DateTimeFormatter.*ofLocalizedDate*(FormatStyle.***FULL***)); *// LONG, MEDIUM, SHORT* }  
  
 **private** String validateString(String destiny){  
 **if**(**null** == destiny || destiny.trim().isEmpty() ){  
 **throw new** IllegalArgumentException(**"Origin and Destiny can't be null"**);  
 }  
 **return** destiny;  
 }  
}

…

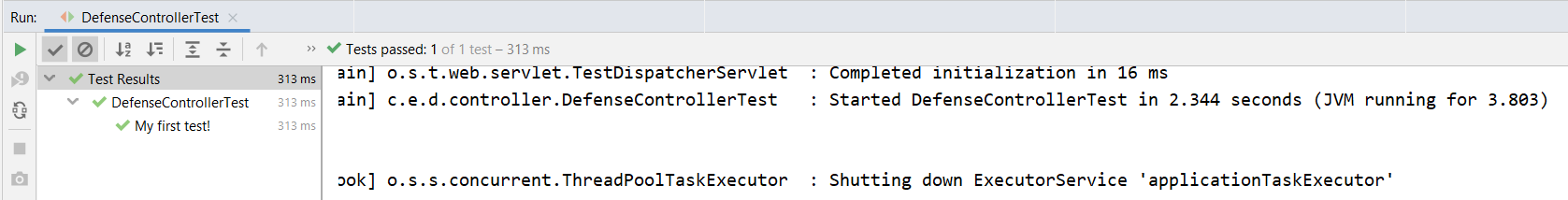
**import** lombok.AllArgsConstructor;  
**import** lombok.Builder;  
**import** lombok.Data;  
**import** lombok.NoArgsConstructor;  
  
**@Data  
@AllArgsConstructor  
@NoArgsConstructor  
@Builder  
public class** Train {  
 **private** Integer **line**;  
 **private** String **date**;  
 **private** String **destiny**;  
  
}

…

**import** com.example.defense.model.Train;  
**import** com.example.defense.service.SearchRequest;  
**import** com.example.defense.service.TrainService;  
**import** com.example.defense.service.TrainStore;  
**import** com.example.defense.service.TrainStoreImpl;  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** org.junit.jupiter.api.DisplayName;  
**import** org.junit.jupiter.api.Test;  
**import** org.junit.jupiter.api.extension.ExtendWith;  
**import** org.mockito.InjectMocks;  
**import** org.mockito.Mock;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;  
**import** org.springframework.boot.test.context.SpringBootTest;  
**import** org.springframework.boot.test.mock.mockito.MockBean;  
**import** org.springframework.http.MediaType;  
**import** org.springframework.test.context.junit.jupiter.SpringExtension;  
**import** org.springframework.test.web.servlet.MockMvc;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import static** org.hamcrest.Matchers.*is*;  
**import static** org.mockito.Mockito.*when*;  
**import static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*;  
**import static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*jsonPath*;  
**import static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;  
 **@ExtendWith(SpringExtension.class)  
@SpringBootTest  
@AutoConfigureMockMvc  
class DefenseControllerTest**{

**@InjectMocks**  
 **private** TrainService **trainService**; **// como trainService llama a un 3o, entonces tender que injector sus mocks!**

**@Mock  
 TrainStore trainStore = new TrainStoreImpl();** **// porque TrainStore es de 2a generacion, tengo que mockearle!**  
 **@Autowired** **private** MockMvc **mockMvc**;  
  
 List<Train> **trains**;  
  
 **@Test  
 @DisplayName("My first test!")** **void getTrains**() **throws Exception**{  
  
 **trains** = **new** ArrayList<>();  
 Train trainR = Train.*builder*().line(10).date(**"06-05-1999"**).destiny(**"Chicago"**).build();  
 **trains**.add(trainR);  
  
 Train train = Train.*builder*().line(10).date(**"03-10-1999"**).destiny(**"London"**).build();  
  
 SearchRequest searchRequest = **new** SearchRequest(10, **"22-09-2010"**, **"Liverpool"**);  
  
 *when*(**trainService**.search(searchRequest)).thenReturn(**trains**);  
  
 **mockMvc**.perform(*get*(**"/trains"**).contentType(MediaType.***APPLICATION\_JSON***).content(asJsonString(train)))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*((**"$[1].destiny"**), *is*(**"San Antoni"**)));  
 }  
  
 **private** String **asJsonString**(Object object){  
 **try**{  
 **return new** ObjectMapper().writeValueAsString(object);  
 } **catch**(JsonProcessingException e){  
 e.printStackTrace();  
 **throw new** RuntimeException();  
 }  
 }  
}



**TDD --- Lynda.com**

**package (src/test/java)** com.tdd.lynda;  
**import** org.junit.Before;  
**import** org.junit.Ignore;  
**import** org.junit.Test;  
**import static** org.junit.Assert.*assertEquals*;  
**import static** org.junit.Assert.*assertTrue*;  
  
**public class** BankAccountTest{  
  
 */\* Creo una BankAccount:  
 Deposit 50;  
 Balance 50;  
 Withdraw 30;  
 Balance 20; --> variable: balance, methods: deposit and withdraw \*/* BankAccount **bankAccount**;  
  
 @Before  
 **public void** setUp(){  
 **bankAccount** = **new** BankAccount();  
 **bankAccount**.deposit(100);  
 }  
  
 @Test  
 **public void** testDeposit(){  
  
 **bankAccount**.deposit(100);  
 *assertEquals*(200, **bankAccount**.getBalance());  
  
 }  
  
 @Test  
 @Ignore  
 **public void** testWithdraw(){  
 **bankAccount**.withdraw(50);  
 *assertEquals*(50, **bankAccount**.getBalance());  
 }  
  
}

…

**package (src/main/java)** com.tdd.lynda;  
  
**public class** BankAccount{  
  
 **private int balance**;  
  
 **public int** getBalance(){  
 **return balance**;  
 }  
  
 **public void** deposit(**int** integer){  
 **this**.**balance** += integer;  
 }  
  
  
 **public void** withdraw(**int** integer){  
 **this**.**balance** -= integer;  
  
 }  
}

…

build.gradle

plugins **{** id **'java'  
}**group **'com.example'**version **'1.0-SNAPSHOT'**sourceCompatibility = 1.8  
  
repositories **{** mavenCentral()  
**}**dependencies **{** testCompile **group**: **'junit'**, **name**: **'junit'**, **version**: **'4.12'  
}**apply **plugin**: **'jacoco'***// check.dependsOn jacocoTestCoverageVerification -- ENABLE PARA EJECUTAR EL CHEQUEO CON CLEAN BUILD!*jacocoTestCoverageVerification **{** violationRules **{** rule **{** limit **{** minimum = 0.7  
 **}  
 }  
 }  
}**

…

readme.txt

From the Project dialog, right click to bring the context menu and select New | Directory.  
Type src/test/java/com/tddjava and click on the OK button to create the tests package.  
Repeat the same steps with the src/main/java/com/tddjava directory to create the implementation package.  
  
DIRECTORIES: 1) src/main/java  
 2) src/test/java

**Test-Driven Java Development, Viktor Farcic, 2015 Packt Publishing**

**1- HOW TO SET A RULE**

**import** org.junit.Before;  
**import** org.junit.Ignore;  
**import** org.junit.Rule;  
**import** org.junit.Test;  
**import** org.junit.rules.ExpectedException;  
  
**public class FooTest**{  
  
 **@Rule** **public** ExpectedException **exception** = ExpectedException.*none*();  
  
 **private** Foo **foo**;  
  
 **@Before** **public final void setUp**() {  
 **foo** = **new** Foo();  
 }  
  
 **@Test** **public void whenDoFooThenThrowRuntimeException**() {  
 **exception**.expect(RuntimeException.**class**);  
 **foo**.doFoo();  
 }  
}

…

**public class Foo**{  
 **public void doFoo**(){  
 **throw new RuntimeException**();  
 }  
}

**TIC-TAC-TOE GAME DEVELOPMENT**

**Requirement 1**

**=============**

We should start by defining the boundaries and what constitutes an invalid placement of a piece**.** A piece can be placed on any empty space of a 3×3 board**.**

We can split this requirement into three tests:

• When a piece is placed anywhere outside the X axis**,** then RuntimeException is thrown**.**

• When a piece is placed anywhere outside the Y axis**,** then RuntimeException is thrown**.**

**TEST**

public class **TicTacToeSpec** **{**

**@Rule**

public ExpectedException exception **=** ExpectedException**.**none**();**

private TicTacToe ticTacToe**;**

**@Before**

public final void **setUp()** **{**

ticTacToe **=** **new** TicTacToe**();**

**}**

**@Test**

public void **whenXOutsideBoardThenRuntimeException()** **{**

exception**.**expect**(RuntimeException.**class**);**

ticTacToe**.**play**(**5**,** 2**);**

**}**

**}**

**IMPLEMENTATION**

public class **TicTacToe** **{**

public void **play(**int x**,** int y**)** **{**

**if** **(**x **<** 1 **||** x **>** 3**)** **{**

**throw** **new** **RuntimeException(**"X is outside board"**);**

**}** **else** **if** **(**y **<** 1 **||** y **>** 3**)** **{**

**throw** **new** **RuntimeException(**"X is outside board"**);**

**}**

**}**

**}**

• When a piece is placed on an occupied space**,** then RuntimeException is thrown**.**

public class **TicTacToe** **{**

public void **play(**int x**,** int y**)** **{**

private Character**[][]** board **=** **{{**'\0'**,** '\0'**,** '\0'**},** **{**'\0'**,** '\0'**,** \0'**},** **{**'\0'**,** '\0'**,** '\0'**}};**

public void **play(**int x**,** int y**)** **{**

**if** **(**x **<** 1 **||** x **>** 3**)** **{**

**throw** **new** **RuntimeException(**"X is outside board"**);**

**}** **else** **if** **(**y **<** 1 **||** y **>** 3**)** **{**

**throw** **new** **RuntimeException(**"Y is outside board"**);**

**}**

**if** **(**board**[**x **-** 1**][**y **-** 1**]** **!=** '\0'**)** **{**

**throw** **new** **RuntimeException(**"Box is occupied"**);**

**}** **else** **{**

board**[**x **-** 1**][**y **-** 1**]** **=** 'X'**;**

**}**

**}**

**}**

**}**

**REFACTOR**

public void **play(**int x**,** int y**)** **{**

**checkAxis(x);**

**checkAxis(y);**

**setBox(x, y);**

**}**

private void **checkAxis(**int axis**)** **{**

**if** **(**axis **<** 1 **||** axis **>** 3**)** **{**

**throw** **new** **RuntimeException(**"X is outside board"**);**

**}**

**}**

private void **setBox(**int x**,** int y**)** **{**

**if** **(**board**[**x **-** 1**][**y **-** 1**]** **!=** '\0'**)** **{**

**throw** **new** **RuntimeException(**"Box is occupied"**);**

**}** **else** **{**

board**[**x **-** 1**][**y **-** 1**]** **=** 'X'**;**

**}**

**}**

**Requirement 2**

**=============**

Until this moment, we haven't used any of the JUnit's asserts. To use them, we need to **import** the static methods from the **org.junit.Assert** class**:**

**import static org.junit.Assert.\*;**

Now it's time to work on the specification of which player is about to play his turn.

There should be a way to find out which player should play next.

We can split this requirement into three tests:

**• The first turn should be played by played X**

**• If the last turn was played by X, then the next turn should be played by O**

**• If the last turn was played by O, then the next turn should be played by X**

**TEST**

**@Test**

public void **givenFirstTurnWhenNextPlayerThenX()** **{**

assertEquals**(**'X'**,** ticTacToe**.**nextPlayer**());**

**}**

The first turn should be played by player X**.**

**IMPLEMENTATION**

public **char** **nextPlayer()** **{**

**return** 'X'**;**

**}**

**TEST**

**@Test**

public void **givenLastTurnWasXWhenNextPlayerThenO()** **{**

ticTacToe**.**play**(**1**,** 1**);**

assertEquals**(**'O'**,** ticTacToe**.**nextPlayer**());**

**}**

If the last turn was played by X**,** then the next turn should be played by O**.**

**IMPLEMENTATION**

private char **lastPlayer** **=** '\0'**;**

public void **play(**int x**,** int y**)** **{**

**checkAxis(x);**

**checkAxis(y);**

**setBox(x, y);**

**lastPlayer = nextPlayer();**

**}**

public char **nextPlayer()** **{**

**if (lastPlayer == 'X') {**

**return 'O';**

**}**

**return 'X';**

**}**

**Requirement 3**

**=============**

It's time to work on winning the rules of the game. This is the part where, when compared with the previous code, work becomes a bit more tedious. We should check all the possible winning combinations and, if one of them is fulfilled, declare

a winner**.**

• A player wins by being the first to connect a line of friendly pieces from one side or corner of the board to the other**.**

To check whether a line of friendly pieces is connected**,** we should verify horizontal**,** vertical**,** and diagonal lines**.**

**TEST**

@Test

public void **whenPlayThenNoWinner()** **{**

String actual **=** ticTacToe**.play(**1**,**1**);**

assertEquals**(**"No winner"**,** actual**);**

**}**

If no winning condition is fulfilled**,** then there is no winner**.**

**IMPLEMENTATION**

public String **play(**int x**,** int y**)** **{**

**checkAxis(x);**

**checkAxis(y);**

**setBox(x, y);**

**lastPlayer = nextPlayer();**

**return "No winner";**

**}**

**===**

**TEST**

Now that we have declared what the default response is (no winner), it's time to start working on different winning conditions:

**@Test**

public void **whenPlayAndWholeHorizontalLineThenWinner()** **{**

ticTacToe**.**play**(**1**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 1**);** // X

ticTacToe**.**play**(**2**,** 2**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 1**);** // X

**assertEquals("X is the winner", actual);**

**}**

The player wins when the whole horizontal line is occupied by his pieces**.**

**IMPLEMENTATION**

To fulfill this test, we need to check whether any horizontal line is filled by the same mark as the current player. Until this moment, we didn't care what was put to the board array. Now, we need to introduce not only which board boxes are empty,

but also which player played them:

public String **play(**int x**,** int y**)** **{**

**checkAxis(x);**

**checkAxis(y);**

**lastPlayer = nextPlayer();**

**setBox(**x**,** y**,** **lastPlayer);**

**for** **(**int index **=** 0**;** index **<** 3**;** index**++)** **{**

**if** **(**board**[**0**][**index**]** **==** lastPlayer **&&** board**[**1**][**index**]** **==** lastPlayer **&&** board**[**2**][**index**]** **==** lastPlayer**)** **{**

**return** lastPlayer **+** " is the winner"**;**

**}**

**}**

**return** "No winner"**;**

**}**

private void **setBox(**int x**,** int y**,** char lastPlayer**)** **{**

**if** **(**board**[**x **-** 1**][**y **-** 1**]** **!=** '\0'**)** **{**

**throw** **new** **RuntimeException(**"Box is occupied"**);**

**}** **else** **{**

board**[**x **-** 1**][**y **-** 1**]** **=** lastPlayer**;**

**}**

**}**

**REFACTOR**

private static final int SIZE **=** 3**;**

public String **play(**int x**,** int y**)** **{**

**checkAxis(x);**

**checkAxis(y);**

**lastPlayer = nextPlayer();**

**setBox(x, y, lastPlayer);**

**if** **(isWin())** **{**

**return** lastPlayer **+** " is the winner"**;**

**}**

**return** "No winner"**;**

**}**

private boolean **isWin()** **{**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

**if** **(**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**]** **==** **(**lastPlayer **\*** SIZE**))** **{**

**return** **true;**

**}**

**}**

**return** **false;**

**}**

**TEST**

We should also check whether there is a win by filling the vertical line**:**

**@Test**

public void **whenPlayAndWholeVerticalLineThenWinner()** **{**

ticTacToe**.**play**(**2**,** 1**);** // X

ticTacToe**.**play**(**1**,** 1**);** // O

ticTacToe**.**play**(**3**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

String actual **=** ticTacToe**.**play**(**1**,** 3**);** // O

assertEquals**(**"O is the winner"**,** actual**);**

**}**

The player wins when the whole vertical line is occupied by his pieces**.**

**IMPLEMENTATION**

This implementation should be similar to the previous one. We already have horizontal verification and now we need to do the same vertically:

private boolean **isWin()** **{**

int playerTotal **=** lastPlayer **\*** 3**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

**if** **(**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**]** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **)** **{**

**return** **true;**

**}**

**}**

**return** **false;**

**}**

**TEST**

Now that horizontal and vertical lines are covered**,** we should move our attention to diagonal combinations**:**

**@Test**

public void **whenPlayAndTopBottomDiagonalLineThenWinner()** **{**

ticTacToe**.**play**(**1**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

ticTacToe**.**play**(**1**,** 3**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 3**);** // O

assertEquals**(**"X is the winner"**,** actual**);**

**}**

The player wins when the whole diagonal line from the top**-**left to bottom**-**right is occupied by his pieces**.**

**IMPLEMENTATION**

private boolean **isWin()** **{**

int playerTotal **=** lastPlayer **\*** 3**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

**if** **(**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**]** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **)** **{**

**return** **true;**

**}**

**}**

**if** **((**board**[**0**][**0**]** **+** board**[**1**][**1**]** **+** board**[**2**][**2**])** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **(**board**[**0**][**2**]** **+** board**[**1**][**1**]** **+** board**[**2**][**0**]))** **{**

**return** **true;**

**}**

**return** **false;**

**}**

**TEST**

Finally**,** there is the last possible winning condition to tackle**:**

@Test

public void **whenPlayAndBottomTopDiagonalLineThenWinner()** **{**

ticTacToe**.**play**(**1**,** 3**);** // X

ticTacToe**.**play**(**1**,** 1**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 1**);** // O

assertEquals**(**"X is the winner"**,** actual**);**

**}**

The player wins when the whole diagonal line from the bottom**-**left to top**-**right is occupied by his pieces

**IMPLEMENTATION**

The implementation of this test should be almost the same as the previous one:

private boolean **isWin()** **{**

int playerTotal **=** lastPlayer **\*** 3**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

**if** **(**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**]** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **)** **{**

**return** **true;**

**}**

**}**

**if** **((**board**[**0**][**0**]** **+** board**[**1**][**1**]** **+** board**[**2**][**2**])** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **(**board**[**0**][**2**]** **+** board**[**1**][**1**]** **+** board**[**2**][**0**]))** **{**

**return** **true;**

**}**

**return** **false;**

**}**

**REFACTOR**

private boolean isWin**()** **{**

int playerTotal **=** lastPlayer **\*** 3**;**

char diagonal1 **=** '\0'**;**

char diagonal2 **=** '\0'**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

diagonal1 **+=** board**[**i**][**i**];**

diagonal2 **+=** board**[**i**][**SIZE **-** i **-** 1**];**

**if** **(**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**])** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **(**playerTotal **==** **)** **{**

**return** **true;**

**}**

**}**

**if** **(**diagonal1 **==** playerTotal **||** diagonal2 **==** playerTotal**)** **{**

**return** **true;**

**}**

**return** **false;**

**}**

FINAL CODE IN STASH

**===================**

private boolean isWin**()** **{**

int playerTotal **=** lastPlayer **\*** SIZE**;**

char diagonal1 **=** '\0'**;**

char diagonal2 **=** '\0'**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

diagonal1 **+=** board**[**i**][**i**];**

diagonal2 **+=** board**[**i**][**SIZE **-** i **-** 1**];**

**if** **((**board**[**0**][**i**]** **+** board**[**1**][**i**]** **+** board**[**2**][**i**])** **==** playerTotal**)** **{**

**return** **true;**

**}** **else** **if** **((**board**[**i**][**0**]** **+** board**[**i**][**1**]** **+** board**[**i**][**2**])** **==** playerTotal**)** **{**

**return** **true;**

**}**

**}**

**if** **(**diagonal1 **==** playerTotal **||** diagonal2 **==** playerTotal**)** **{**

**return** **true;**

**}**

**return** **false;**

**}**

Requirement 4

The only thing missing is how to tackle the draw result**.**

• The result is a draw when all the boxes are filled**.**

@Test

public void whenAllBoxesAreFilledThenDraw**()** **{**

ticTacToe**.**play**(**1**,** 1**);**

ticTacToe**.**play**(**1**,** 2**);**

ticTacToe**.**play**(**1**,** 3**);**

ticTacToe**.**play**(**2**,** 1**);**

ticTacToe**.**play**(**2**,** 3**);**

ticTacToe**.**play**(**2**,** 2**);**

ticTacToe**.**play**(**3**,** 1**);**

ticTacToe**.**play**(**3**,** 3**);**

String actual **=** ticTacToe**.**play**(**3**,** 2**);**

assertEquals**(**"The result is draw"**,** actual**);**

**}**

---

private boolean isDraw**()** **{**

**for** **(**int x **=** 0**;** x **<** SIZE**;** x**++)** **{**

**for** **(**int y **=** 0**;** y **<** SIZE**;** y**++)** **{**

**if** **(**board**[**x**][**y**]** **==** '\0'**)** **{**

**return** **false;**

**}**

**}**

**}**

**return** **true;**

**}**

========================================================

FINAL CODE

**import** org**.**junit**.**Before**;**

**import** org**.**junit**.**Rule**;**

**import** org**.**junit**.**Test**;**

**import** org**.**junit**.**rules**.**ExpectedException**;**

**import** static org**.**junit**.**Assert**.\*;**

public class TicTacToeSpec **{**

@Rule

public ExpectedException exception **=** ExpectedException**.**none**();**

private TicTacToe ticTacToe**;**

@Before

public final void before**()** **{**

ticTacToe **=** **new** TicTacToe**();**

**}**

@Test

public void whenXOutsideBoardThenRuntimeException**()** **{**

exception**.**expect**(**RuntimeException**.**class**);**

ticTacToe**.**play**(**5**,** 2**);**

**}**

@Test

public void whenYOutsideBoardThenRuntimeException**()** **{**

exception**.**expect**(**RuntimeException**.**class**);**

ticTacToe**.**play**(**2**,** 5**);**

**}**

@Test

public void whenOccupiedThenRuntimeException**()** **{**

ticTacToe**.**play**(**2**,** 1**);**

exception**.**expect**(**RuntimeException**.**class**);**

ticTacToe**.**play**(**2**,** 1**);**

**}**

@Test

public void givenFirstTurnWhenNextPlayerThenX**()** **{**

assertEquals**(**'X'**,** ticTacToe**.**nextPlayer**());**

**}**

@Test

public void givenLastTurnWasXWhenNextPlayerThenO**()** **{**

ticTacToe**.**play**(**1**,** 1**);**

assertEquals**(**'O'**,** ticTacToe**.**nextPlayer**());**

**}**

@Test

public void whenPlayThenNoWinner**()** **{**

String actual **=** ticTacToe**.**play**(**1**,** 1**);**

assertEquals**(**"No winner"**,** actual**);**

**}**

@Test

public void whenPlayAndWholeHorizontalLineThenWinner**()** **{**

ticTacToe**.**play**(**1**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 1**);** // X

ticTacToe**.**play**(**2**,** 2**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 1**);** // X

assertEquals**(**"X is the winner"**,** actual**);**

**}**

@Test

public void whenPlayAndWholeVerticalLineThenWinner**()** **{**

ticTacToe**.**play**(**2**,** 1**);** // X

ticTacToe**.**play**(**1**,** 1**);** // O

ticTacToe**.**play**(**3**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

String actual **=** ticTacToe**.**play**(**1**,** 3**);** // O

assertEquals**(**"O is the winner"**,** actual**);**

**}**

@Test

public void whenPlayAndTopBottomDiagonalLineThenWinner**()** **{**

ticTacToe**.**play**(**1**,** 1**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

ticTacToe**.**play**(**1**,** 3**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 3**);** // O

assertEquals**(**"X is the winner"**,** actual**);**

**}**

@Test

public void whenPlayAndBottomTopDiagonalLineThenWinner**()** **{**

ticTacToe**.**play**(**1**,** 3**);** // X

ticTacToe**.**play**(**1**,** 1**);** // O

ticTacToe**.**play**(**2**,** 2**);** // X

ticTacToe**.**play**(**1**,** 2**);** // O

String actual **=** ticTacToe**.**play**(**3**,** 1**);** // O

assertEquals**(**"X is the winner"**,** actual**);**

**}**

@Test

public void whenAllBoxesAreFilledThenDraw**()** **{**

ticTacToe**.**play**(**1**,** 1**);**

ticTacToe**.**play**(**1**,** 2**);**

ticTacToe**.**play**(**1**,** 3**);**

ticTacToe**.**play**(**2**,** 1**);**

ticTacToe**.**play**(**2**,** 3**);**

ticTacToe**.**play**(**2**,** 2**);**

ticTacToe**.**play**(**3**,** 1**);**

ticTacToe**.**play**(**3**,** 3**);**

String actual **=** ticTacToe**.**play**(**3**,** 2**);**

assertEquals**(**"The result is draw"**,** actual**);**

**}**

**}**

…

public class TicTacToe **{**

private Character**[][]** board **=** **{{**'\0'**,** '\0'**,** '\0'**},** **{**'\0'**,** '\0'**,** '\0'**},** **{**'\0'**,** '\0'**,** '\0'**}};**

private char lastPlayer **=** '\0'**;**

private static final int SIZE **=** 3**;**

public String play**(**int x**,** int y**)** **{**

checkAxis**(**x**);**

checkAxis**(**y**);**

lastPlayer **=** nextPlayer**();**

setBox**(**x**,** y**,** lastPlayer**);**

**if** **(**isWin**(**x**,** y**))** **{**

**return** lastPlayer **+** " is the winner"**;**

**}** **else** **if** **(**isDraw**())** **{**

**return** "The result is draw"**;**

**}** **else** **{**

**return** "No winner"**;**

**}**

**}**

public char nextPlayer**()** **{**

**if** **(**lastPlayer **==** 'X'**)** **{**

**return** 'O'**;**

**}**

**return** 'X'**;**

**}**

private void checkAxis**(**int axis**)** **{**

**if** **(**axis **<** 1 **||** axis **>** SIZE**)** **{**

**throw** **new** RuntimeException**(**"X is outside board"**);**

**}**

**}**

private void setBox**(**int x**,** int y**,** char lastPlayer**)** **{**

**if** **(**board**[**x **-** 1**][**y **-** 1**]** **!=** '\0'**)** **{**

**throw** **new** RuntimeException**(**"Box is occupied"**);**

**}** **else** **{**

board**[**x **-** 1**][**y **-** 1**]** **=** lastPlayer**;**

**}**

**}**

private boolean isWin**(**int x**,** int y**)** **{**

int playerTotal **=** lastPlayer **\*** SIZE**;**

char horizontal**,** vertical**,** diagonal1**,** diagonal2**;**

horizontal **=** vertical **=** diagonal1 **=** diagonal2 **=** '\0'**;**

**for** **(**int i **=** 0**;** i **<** SIZE**;** i**++)** **{**

horizontal **+=** board**[**i**][**y **-** 1**];**

vertical **+=** board**[**x **-** 1**][**i**];**

diagonal1 **+=** board**[**i**][**i**];**

diagonal2 **+=** board**[**i**][**SIZE **-** i **-** 1**];**

**}**

**if** **(**horizontal **==** playerTotal

**||** vertical **==** playerTotal

**||** diagonal1 **==** playerTotal

**||** diagonal2 **==** playerTotal**)** **{**

**return** **true;**

**}**

**return** **false;**

**}**

private boolean isDraw**()** **{**

**for** **(**int x **=** 0**;** x **<** SIZE**;** x**++)** **{**

**for** **(**int y **=** 0**;** y **<** SIZE**;** y**++)** **{**

**if** **(**board**[**x**][**y**]** **==** '\0'**)** **{**

**return** **false;**

**}**

**}**

**}**

**return** **true;**

**}**

**}**

**CONNECT 4 REQUIREMENTS IMPLEMENTATION**

**Requirements**

In order to code the two implementations of Connect4, the game rules are transcribed below in the form of requirements. These requirements are the starting point for both the developments. We will go through the code with some explanations and compare both implementations at the end:

1. The board is composed of seven columns and six rows, all positions are emptys.

The implementation of this requirement is pretty straightforward. We just need the representation of an empty position and the data structure to hold the game. Note that the colors used by players are also defined:

public class Connect4 **{**

public enum Color **{**

RED**(**'R'**),** GREEN**(**'G'**),** EMPTY**(**' '**);**

private final char value**;**

Color**(**char value**)** **{** **this.**value **=** value**;** **}**

@Override

public String toString**()** **{**

**return** String**.**valueOf**(**value**);**

**}**

**}**

public static final int COLUMNS **=** 7**;**

public static final int ROWS **=** 6**;**

private Color**[][]** board **=** **new** Color**[**COLUMNS**][**ROWS**];**

public Connect4**()** **{**

**for** **(**Color**[]** column **:** board**)** **{**

Arrays**.**fill**(**column**,** Color**.**EMPTY**);**

**}**

**}**

**}**

1. Players introduce discs on the top of the columns. The introduced disc drops down the board if the column is empty. Future discs introduced in the same column will stack over the previous ones.

In this part, board bounds become relevant. We need to mark what positions are already taken, using Color.RED to indicate them. Finally, the first private method is created. It is a helper method that calculates the number of discs introduced in a given column:

public void putDisc**(**int column**)** **{**

**if** **(**column **>** 0 **&&** column **<=** COLUMNS**)** **{**

int numOfDiscs **=** getNumberOfDiscsInColumn**(**column **-** 1**);**

**if** **(**numOfDiscs **<** ROWS**)** **{**

board**[**column **-** 1**][**numOfDiscs**]** **=** Color**.**RED**;**

**}**

**}**

**}**

private int getNumberOfDiscsInColumn**(**int column**)** **{**

**if** **(**column **>=** 0 **&&** column **<** COLUMNS**)** **{**

int row**;**

**for** **(**row **=** 0**;** row **<** ROWS**;** row**++)** **{**

**if** **(**Color**.**EMPTY **==** board**[**column**][**row**])** **{**

**return** row**;**

**}**

**}**

**return** row**;**

**}**

**return** **-**1**;**

**}**

1. It is a two-person game, so there is one color for each player. One player uses red ('R') and the other one uses green ('G'). Players alternate turns, inserting one disc every time.

We need to save the current player in order to determine which player is playing this turn. We also need a function to switch the players in order to recreate the logic of turns. Some lines of code become relevant in the putDisc function. Specifically, the board position assignment is made using the current player, and it is switched afterevery move, as the game rules say:

**...**

private Color currentPlayer **=** Color**.**RED**;**

private void switchPlayer**()** **{**

**if** **(**Color**.**RED **==** currentPlayer**)**

currentPlayer **=** Color**.**GREEN**;**

**}** **else** **{**

currentPlayer **=** Color**.**RED**;**

**}**

**}**

public void putDisc**(**int column**)** **{**

**if** **(**column **>** 0 **&&** column **<=** COLUMNS**)** **{**

int numOfDiscs **=** getNumberOfDiscsInColumn**(**column **-** 1**);**

**if** **(**numOfDiscs **<** ROWS**)** **{**

board**[**column **-** 1**][**numOfDiscs**]** **=** currentPlayer**;**

switchPlayer**();**

**}**

**}**

**}**

**...**

1. We want feedback when either an event or an error occurs within the game. The output shows the status of the board after every move.

No output channel is specified. To make it easier, we decided to use the system standard output to print an event when it occurs. A few lines have been added on every action to let the user know about the status of the game:

**...**

private static final String DELIMITER **=** "|"**;**

private void switchPlayer**()** **{**

**if** **(**Color**.**RED **==** currentPlayer**)** **{**

currentPlayer **=** Color**.**GREEN**;**

**}** **else** **{**

currentPlayer **=** Color**.**RED**;**

**}**

System**.**out**.**println**(**"Current turn: " **+** currentPlayer**);**

**}**

public void printBoard**()** **{**

**for** **(**int row **=** ROWS **-** 1**;** row **>=** 0**;** **--**row**)** **{**

StringJoiner stringJoiner **=** **new** StringJoiner**(**DELIMITER**,** DELIMITER**,** DELIMITER**);**

**for** **(**int col **=** 0**;** col **<** COLUMNS**;** **++**col**)** **{**

stringJoiner**.**add**(**board**[**col**][**row**].**toString**());**

**}**

System**.**out**.**println**(**

stringJoiner**.**toString**());**

**}**

**}**

public void putDisc**(**int column**)** **{**

**if** **(**column **>** 0 **&&** column **<=** COLUMNS**)** **{**

int numOfDiscs **=** getNumberOfDiscsInColumn**(**column **-** 1**);**

**if** **(**numOfDiscs **<** ROWS**)** **{**

board**[**column **-** 1**][**numOfDiscs**]** **=** currentPlayer**;**

printBoard**();**

switchPlayer**();**

**}** **else** **{**

System**.**out**.**println**(**numOfDiscs**);**

System**.**out**.**println**(**"There's no room " **+** "for a new disc in this column"**);**

printBoard**();**

**}**

**}** **else** **{**

System**.**out**.**println**(**"Column out of bounds"**);**

printBoard**();**

**}**

**}**

**...**

1. When no more discs can be inserted, the game finishes, and it is considered a draw.

**...**

public boolean isFinished**()** **{**

int numOfDiscs **=** 0**;**

**for** **(**int col **=** 0**;** col **<** COLUMNS**;** **++**col**)** **{**

numOfDiscs **+=** getNumberOfDiscsInColumn**(**col**);**

**}**

**if** **(**numOfDiscs **>=** COLUMNS **\*** ROWS**)** **{**

System**.**out**.**println**(**"It's a draw"**);**

**return** **true;**

**}**

**return** **false;**

**}**

**...**

1. If a player inserts a disc and connects more than three discs of his color in a straight vertical line, then that player wins.

The checkWinCondition private method implements this rule by scanning whether or not the last move is a winning one:

**...**

private Color winner**;**

public static final int DISCS\_FOR\_WIN **=** 4**;**

public void putDisc**(**int column**)** **{**

**...**

**if** **(**numOfDiscs **<** ROWS**)** **{**

board**[**column **-** 1**][**numOfDiscs**]** **=**

currentPlayer**;**

printBoard**();**

checkWinCondition**(**column – 1**,**

numOfDiscs**);**

switchPlayer**();**

**...**

**}**

private void checkWinCondition**(**int col**,** int row**)** **{**

Pattern winPattern **=** Pattern**.**compile**(**".\*" **+** currentPlayer **+** "{" **+** DISCS\_FOR\_WIN **+** "}.\*"**);**

// Vertical check

StringJoiner stringJoiner **=** **new** StringJoiner**(**""**);**

**for** **(**int auxRow **=** 0**;** auxRow **<** ROWS**;** **++**auxRow**)** **{**

stringJoiner**.**add**(**board**[**col**][**auxRow**].**toString**());**

**}**

**if** **(**winPattern**.**matcher**(**stringJoiner**.**toString**()).**matches**())** **{**

winner **=** currentPlayer**;**

System**.**out**.**println**(**currentPlayer **+** " wins"**);**

**}**

**}**

public boolean isFinished**()** **{**

**if** **(**winner **!=** **null)** **return** **true;**

**...**

**}**

**...**

1. The same happens in a horizontal line direction.
2. The same happens in a diagonal line direction.

**FULL CODE**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**Scanner**;**

**import** java**.**util**.**StringJoiner**;**

**import** java**.**util**.**regex**.**Pattern**;**

/\*\*

\* Test-last implementation

\*/

public class Connect4 **{**

public enum Color **{**

RED**(**'R'**),** GREEN**(**'G'**),** EMPTY**(**' '**);**

private final char value**;**

Color**(**char value**)** **{**

**this.**value **=** value**;**

**}**

@Override

public String toString**()** **{**

**return** String**.**valueOf**(**value**);**

**}**

**}**

private Color winner**;**

private Color currentPlayer **=** Color**.**RED**;**

public static final int COLUMNS **=** 7**;**

public static final int ROWS **=** 6**;**

public static final int DISCS\_FOR\_WIN **=** 4**;**

private static final String DELIMITER **=** "|"**;**

private Color**[][]** board **=** **new** Color**[**COLUMNS**][**ROWS**];**

public Connect4**()** **{**

**for** **(**Color**[]** column **:** board**)** **{**

Arrays**.**fill**(**column**,** Color**.**EMPTY**);**

**}**

**}**

private void switchPlayer**()** **{**

**if** **(**Color**.**RED **==** currentPlayer**)** **{**

currentPlayer **=** Color**.**GREEN**;**

**}** **else** **{**

currentPlayer **=** Color**.**RED**;**

**}**

System**.**out**.**println**(**"Current turn: " **+** currentPlayer**);**

**}**

public void printBoard**()** **{**

**for** **(**int row **=** ROWS **-** 1**;** row **>=** 0**;** **--**row**)** **{**

StringJoiner stringJoiner **=** **new** StringJoiner**(**DELIMITER**,** DELIMITER**,** DELIMITER**);**

**for** **(**int col **=** 0**;** col **<** COLUMNS**;** **++**col**)** **{**

stringJoiner**.**add**(**board**[**col**][**row**].**toString**());**

**}**

System**.**out**.**println**(**stringJoiner**.**toString**());**

**}**

**}**

public void putDisc**(**int column**)** **{**

**if** **(**column **>** 0 **&&** column **<=** COLUMNS**)** **{**

int numOfDiscs **=** getNumberOfDiscsInColumn**(**column **-** 1**);**

**if** **(**numOfDiscs **<** ROWS**)** **{**

board**[**column **-** 1**][**numOfDiscs**]** **=** currentPlayer**;**

printBoard**();**

checkWinCondition**(**column **-** 1**,** numOfDiscs**);**

switchPlayer**();**

**}** **else** **{**

System**.**out**.**println**(**numOfDiscs**);**

System**.**out**.**println**(**"There's no room for a new disc in this column"**);**

printBoard**();**

**}**

**}** **else** **{**

System**.**out**.**println**(**"Column out of bounds"**);**

printBoard**();**

**}**

**}**

private int getNumberOfDiscsInColumn**(**int column**)** **{**

**if** **(**column **>=** 0 **&&** column **<** COLUMNS**)** **{**

int row**;**

**for** **(**row **=** 0**;** row **<** ROWS**;** row**++)** **{**

**if** **(**Color**.**EMPTY **==** board**[**column**][**row**])** **return** row**;**

**}**

**return** row**;**

**}**

**return** **-**1**;**

**}**

private void checkWinCondition**(**int col**,** int row**)** **{**

Pattern winPattern **=** Pattern**.**compile**(**".\*" **+** currentPlayer **+** "{" **+** DISCS\_FOR\_WIN **+** "}.\*"**);**

// Vertical check

StringJoiner stringJoiner **=** **new** StringJoiner**(**""**);**

**for** **(**int auxRow **=** 0**;** auxRow **<** ROWS**;** **++**auxRow**)** **{**

stringJoiner**.**add**(**board**[**col**][**auxRow**].**toString**());**

**}**

**if** **(**winPattern**.**matcher**(**stringJoiner**.**toString**()).**matches**())** **{**

winner **=** currentPlayer**;**

System**.**out**.**println**(**currentPlayer **+** " wins"**);**

**return;**

**}**

// Horizontal check

stringJoiner **=** **new** StringJoiner**(**""**);**

**for** **(**int column **=** 0**;** column **<** COLUMNS**;** **++**column**)** **{**

stringJoiner**.**add**(**board**[**column**][**row**].**toString**());**

**}**

**if** **(**winPattern**.**matcher**(**stringJoiner**.**toString**()).**matches**())** **{**

winner **=** currentPlayer**;**

System**.**out**.**println**(**currentPlayer **+** " wins"**);**

**return;**

**}**

// Diagonal checks

int startOffset **=** Math**.**min**(**col**,** row**);**

int column **=** col **-** startOffset**,** auxRow **=** row **-** startOffset**;**

stringJoiner **=** **new** StringJoiner**(**""**);**

**do** **{**

stringJoiner**.**add**(**board**[**column**++][**auxRow**++].**toString**());**

**}** **while** **(**column **<** COLUMNS **&&** auxRow **<** ROWS**);**

**if** **(**winPattern**.**matcher**(**stringJoiner**.**toString**()).**matches**())** **{**

winner **=** currentPlayer**;**

System**.**out**.**println**(**currentPlayer **+** " wins"**);**

**return;**

**}**

startOffset **=** Math**.**min**(**col**,** ROWS **-** 1 **-** row**);**

column **=** col **-** startOffset**;**

auxRow **=** row **+** startOffset**;**

stringJoiner **=** **new** StringJoiner**(**""**);**

**do** **{**

stringJoiner**.**add**(**board**[**column**++][**auxRow**--].**toString**());**

**}** **while** **(**column **<** COLUMNS **&&** auxRow **>=** 0**);**

**if** **(**winPattern**.**matcher**(**stringJoiner**.**toString**()).**matches**())** **{**

winner **=** currentPlayer**;**

System**.**out**.**println**(**currentPlayer **+** " wins"**);**

**}**

**}**

public boolean isFinished**()** **{**

**if** **(**winner **!=** **null)** **return** **true;**

int numOfDiscs **=** 0**;**

**for** **(**int col **=** 0**;** col **<** COLUMNS**;** **++**col**)** **{**

numOfDiscs **+=** getNumberOfDiscsInColumn**(**col**);**

**}**

**if** **(**numOfDiscs **>=** COLUMNS **\*** ROWS**)** **{**

System**.**out**.**println**(**"It's a draw"**);**

**return** **true;**

**}**

**return** **false;**

**}**

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

Connect4 game **=** **new** Connect4**();**

Scanner scanner **=** **new** Scanner**(**System**.**in**);**

**while** **(!**game**.**isFinished**())** **{**

System**.**out**.**println**(**"Where do the next disc should be placed?"**);**

game**.**putDisc**(**scanner**.**nextInt**());**

**}**

**}**

**}**

===

Requirement 1

The board is composed by seven horizontal and six vertical empty positions**.**

**Tests**

public class Connect4TDDSpec **{**

private Connect4TDD tested**;**

@Before

public void beforeEachTest**()** **{**

tested **=** **new** Connect4TDD**();**

**}**

@Test

public void whenTheGameIsStartedTheBoardIsEmpty**()** **{**

assertThat**(**tested**.**getNumberOfDiscs**(),** is**(**0**));**

**}**

**}**

**...**

**Code**

public class Connect4TDD **{**

public int getNumberOfDiscs**()** **{**

**return** 0**;**

**}**

**}**

**===**

Requirement 2

This is the implementation of the second requirement**.**

**Tests**

Players introduce discs on the top of the columns**.** An introduced disc drops down the board **if** the column is empty**.** Future discs introduced in the same column will stack over the previous ones**.** We can split **this** requirement into the following tests**:**

• When a disc is inserted into an empty column**,** its position is 0

• When a second disc is inserted into the same column**,** its position is 1

• When a disc is inserted into the board**,** the total number of discs increases

• When a disc is put outside the boundaries**,** a Runtime Exception is thrown

• When a disc is inserted in to a column and there's no room available for it, then a Runtime Exception is thrown

Also**,** these other tests are derived from the first requirement**.** They are related to the board limits or board behaviour**.**

@Test

public void

whenDiscOutsideBoardThenRuntimeException**()** **{**

int column **=** **-**1**;**

exception**.**expect**(**RuntimeException**.**class**);**

exception**.**expectMessage**(**"Invalid column " **+** column**);**

tested**.**putDiscInColumn**(**column**);**

**}**

@Test

public void

whenFirstDiscInsertedInColumnThenPositionIsZero**()** **{**

int column **=** 1**;**

assertThat**(**tested**.**putDiscInColumn**(**column**),** is**(**0**));**

**}**

@Test

public void

whenSecondDiscInsertedInColumnThenPositionIsOne**()** **{**

int column **=** 1**;**

tested**.**putDiscInColumn**(**column**);**

assertThat**(**tested**.**putDiscInColumn**(**column**),** is**(**1**));**

**}**

@Test

public void whenDiscInsertedThenNumberOfDiscsIncreases**()** **{**

int column **=** 1**;**

tested**.**putDiscInColumn**(**column**);**

assertThat**(**tested**.**getNumberOfDiscs**(),** is**(**1**));**

**}**

@Test

public void whenNoMoreRoomInColumnThenRuntimeException**()** **{**

int column **=** 1**;**

int maxDiscsInColumn **=** 6**;** // the number of rows

**for** **(**int times **=** 0**;** times **<** maxDiscsInColumn**;** **++**times**)** **{**

tested**.**putDiscInColumn**(**column**);**

**}**

exception**.**expect**(**RuntimeException**.**class**);**

exception**.**expectMessage**(**"No more room in column " **+** column**);**

tested**.**putDiscInColumn**(**column**);**

**}**

**...**

**Code**

private static final int ROWS **=** 6**;**

private static final int COLUMNS **=** 7**;**

private static final String EMPTY **=** " "**;**

private String**[][]** board **=** **new** String**[**ROWS**][**COLUMNS**];**

public Connect4TDD**()** **{**

**for** **(**String**[]** row **:** board**)**

Arrays**.**fill**(**row**,** EMPTY**);**

**}**

public int getNumberOfDiscs**()** **{**

**return** IntStream**.**range**(**0**,** COLUMNS**).**map**(this::**getNumberOfDiscsInColumn**).**sum**();**

**}**

private int getNumberOfDiscsInColumn**(**int column**)** **{**

**return** **(**int**)** IntStream**.**range**(**0**,** ROWS**).**filter**(**row **->** **!**EMPTY**.**equals**(**board**[**row**][**column**])).**count**();**

**}**

public int putDiscInColumn**(**int column**)** **{**

checkColumn**(**column**);**

int row **=** getNumberOfDiscsInColumn**(**column**);**

checkPositionToInsert**(**row**,** column**);**

board**[**row**][**column**]** **=** "X"**;**

**return** row**;**

**}**

private void checkColumn**(**int column**)** **{**

**if** **(**column **<** 0 **||** column **>=** COLUMNS**)**

**throw** **new** RuntimeException**(**"Invalid column " **+** column**);**

**}**

private void checkPositionToInsert**(**int row**,** int column**)** **{**

**if** **(**row **==** ROWS**)**

**throw** **new** RuntimeException**(**"No more room in column " **+** column**);**

**}**

**===**

Requirement 3

The third requirement specifies the game logic**.**

**Tests**

It is a two**-**person game**,** so there is one colour **for** each player**.** One player uses red **(**'R'**)** and the other one uses green **(**'G'**).** Players alternate turns**,** inserting one disc every time**.**

@Test

public void

whenFirstPlayerPlaysThenDiscColorIsRed**()** **{**

assertThat**(**tested**.**getCurrentPlayer**(),** is**(**"R"**));**

**}**

@Test

public void whenSecondPlayerPlaysThenDiscColorIsRed**()** **{**

int column **=** 1**;**

tested**.**putDiscInColumn**(**column**);**

assertThat**(**tested**.**getCurrentPlayer**(),** is**(**"G"**));**

**}**

**...**

**Code**

A couple of methods need to be created to cover **this** functionality**.**

The switchPlayer method is called before returning the row in the putDiscInColumn method**:**

private static final String RED **=** "R"**;**

private static final String GREEN **=** "G"**;**

private String currentPlayer **=** RED**;**

public Connect4TDD**()** **{**

**for** **(**String**[]** row **:** board**)**

Arrays**.**fill**(**row**,** EMPTY**);**

**}**

public String getCurrentPlayer**()** **{**

**return** currentPlayer**;**

**}**

private void switchPlayer**()** **{**

**if** **(**RED**.**equals**(**currentPlayer**))**

currentPlayer **=** GREEN**;**

**else** currentPlayer **=** RED**;**

**}**

public int putDiscInColumn**(**int column**)** **{**

**...**

switchPlayer**();**

**return** row**;**

**}**

**===**

Requirement 4

Next**,** we should let the player know the status of the game**.**

We want feedback when either an event or an error occurs within the game**.** The output shows the status of the board on every move**.**

**Tests**

As we are throwing exceptions when an error occurs**,** **this** is already covered**,** so we only need to implement these two tests**.** Furthermore**,** **for** the sake of testability**,** we need to introduce a parameter within the constructor**.** By introducing **this** parameter**,** the output becomes easier to test**:**

private OutputStream output**;**

@Before

public void beforeEachTest**()** **{**

output **=** **new** ByteArrayOutputStream**();**

tested **=** **new** Connect4TDD**(**

**new** PrintStream**(**output**));**

**}**

@Test

public void

whenAskedForCurrentPlayerTheOutputNotice**()** **{**

tested**.**getCurrentPlayer**();**

assertThat**(**output**.**toString**(),**

containsString**(**"Player R turn"**));**

**}**

@Test

public void

whenADiscIsIntroducedTheBoardIsPrinted**()** **{**

int column **=** 1**;**

tested**.**putDiscInColumn**(**column**);**

assertThat**(**output**.**toString**(),**

containsString**(**"| |R| | | | | |"**));**

**}**

**...**

**Code**

One possible implementation is to pass the above tests**.** As you can see**,** the class constructor now has one parameter**.** This parameter is used in several methods to print the event or action description**:**

private static final String DELIMITER **=** "|"**;**

public Connect4TDD**(**PrintStream out**)** **{**

outputChannel **=** out**;**

**for** **(**String**[]** row **:** board**)**

Arrays**.**fill**(**row**,** EMPTY**);**

**}**

public String getCurrentPlayer**()** **{**

outputChannel**.**printf**(**"Player %s turn%n"**,** currentPlayer**);**

**return** currentPlayer**;**

**}**

private void printBoard**()** **{**

**for** **(**int row **=** ROWS **-** 1**;** row **>=** 0**;** row**--)** **{**

StringJoiner stringJoiner **=** **new** StringJoiner**(**DELIMITER**,** DELIMITER**,** DELIMITER**);**

Stream**.**of**(**board**[**row**]).**forEachOrdered**(**stringJoiner**::**add**);**

outputChannel**.**println**(**stringJoiner**.**toString**());**

**}**

**}**

public int putDiscInColumn**(**int column**)** **{**

**...**

printBoard**();**

switchPlayer**();**

**return** row**;**

**}**

**===**

Requirement 5

This requirement tells the system whether the game is finished**.**

When no more discs can be inserted**,** the game finishes and it is considered a draw**.**

**Tests**

There are two conditions to test**.** The first condition is that **new** games must be unfinished**;** the second condition is that full board games must be finished**:**

@Test

public void whenTheGameStartsItIsNotFinished**()** **{**

assertFalse**(**"The game must not be finished"**,** tested**.**isFinished**());**

**}**

@Test

public void whenNoDiscCanBeIntroducedTheGamesIsFinished**()** **{**

**for** **(**int row **=** 0**;** row **<** 6**;** row**++)**

**for** **(**int column **=** 0**;** column **<** 7**;** column**++)**

tested**.**putDiscInColumn**(**column**);**

assertTrue**(**"The game must be finished"**,** tested**.**isFinished**());**

**}**

**Code**

An easy and simple solution to these two tests is as follows**:**

public boolean isFinished**()** **{**

**return** getNumberOfDiscs**()** **==** ROWS **\*** COLUMNS**;**

**}**

**===**

Requirement 6

This is the first win condition requirement**.**

If a player inserts a disc and connects more than three discs of his color in a straight vertical line**,** then that player wins**.**

Tests

In fact**,** **this** requires one single check**.** If the current inserted disc connects other three discs in a vertical line**,** the current player wins the game**:**

**Tests**

@Test

public void when4VerticalDiscsAreConnectedThenPlayerWins**()** **{**

**for** **(**int row **=** 0**;** row **<** 3**;** row**++)** **{**

tested**.**putDiscInColumn**(**1**);** // R

tested**.**putDiscInColumn**(**2**);** // G

**}**

assertThat**(**tested**.**getWinner**(),** isEmptyString**());**

tested**.**putDiscInColumn**(**1**);** // R

assertThat**(**tested**.**getWinner**(),** is**(**"R"**));**

**}**

**Code**

There are a couple of changes to the putDiscInColumn method**.** Also**,** a **new** method called checkWinner has been created**:**

private static final int DISCS\_TO\_WIN **=** 4**;**

private String winner **=** ""**;**

private void checkWinner**(**int row**,** int column**)** **{**

**if** **(**winner**.**isEmpty**())** **{**

String colour **=** board**[**row**][**column**];**

Pattern winPattern **=** Pattern**.**compile**(**".\*" **+** colour **+** "{" **+** DISCS\_TO\_WIN **+** "}.\*"**);**

String vertical **=** IntStream**.**range**(**0**,** ROWS**).**mapToObj**(**r **->** board**[**r**][**column**]).**reduce**(**String**::**concat**).**get**();**

**if** **(**winPattern**.**matcher**(**vertical**).**matches**())**

winner **=** colour**;**

**}**

**}**

**Development practices**

Practices listed in **this** section are focused on the best way to write tests**.**

1**-** Write the simplest code to pass the test**:** Benefits **->** It ensures cleaner and clearer design**;** avoids unnecessary features**.**

The idea is that the simpler the implementation**,** the better and easier it is to maintain the product**.** The idea adheres to the keep it simple stupid **(**KISS**)** principle**.** This states that most systems work best **if** they are kept simple rather than made complex**;** therefore**,** simplicity should be a key goal in design**,** and unnecessary complexity should be avoided**.**

2**-** Write assertions first**,** act later**:** Benefits **->** This clarifies the purpose of the requirements and tests early**.**

Once the assertion is written**,** the purpose of the test is clear and the developer can concentrate on the code that will accomplish that assertion and**,** later on**,** on the actual implementation**.**

3**-** Minimize assertions in each test**:** Benefits **->** This avoids assertion roulette**;** allows execution of more asserts**.**

If multiple assertions are used within one test method**,** it might be hard to tell which of them caused a test failure**.** This is especially common when tests are executed as part of the continuous integration process**.** If the problem cannot be reproduced on a developer's machine (as may be the case if the problem is caused by environmental issues), fixing the problem may be difficult and time consuming.

@Test

public final void whenOneNumberIsUsedThenReturnValueIsThatSameNumber**()** **{**

Assert**.**assertEquals**(**3**,** StringCalculator**.**add**(**"3"**));**

**}**

@Test

public final void whenTwoNumbersAreUsedThenReturnValueIsTheirSum**()** **{**

Assert**.**assertEquals**(**3**+**6**,** StringCalculator**.**add**(**"3,6"**));**

**}**

The preceding code contains two specifications that clearly define what the objective of those tests is**.** By reading the method names and looking at the **assert,** there should be clarity on what is being tested**.** Consider the following **for** example**:**

@Test

public final void whenNegativeNumbersAreUsedThenRuntimeExceptionIsThrown**()** **{**

RuntimeException exception **=** **null;**

**try** **{**

StringCalculator**.**add**(**"3,-6,15,-18,46,33"**);**

**}** **catch** **(**RuntimeException e**)** **{**

exception **=** e**;**

**}**

Assert**.**assertNotNull**(**"Exception was not thrown"**,** exception**);**

Assert**.**assertEquals**(**"Negatives not allowed: [-6, -18]"**,**

exception**.**getMessage**());**

**}**

This specification has more than one **assert,** but they are testing the same logical unit of functionality**.** The first **assert** is confirming that the exception exists**,** and the second that its message is correct**.** When multiple asserts are used in one test method**,** they should all contain messages that explain the failure**.** This way debugging the failed **assert** is easier**.** In the **case** of one **assert** per test method**,** messages are welcome**,** but not necessary since it should be clear from the method name what the objective of the test is**.**

@Test

public final void whenAddIsUsedThenItWorks**()** **{**

Assert**.**assertEquals**(**0**,** StringCalculator**.**add**(**""**));**

Assert**.**assertEquals**(**3**,** StringCalculator**.**add**(**"3"**));**

Assert**.**assertEquals**(**3**+**6**,** StringCalculator**.**add**(**"3,6"**));**

Assert**.**assertEquals**(**3**+**6**+**15**+**18**+**46**+**33**,**

StringCalculator**.**add**(**"3,6,15,18,46,33"**));**

Assert**.**assertEquals**(**3**+**6**+**15**,** StringCalculator**.**add**(**"3,6n15"**));**

Assert**.**assertEquals**(**3**+**6**+**15**,**

StringCalculator**.**add**(**"//;n3;6;15"**));**

Assert**.**assertEquals**(**3**+**1000**+**6**,**

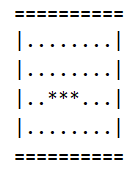
StringCalculator**.**add**(**"3,1000,1001,6,1234"**));**

**}**

This test has many asserts**.** It is unclear what the functionality is**,** and **if** one of them fails**,** it is unknown whether the rest would work or not**.** It might be hard to understand the failure when **this** test is executed through some of the CI tools**.**

**GAME OF LIFE**

**1- TO PRINT A TABLE**

**public class AppGame**{  
  
 **int alto**;  
 **int ancho**;  
 **int**[][] **tablero**;  
  
 **public AppGame**(**int** alto, **int** ancho){  
 **this**.**alto** = alto;  
 **this**.**ancho** = ancho;  
 **this**.**tablero** = **new int**[alto][ancho];  
 }  
  
 **public void imprimeTablero**(){  
 System.***out***.println(**"=========="**);  
 **for**(**int** y = 0; y < **alto** ; y++){  
 String line = **"|"**;  
 **for**(**int** x = 0; x < **ancho** ; x++){  
 **if**(**this**.**tablero**[y][x] == 0){  
 line += **"."**;  
 } **else** {  
 line += **"\*"**;  
 }  
 }  
 line += **"|"**;  
 System.***out***.println(line);  
 }  
 System.***out***.println(**"==========\n"**);  
 }  
  
 **public void setAlive**(**int** y, **int** x){  
 **this**.**tablero**[y][x] = 1;  
 }  
  
 **public void setDead**(**int** y, **int** x){  
 **this**.**tablero**[y][x] = 0;  
 }  
  
 **public static void main**(String[] args){  
 **AppGame appGame = new AppGame(4, 8);  
 appGame.setAlive(2, 2);  
 appGame.setAlive(2, 3);  
 appGame.setAlive(2, 4);  
  
 appGame.imprimeTablero();** }  
}

2- SABER CUANTOS VECINOS TENGO VIVOS

**public class** AppGame{  
  
**…**   
 **public int** getVecinosVivos(**int** y, **int** x){  
 **int** count = 0;  
 count += **this**.**tablero**[y-1][x-1];  
 count += **this**.**tablero**[y][x-1];  
 count += **this**.**tablero**[y+1][x-1];  
  
 count += **this**.**tablero**[y-1][x];  
 count += **this**.**tablero**[y+1][x];  
  
 count += **this**.**tablero**[y-1][x+1];  
 count += **this**.**tablero**[y][x+1];  
 count += **this**.**tablero**[y+1][x+1];  
  
 **return** count;  
 }  
  
…

**public static void** main(String[] args){  
 AppGame appGame = **new** AppGame(4, 8);  
 appGame.setAlive(2, 2);  
 appGame.setAlive(2, 3);  
 appGame.setAlive(2, 4);  
  
 appGame.imprimeTablero();  
  
 System.***out***.println(appGame.getVecinosVivos(2, 3));  
  
 }  
}

**JUEGO FINAL CON 8 COLUMNAS y 6 FILAS (6 de ALTO y 8 de ANCHO)**

**TDD**

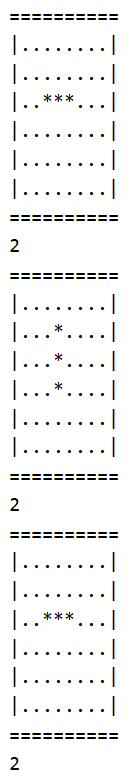
**import** org.junit.Before;  
**import** org.junit.Test;  
**import static** org.junit.Assert.*assertEquals*;

**public class AppGameTest**{  
  
 AppGame **appGame**;  
  
 @Before  
 **public void** setUp(){  
 **appGame** = **new** AppGame(6, 8);  
 }  
  
 @Test  
 **public void** testTieneDosVecinosVivos(){  
 **appGame**.setAlive(2, 2);  
 **appGame**.setAlive(2, 3);  
 **appGame**.setAlive(2, 4);  
 *assertEquals*(2,**appGame**.getVecinosVivos(2, 3));  
 }  
}

…

**public class** AppGame{  
  
 **int alto**;  
 **int ancho**;  
 **int**[][] **tablero**;  
  
 **public** AppGame(**int** alto, **int** ancho){  
 **this**.**alto** = alto;  
 **this**.**ancho** = ancho;  
 **this**.**tablero** = **new int**[alto][ancho];  
 }  
  
 **public void** imprimeTablero(){  
 System.***out***.println(**"=========="**);  
 **for**(**int** y = 0; y < **alto**; y++){  
 String line = **"|"**;  
 **for**(**int** x = 0; x < **ancho**; x++){  
 **if**(**this**.**tablero**[y][x] == 0){ *// RECORRO VERTICAL!!!* line += **"."**;  
 } **else**{  
 line += **"\*"**;  
 }  
 }  
 line += **"|"**;  
 System.***out***.println(line);  
 }  
 System.***out***.println(**"=========="**);  
 }  
  
 **public void** setAlive(**int** y, **int** x){  
 **this**.**tablero**[y][x] = 1;  
 }  
  
 **public void** setDead(**int** y, **int** x){  
 **this**.**tablero**[y][x] = 0;  
 }  
  
 **public int** getVecinosVivos(**int** y, **int** x){  
 **int** count = 0;  
  
 count += esFrontera(y - 1, x - 1);  
 count += esFrontera(y, x - 1);  
 count += esFrontera(y + 1, x - 1);  
  
 count += esFrontera(y - 1, x);  
 count += esFrontera(y + 1, x);  
  
 count += esFrontera(y - 1, x + 1);  
 count += esFrontera(y, x + 1);  
 count += esFrontera(y + 1, x + 1);  
  
 **return** count;  
 }  
  
 **public int** esFrontera(**int** y, **int** x){  
 **if**(y < 0 || y >= **alto**){  
 **return** 0;  
 }  
  
 **if**(x < 0 || x >= **ancho**){  
 **return** 0;  
 }  
 **return this**.**tablero**[y][x];  
 }  
  
 **public void** siguientePaso(){  
 **int**[][] nuevoTablero = **new int**[**alto**][**ancho**];  
  
 **for**(**int** y = 0; y < **alto**; y++){  
 **for**(**int** x = 0; x < **ancho**; x++){  
 **int** vecinosVivos = getVecinosVivos(y, x);  
  
 **if**(esFrontera(y, x) == 1){  
 **if**(vecinosVivos < 2){  
 nuevoTablero[y][x] = 0;  
 } **else if**(vecinosVivos == 2 || vecinosVivos == 3){  
 nuevoTablero[y][x] = 1;  
 } **else if**(vecinosVivos > 3){  
 nuevoTablero[y][x] = 0;  
 }  
 } **else**{  
 **if**(vecinosVivos == 3){  
 nuevoTablero[y][x] = 1;  
 }  
 }  
 }  
 }  
 **this**.**tablero** = nuevoTablero;  
 }  
  
 **public static void** main(String[] args){  
 AppGame appGame = **new** AppGame(6, 8);  
  
 appGame.setAlive(2, 2);  
 appGame.setAlive(2, 3);  
 appGame.setAlive(2, 4);  
  
 appGame.imprimeTablero();  
  
 System.***out***.println(appGame.getVecinosVivos(2, 3));  
  
 appGame.siguientePaso();  
  
 appGame.imprimeTablero();  
  
 System.***out***.println(appGame.getVecinosVivos(2, 3));  
  
 appGame.siguientePaso();  
  
 appGame.imprimeTablero();  
  
 System.***out***.println(appGame.getVecinosVivos(2, 3));  
  
 }  
}

…



**OJO CON ESTO**

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

AppGame appGame **=** **new** AppGame**(**6**,** 8**);**

appGame**.**setAlive**(**2**,** 2**);**

appGame**.**setAlive**(**2**,** 3**);**

appGame**.**setAlive**(**2**,** 4**);**

appGame**.**setAlive**(**3**,** 2**);**

appGame**.**setAlive**(**3**,** 3**);**

appGame**.**imprimeTablero**();**

System**.**out**.**println**(**appGame**.**getVecinosVivos**(**2**,** 3**));**

appGame**.**siguientePaso**();**

appGame**.**imprimeTablero**();**

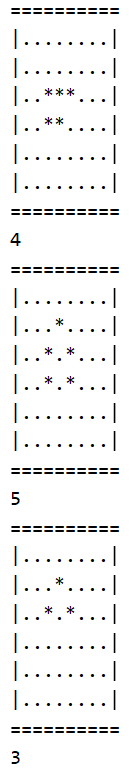
System**.**out**.**println**(**appGame**.**getVecinosVivos**(**2**,** 3**));**

appGame**.**siguientePaso**();**

appGame**.**imprimeTablero**();**

System**.**out**.**println**(**appGame**.**getVecinosVivos**(**2**,** 3**));**

**}**

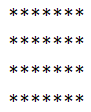


**TEST**

**@Test  
public void testMutacion**(){  
 **appGame**.setAlive(2, 2);  
 **appGame**.setAlive(2, 3);  
 **appGame**.setAlive(2, 4);  
 **appGame**.setAlive(3, 2);  
 **appGame**.setAlive(3, 3);  
  
 *assertEquals*(4, **appGame**.getVecinosVivos(2, 3));  
 **appGame**.siguientePaso();  
 *assertEquals*(5, **appGame**.getVecinosVivos(2, 3));  
 **appGame**.siguientePaso();  
 *assertEquals*(3, **appGame**.getVecinosVivos(2, 3));  
}

**COMO SE ITERA UN ARRAY Y SE PINTA!!!**

**public class App**{  
  
 **public static void main**(String[] args){  
  
 String[][] **array** = **new** String[7][4]; // **el array 2d comienza con ‘x’ e ‘y’**  
  
 **for**(**int** i = 0; i < 4; i++){  
 **for**(**int** j = 0; j < 7; j++){  
 **if**(array[j][i] == **null**){  
 array[j][i] = **"\*"**;  
 System.***out***.print(array[j][i]);  
 }  
 }  
 System.***out***.println(**""**);  
 }  
 }  
}

**OJO!!! EL Arrays.toString es solo para unidimensionales primitivos**

**public class App** {  
 **public static void main**(String[] args){  
 **int**[] array = **new int**[]{1, 2, 3, 4, 5, 6};  
 System.***out***.print(Arrays.*toString*(array));  
 }  
}

**[1, 2, 3, 4, 5, 6]**

**FizzBuzz al reves!!!**

**public class BizzBuzz**{  
  
 **public static void main**(String[] args){  
 ***itera*(100);** }  
  
 **public static void itera**(**int** integer){  
 **for**(**int** j = integer; j > 0; j--){  
 **if(j % 3 == 0 && j % 5 == 0){** System.***out***.println(**"FizzBuzz"**);  
 }  
 **else if(j % 3 == 0){** System.***out***.println(**"Fizz"**);  
 }  
 **else if(j % 5 == 0){** System.***out***.println(**"Buzz"**);  
 **} else {** System.***out***.println(j);  
 }  
 }  
 }  
}

…

Buzz

Fizz

98

97

Fizz

Buzz

94

**TWO NUMBERS**

public class **TwoNumbers{**

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

int**[]** array **=** **new** int**[]** **{**2**,**3**,**7**,**4**,**8**};**

int target **=** 7**;**

int**[]** mapping **=** **mapping(**array**,** target**);**

System**.**out**.**println**(**Arrays**.**toString**(**mapping**));**

**}**

private static int**[]** **mapping(**int**[]** numbers**,** int target**){**

Map**<**Integer**,** Integer**>** map **=** **new** HashMap**<>();**

**for(**int i **=** 0**;** i **<** numbers**.**length**;** i**++){**

int eureka **=** target **-** numbers**[**i**];**

**if(**map**.**containsKey**(**eureka**)){**

**return** **new** int**[]** **{**i**,** map**.**get**(**eureka**)};**

**}**

**map.put(numbers[i],i);**

**}**

**return** **new** int**[]** **{-**1**,-**1**};**

**}**

**}**

…

[3, 1]

**STRING REVERSE**

public class **StringReverse** **{**

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

readBack**(**"Hello World!"**);**

System**.**out**.**println**();**

withStrBuilder**(**"Hello World!"**);**

**}**

private static void **readBack(**String string**){**

char**[]** chars **=** string**.toCharArray();**

**for(**int i **=** chars**.**length **-**1**;** i **>=** 0 **;** i**--)** **{**

System**.**out**.**print**(**chars**[**i**]);**

**}**

**}**

public static void **withStrBuilder(**String builder**){**

StringBuilder stringBuilder **=** **new** StringBuilder**(builder);**

StringBuilder **reverse** **=** stringBuilder**.reverse();**

System**.**out**.**print**(reverse.toString());**

**}**

**}**

…

!dlroW olleH

!dlroW olleH

**STRING REVERSE**

public class **RomanWayOne** **{**

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

System**.**out**.**println**(*intToRoman*(**46**));**

**}**

public static String **intToRoman(**int figura**)** **{**

StringBuilder sb **=** **new** StringBuilder**();**

int veces **=** 0**;**

int**[]** **numeros** **=** **new** int**[]** **{** 50**,** 40**,** 10**,** 9**,** 5**,** 4**,** 1 **}** **;**

String**[]** **romanos** **=** **new** String**[]** **{** "L"**,** "XL"**,** "X"**,** "IX"**,** "V"**,** "IV"**,** "I" **};**

**for** **(**int i **=** 0**;** i **<** **numeros.length** **;** i**++)** **{**

veces **=** **figura** **/** **numeros[**i**];** // 1a 46/50 = 0 --- 2a 46/40 = 1

figura **=** **figura** **%** **numeros[**i**];** // 1a 46%50 = 46 --- 2a 46%40 = 6

**while** **(**veces **>** 0**)** **{**

sb**.**append**(romanos[**i**]);**

veces**--;**

**}**

**}**

**return** **sb.toString();**

**}**

**}**

…

**XLVI**

===

public class **RomanWayTwo** **{**

public static String **intToRoman(**int num**)** **{**

String**[]** **teens** **=** **{**""**,** "X"**,** "XX"**,** "XXX"**,** "XL"**,** "L"**,** "LX"**,** "LXX"**,** "LXXX"**,** "XC"**};**

String**[]** **units** **=** **{**""**,** "I"**,** "II"**,** "III"**,** "IV"**,** "V"**,** "VI"**,** "VII"**,** "VIII"**,** "IX"**,** "X"**};**

System**.**out**.**println**((**62 **%** 100**)** **/** 10**);**

System**.**out**.**println**(**62 **%** 10**);**

**return** teens**[** **(**num **%** 100**)** **/** 10 **]** **+** // esto es index 6

units**[**num **%** 10**];** // esto es index 2

**}**

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

System**.**out**.**println**(**intToRoman**(**62**));**

**}**

**}**

**…**

**6**

**2**

**LXII**

**COMO REFACTORIZAR CON LAMBDAS (**[**Mastering Object-oriented Programming in Java**](https://app.pluralsight.com/courses/154a2b14-1de5-471f-8ac6-ff640943b02f/table-of-contents) **--- PLURALSIGHT COURSE 13-04-20)**

**1) Tengo un INTERFACE**

**public interface Painter** {  
 **boolean** isAvailable();  
 Duration estimateTimeToPaint(**double** sqMeters);  
 Money estimateCompensation(**double** sqMeters);  
 String getName();  
}

**2) Una class Money**

**import** java.math.BigDecimal;  
**import** java.math.RoundingMode;  
  
**public class Money** **implements Comparable<Money>** {  
 **private** BigDecimal **amount**;  
  
 **private** BigDecimal getAmount() { **return this**.**amount**; }  
  
 **public static** Money *ZERO* =  
 **new** Money(BigDecimal.***ZERO***.setScale(2, RoundingMode.***HALF\_UP***));  
  
 **public** Money(BigDecimal amount) {  
 **this**.**amount** = amount.setScale(2, RoundingMode.***HALF\_UP***);  
 }  
  
 **public** Money scale(**long** multiply, **long** divide) {  
 **return this**.scale(**new** BigDecimal(multiply), **new** BigDecimal(divide));  
 }  
  
 **public** Money scale(**double** factor) {  
 BigDecimal newAmount = **this**.getAmount().multiply(**new** BigDecimal(factor));  
 **return new** Money(newAmount);  
 }  
  
 **public** Money add(Money other) {  
 **return new** Money(**this**.getAmount().add(other.getAmount()));  
 }  
  
 **private** Money scale(BigDecimal multiply, BigDecimal divide) {  
 **return new** Money(**this**.getAmount().multiply(multiply).divide(divide, 2, RoundingMode.***HALF\_UP***));  
 }  
  
 **public int** compareTo(Money other) {  
 **return this**.getAmount().compareTo(other.**amount**);  
 }  
  
 **public** String toString() {  
 **return "$"** + **this**.getAmount();  
 }  
}

**3) Como lo implemento para saber cual es el pintor mas barato:**

**public class Demo** {  
 **private static** Painter findCheapest1(**double** sqMeters, List<Painter> painters) {  
 Money lowestCost = Money.*ZERO*;  
 Painter winner = **null**;  
  
 **for** (Painter candidate: painters) {  
 **if** (candidate.isAvailable()) {  
 Money cost = candidate.estimateCompensation(sqMeters);  
 **if** (winner == **null** || cost.compareTo(lowestCost) <= 0) {  
 winner = candidate;  
 lowestCost = cost;  
 }  
 }  
 }  
 **return** winner;  
 }  
  
 **public void run**() {  
 }  
}

4) UN RUN IRRELEVANTE

**public class App** {  
 **public static void main**(String[] args) {  
 **new Demo**().**run**();

**Demo demo = new Demo();  
demo.run();**

}  
}

…

**MIRA COMO SE REFACTORIZA CON LAMBDAS**

**import** java.util.Comparator;  
**import** java.util.List;  
  
**public class** Demo {  
 **private static** Painter findCheapest1(**double** sqMeters, List<Painter> painters) {  
 **return** painters.stream()  
 .filter(Painter::isAvailable)  
 .min(Comparator.*comparing*(painter -> painter.estimateCompensation(sqMeters)))  
 .get();  
 }  
  
 **private static** Money getTotalCost(**double** sqMeters, List<Painter> painters) {  
 **return** painters.stream()  
 .filter(Painter::isAvailable)  
 .map(painter -> painter.estimateCompensation(sqMeters))  
 .reduce(Money::add)  
 .orElse(Money.*ZERO*);  
 }  
  
 **public void run**() {  
 }  
}

…

**REFACTORIZO EL METODO CON OPTIONAL**

import java.util.Optional;

public class Demo {

**private static** Optional<Painter> findCheapest1(**double** sqMeters, List<Painter> painters) {  
 **return** painters.stream()  
 .filter(Painter::isAvailable)  
 .min(Comparator.*comparing*(painter -> painter.estimateCompensation(sqMeters)));  
 }

…

}

**REALIZO UNA MEJOR REFACTORIZACION**

**import** java.util.Comparator;  
**import** java.util.List;  
**import** java.util.Optional;  
  
**public class** Demo {  
 **private static** Optional<Painter> findCheapest1(**double** sqMeters, List<Painter> painters) {  
 **return** painters.stream()  
 .filter(Painter::isAvailable)  
 .min(Comparator.*comparing*(painter -> painter.estimateCompensation(sqMeters)));  
 }  
  
 **private static** Optional<Painter> findCheapest2(**double** sqMeters, List<Painter> painters) {  
 **return** Painter.*stream*(painters).available().cheapest(sqMeters);  
 }  
  
 **private static** Money getTotalCost(**double** sqMeters, List<Painter> painters) {  
 **return** painters.stream()  
 .filter(Painter::isAvailable)  
 .map(painter -> painter.estimateCompensation(sqMeters))  
 .reduce(Money::add)  
 .orElse(Money.*ZERO*);  
 }  
  
 **public void** run() {  
 }  
}

…

**import** java.time.Duration;  
**import** java.util.List;  
  
**public interface** Painter {  
 **boolean** isAvailable();  
 Duration estimateTimeToPaint(**double** sqMeters);  
 Money estimateCompensation(**double** sqMeters);  
 String getName();  
  
 **static** PaintersStream stream(List<Painter> painters) {  
 **return new** PaintersStream(painters.stream());  
 }  
}

…

**import** java.util.Comparator;  
**import** java.util.Optional;  
**import** java.util.stream.Stream;  
  
**public class PaintersStream** **implements ForwardingStream<Painter>** {  
 **private** Stream<Painter> **stream**;  
  
 **public PaintersStream**(Stream<Painter> stream) {  
 **this**.**stream** = stream;  
 }  
  
 **@Override** // es el unico metodo, es un interface functional!!! Los demas son DEFAULT!!!  
 **public** Stream<Painter> **getStream**() { **return this**.**stream**; }  
  
 **public** PaintersStream **available**() {  
 **return new** PaintersStream(**this**.getStream().filter(Painter::isAvailable));  
 }  
  
 **public** Optional<Painter> **cheapest**(**double** sqMeters) {  
 **return this**.**getStream**()  
 .**min**(Comparator.*comparing*(painter -> painter.estimateCompensation(sqMeters)));  
 }

// Lo anado en la siguiente MEJORA

public Optional<Painter> **fastest**(double sqMeters) {

return **this.getStream()**

**.min(Comparator.comparing(painter -> painter.estimateTimeToPaint(sqMeters)));**

}

}

…

**import** java.util.Comparator;  
**import** java.util.Iterator;  
**import** java.util.Optional;  
**import** java.util.Spliterator;  
**import** java.util.function.\*;  
**import** java.util.stream.\*;  
  
**public interface ForwardingStream**<T> **extends Stream**<T> {  
 **Stream<T> getStream();** **default** Stream<T> filter(Predicate<? **super** T> predicate) { **return this**.getStream().filter(predicate); }  
 **default** <R> Stream<R> map(Function<? **super** T, ? **extends** R> mapper) { **return this**.getStream().map(mapper); }  
 **default …**

…

}