

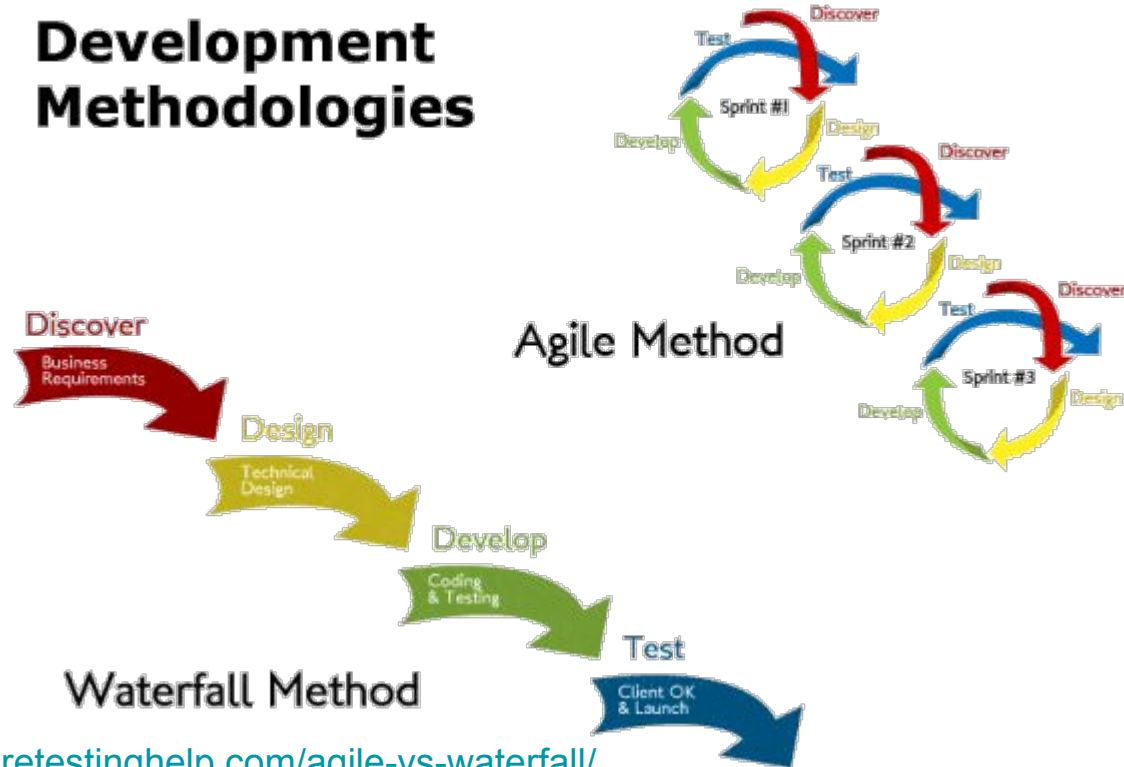
UNIT TESTING

TODAY'S OBJECTIVES

- Pros and cons of manual vs. automated testing
- Exploratory vs. Regression Testing
- Test Types:
 - Unit
 - Integration
 - Acceptance
- Writing unit tests
 - Choosing proper asserts
 - Boundary cases: what are they and how to find them

SDLC - SOFTWARE DEVELOPMENT LIFE CYCLE

Development Methodologies



TESTING

- Goes without saying... we need a way to test the code we've written.
- Testing is a critical part of EVERY SDLC methodology.
- The sooner you, as a developer, test, the sooner you identify problems and can move to QA, UAT, and Production.

MANUAL TESTING VS. AUTOMATED TESTING

- Historically, tests were written on a third party tool (i.e. Excel) with a script a tester should follow. The results are recorded.
 - This is a very error-prone manual process.
- Over time, testing frameworks were introduced so that we could write code that tests code in your system.
 - This made testing more automated.
 - However, the quality of the tests now partially depends on the developer's knowledge of the testing framework.

TYPES OF TESTING

- **Unit Testing**: Tests the smallest units possible (i.e. methods of a class).
- **Integration Testing**: Tests how various units or parts of the program interact with each other.
 - It can also be used to validate some external dependencies like database systems or API's.
- **User Acceptance Testing**: Tests the functionality from the end user's perspective. It can be conducted by a non-technical user.

OTHER TYPES OF TESTING

- **Security Testing**: Is our data safe from unauthorized users?
- **Performance Testing**: it works with 1 user, what about a million?
- **Platform Testing**: Works great on my laptop, what if I pull up the app from my phone?
- **Test-Driven Development (TDD)**: Code is written by creating tests that initially fail and writing all the needed code to make them pass.

UNIT TESTING IN JAVA: INTRODUCTION

The most commonly used testing framework in Java is **JUnit**.

- **JUnit** is written in Java and will leverage all the concepts you've learned so far: declaring variables, calling methods, instantiating objects.
- All related tests can be written in a single test class containing several methods, each method could be a test.
- Each method should contain an assertion, which compares the result of your code against an expected value.

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

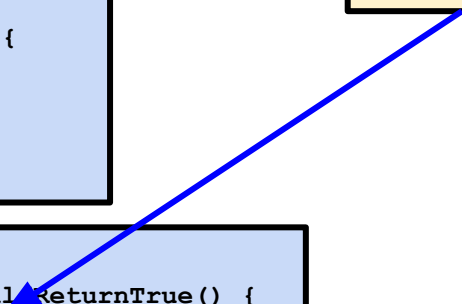
```
@Test  
public void isEven_withEvenValue_shouldReturnTrue() {  
    Example example = new Example();  
    boolean expected = true;  
    boolean result = example.isEven(6);  
    assertEquals(expected, result);  
}
```

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

Create object.



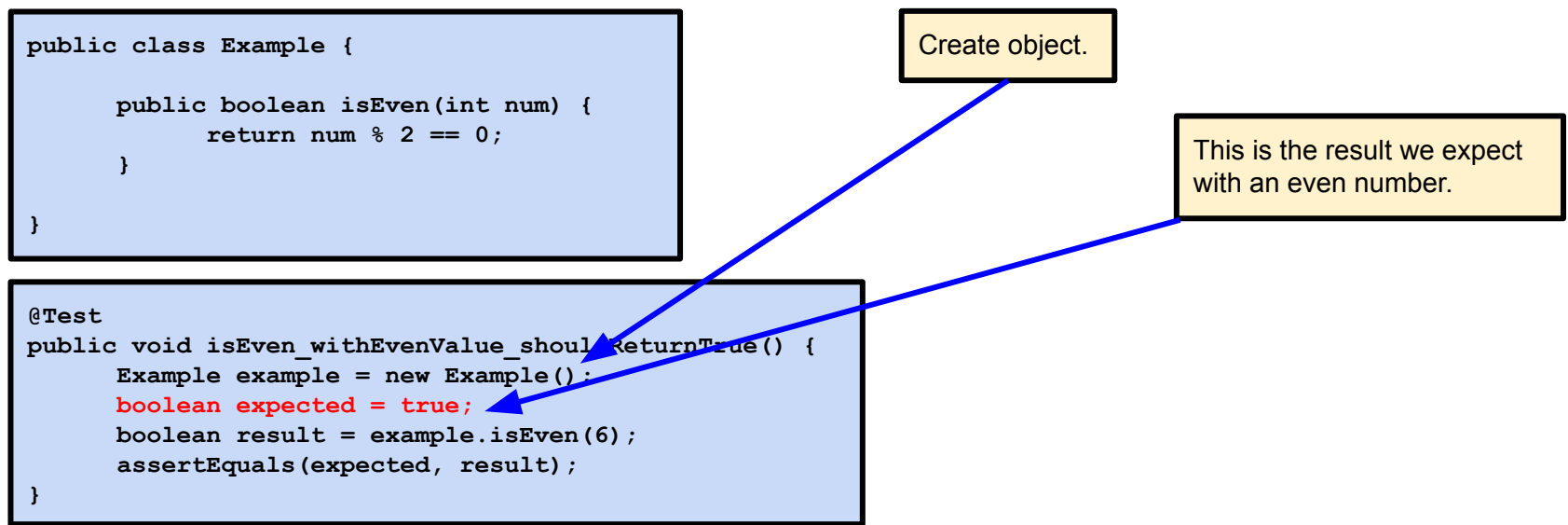
```
@Test  
public void isEven_withEvenValue_shouldReturnTrue() {  
    Example example = new Example();  
    boolean expected = true;  
    boolean result = example.isEven(6);  
    assertEquals(expected, result);  
}
```

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

Create object.



This is the result we expect with an even number.

```
@Test  
public void isEven_withEvenValue_shouldReturnTrue() {  
    Example example = new Example();  
    boolean expected = true;  
    boolean result = example.isEven(6);  
    assertEquals(expected, result);  
}
```

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

Create object.

This is the result we expect with an even number.

Call method with even number and store result.

```
@Test  
public void isEven_withEvenValue_shouldReturnTrue() {  
    Example example = new Example();  
    boolean expected = true;  
    boolean result = example.isEven(6);  
    assertEquals(expected, result);  
}
```

TESTING USING ASSERTIONS

JUnit leverages the concept of assertions. An assertion tests a condition and continues silently if the condition passes but fails with info about the condition that fails if the test does not pass.

```
public class Example {  
  
    public boolean isEven(int num) {  
        return num % 2 == 0;  
    }  
  
}
```

```
@Test  
public void isEven_withEvenValue_shouldReturnTrue() {  
    Example example = new Example();  
    boolean expected = true;  
    boolean result = example.isEven(6);  
    assertEquals(expected, result);  
}
```

Create object.

This is the result we expect with an even number.

Call method with even number and store result.

Compares expected to result, fails if they are not equal.

COMMON ASSERTS

- `assertEquals` / `assertNotEquals`
- `assertNotNull` / `assertNull`
- `assertTrue` / `assertFalse`
- `assertArrayEquals`
- `fail`
- `assertThat`

More info on using asserts:

<https://www.baeldung.com/junit-assertions> (only JUnit4 portion applies to what we will be doing)

<https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html>

USING ASSERTTHAT

- **assertThat** is different from the other asserts.
- It uses **Matchers** to allow more complex assert expressions.

JUnit uses the Hamcrest **Matchers** classes. [More on Hamcrest Matchers](#)

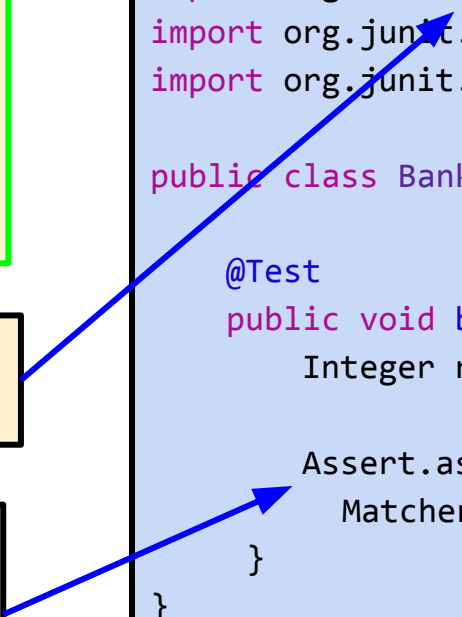
assertThat evaluates the **Matcher** expression using the actual value and a **Matcher** and fails if it evaluates to false.

```
import org.hamcrest.Matchers;
import org.junit.Assert;
import org.junit.Test;

public class BankAccountTests {

    @Test
    public void balance() {
        Integer result = 4;

        Assert.assertThat(result,
            Matchers.greaterThan(5));
    }
}
```



MORE ABOUT THE TEST CLASS

```
public class ExampleTest {  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        Example example = new Example();  
  
        boolean expected = true;  
        boolean result = example.isEven(6);  
  
        assertEquals(expected, result);  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        Example example = new Example();  
  
        boolean expected = false;  
        boolean result = example.isEven(9);  
  
        assertEquals(expected, result);  
    }  
  
}
```

MORE ABOUT THE TEST CLASS

@Test is an annotation indicating this is a test.

```
public class ExampleTest {  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        Example example = new Example();  
  
        boolean expected = true;  
        boolean result = example.isEven(6);  
  
        assertEquals(expected, result);  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        Example example = new Example();  
  
        boolean expected = false;  
        boolean result = example.isEven(9);  
  
        assertEquals(expected, result);  
    }  
}
```

MORE ABOUT THE TEST CLASS

@Test is an annotation indicating this is a test.

```
public class ExampleTest {  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        Example example = new Example();  
  
        boolean expected = true;  
        boolean result = example.isEven(6);  
  
        assertEquals(expected, result);  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        Example example = new Example();  
  
        boolean expected = false;  
        boolean result = example.isEven(9);  
  
        assertEquals(expected, result);  
    }  
}
```

There are two methods: one to test even case and one to test odd case. Tests are usually declared with void return type.

MORE ABOUT THE TEST CLASS

```
public class ExampleTest {  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        Example example = new Example();  
  
        boolean expected = true;  
        boolean result = example.isEven(6);  
  
        assertEquals(expected, result);  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        Example example = new Example();  
  
        boolean expected = false;  
        boolean result = example.isEven(9);  
  
        assertEquals(expected, result);  
    }  
}
```

@Test is an annotation indicating this is a test.

Evaluates result and fails if it doesn't match expected result.

There are two methods: one to test even case and one to test odd case. Tests are usually declared with void return type.

@BEFORE AND @AFTER METHODS

```
public class ExampleTest {  
  
    @Before  
    public void setUp() {  
        // do test setup  
    }  
  
    @After  
    public void tearDown() {  
        // do test cleanup  
    }  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        // test code  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        // test code  
    }  
  
}
```

@BEFORE AND @AFTER METHODS

A method annotated with `@Before` will run before **each** test.

```
public class ExampleTest {  
  
    @Before  
    public void setUp() {  
        // do test setup  
    }  
  
    @After  
    public void tearDown() {  
        // do test cleanup  
    }  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        // test code  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        // test code  
    }  
  
}
```

@BEFORE AND @AFTER METHODS

A method annotated with `@Before` will run before **each** test.

A method annotated with `@After` will run after **each** test.

```
public class ExampleTest {  
  
    @Before  
    public void setUp() {  
        // do test setup  
    }  
  
    @After  
    public void tearDown() {  
        // do test cleanup  
    }  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        // test code  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        // test code  
    }  
  
}
```

@BEFORE AND @AFTER METHODS

A method annotated with `@Before` will run before **each** test.

A method annotated with `@After` will run after **each** test.

```
public class ExampleTest {  
  
    @Before  
    public void setUp() {  
        // do test setup  
    }  
  
    @After  
    public void tearDown() {  
        // do test cleanup  
    }  
  
    @Test  
    public void isEven_withEvenValue_shouldReturnTrue() {  
        // test code  
    }  
  
    @Test  
    public void isEven_withOddValue_shouldReturnFalse() {  
        // test code  
    }  
  
}
```

Test flow:

1. `setUp()`
2. first test
3. `tearDown()`
4. `setUp()`
5. second test
6. `tearDown()`

UNIT TESTING STRUCTURE

- **Arrange**: begin by arranging the conditions of the test, such as setting up test data
- **Act**: perform the action of interest, i.e. the thing we're testing
- **Assert**: validate that the expected outcome occurred by means of an assertion (e.g. a certain value was returned, a file exists, etc).

LET'S LOOK AT SOME
EXAMPLES OF TESTS...

UNIT TESTING BEST PRACTICES

- No external dependencies
- One **logical** assertion per test (i.e. each test should only contain one "concept")
- Test code should be of the same quality as production code

HOW TO UNIT TEST

Find boundary cases in the code

- Is there an if statement?
 - Test around the condition that the if statement tests
- Is there a loop?
 - Test arrays in the loop that are empty, only one element, lots of element
- Is an object passed in?
 - Pass in null, an empty object, an object missing values that the method expects

LET'S *WRITE*
SOME TESTS!

USER STORY TO DEMO TDD

- As a user, I need to be able to transfer money from my account to another account.
- As a user, I should know my balance when a transfer is complete.
- As a user, I should not be able to transfer more than \$500 at once.
- As a user, I should not be able to overdraw my account when transferring money.