1. # Instalación de IntelliJ IDEA, creación del Proyecto con Maven y Tests Unitarios

Vamos a descargar el editor IntelliJ IDEA y crear un proyecto en Maven. Para indicarle a Maven que usaremos Java 8 debemos añadir las siguientes líneas de código:

```
<properties>
    <maven.compiler.source>1.8maven.compiler.source>
    <maven.compiler.target>1.8maven.compiler.target>

properties>
```

2. # Creación de test unitario: lanzar una excepción para alertar sobre un error

Vamos a utilizar una excepción con la función `throw new RuntimeException("Error")` en lugar de la función `System.out.println("Error")` para identificar más fácil los errores. Ahora, los mensajes tendrán un color diferente y pueden mostrarnos un poco más de información sobre los errores: ubicación, el resultado esperado, mensajes personalizados, entre otros.

3. Test unitario con JUnit

Vamos a añadir Junit a nuestro proyecto copiando las siguientes líneas de código:

```
<dependencies>
  <dependency>
    <groupId>junitgroupId>
    <artifactId>junitartifactId>
    <version>4.12version>
     <scope>testscope>
  dependency>
dependencies>
```

La función `assertEquals` de JUnit se encarga de comprobar que dos valores sean iguales, en este caso comprobar si nuestra función `repeat` retorna el valor esperado.

Debemos indicarle a JUnit mediante `@Test` que función va a realizar una prueba.

4. Organización de tests con JUnit

La forma correcta de separar nuestras pruebas es realizar cada una en su propia función, además, el nombre de la función debe describir que estamos probando.

Para indicarle a JUnit que esperamos una excepción lo debemos hacer de la siguiente forma:

```
@Test(expected = IllegalArgumentException.class)
```

5. Test con Mockito para simular un dado

Mockito nos va a servir para simular clases mientras probamos, para añadirlo a nuestro proyecto debemos copiar las siguientes líneas de código:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>2.23.4</version>
  <scope>test</scope>
</dependency>
```

Para instanciar un mock debemos utilizar la función `Mockito.mock()` e indicarle como parámetro la clase que va a simular.

Las funciones `assertFalse` y `assertTrue` tal como su nombre lo indican, sirven para comprobar si un valor es igual a `false` o `true` respectivamente.

6. Análisis de los tests y mejoras

Nuestros test siguen un mismo proceso:

1. Se preparan los objetos que vamos a probar.
2. Llamamos al método que estamos probando.
3. Comprobamos los resultados.

Podemos reducir la cantidad de código moviendo las partes comunes de preparación a una función que se ejecute antes de cada prueba.

Con `@Before` le indicamos a JUnit la función que debe ejecutar antes de cada prueba.

> la anotación @Before se puede utilizar en métodos estáticos para ejecutar instrucciones al principio de todos los tests de esa clase. Esto puede usarse para operaciones costosas como conexiones a bases de datos que no quieras ejecutar antes de cada uno de los tests (que en una aplicación escalable pueden ser muchos).

Las métodos test regularmente cuentan con tres secciones.
Preparación
Preparación de lo que se probara.
Llamada
Es la ejecución del método que se probara
Comprobación
Se evalúa el resultado.
Si se realizaran pruebas sobre los mismo objetos se puede refactorizar dicho código y colocarlo en un método que se nombra **setup **por defecto, a este método se lo coloca el decorador @Before, este nos ayuda a indicarle a los test que este método se debe lanzar antes en cada método.