3
Red-Green-Refactor – From Failure Through Success until Perfection

"Knowing is not enough; we must apply. Willing is not enough; we must do."

– Bruce Lee

The **Red-Green-Refactor** technique is the basis of **test-driven development** (**TDD**). It is a game of ping pong in which we are switching between tests and implementation code at great speed. We'll fail, then we'll succeed, and, finally, we'll improve.

We'll develop a Tic-Tac-Toe game by going through each requirement one at a time. We'll write a test and see if it fails. Then, we'll write code that implements that test, run all the tests, and see them succeed. Finally, we'll refactor the code and try to make it better. This process will be repeated many times until all the requirements are successfully implemented.

We'll start by setting up the environment with Gradle and JUnit. Then, we'll go a bit deeper into the Red-Green-Refactor process. Once we're ready with the setup and theory, we'll go through the high-level requirements of the application.

With everything set, we'll dive right into the code—one requirement at a time. Once everything is done, we'll take a look at the code coverage and decide whether it is acceptable or whether more tests need to be added.

The following topics will be covered in this chapter:

- Setting up the environment with Gradle and JUnit
- The Red-Green-Refactor process
- Tic-Tac-Toe's requirements
- Developing Tic-Tac-Toe
- Code coverage
- More exercises

Setting up the environment with Gradle and JUnit

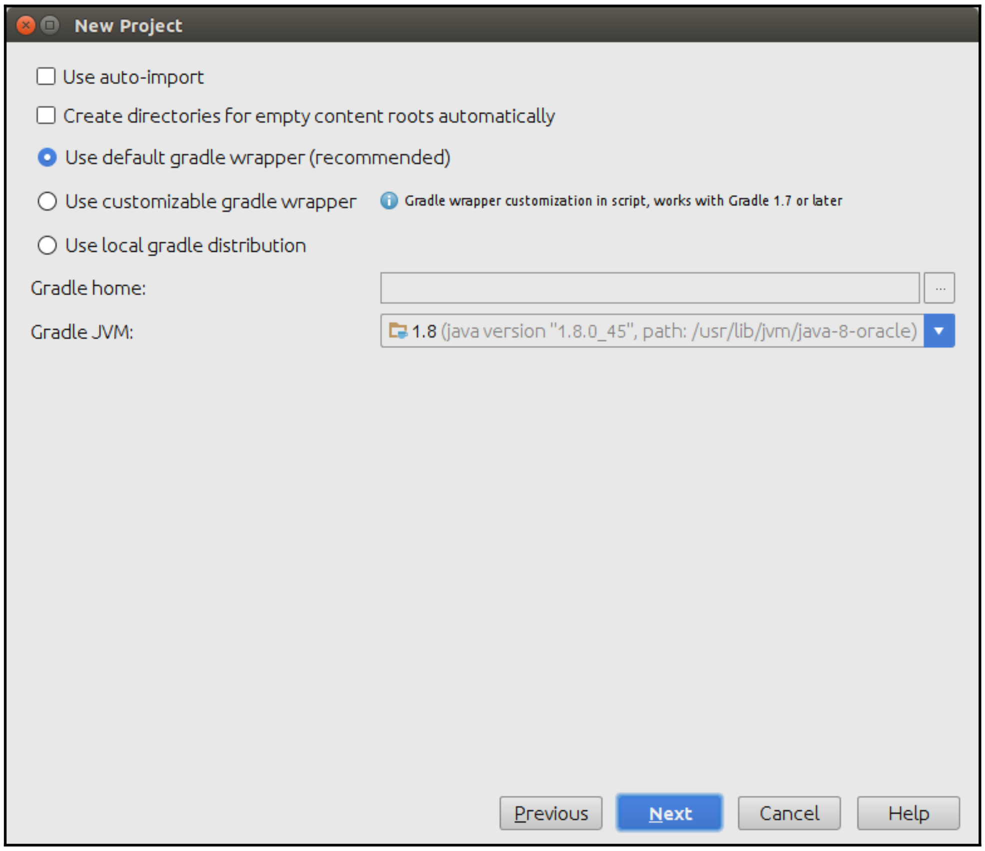
You are probably familiar with the setup of Java projects. However, you might not have worked with IntelliJ IDEA before or you might have used Maven instead of Gradle. In order to make sure that you can follow the exercise, we'll quickly go through the setup.

Setting up Gradle/Java project in IntelliJ IDEA

The main purpose of this book is to teach TDD, so we will not go into detail about Gradle and IntelliJ IDEA. Both are used as an example. All exercises in this book can be done with different choices for IDE and build tools. You can, for example, use Maven and Eclipse instead. For most, it might be easier to follow the same guidelines as those presented throughout the book, but the choice is yours.

The following steps will create a new Gradle project in IntelliJ IDEA:

1. Open **IntelliJ IDEA**. Click on **Create New Project** and select **Gradle** from the left-hand side menu. Then, click on **Next**.
2. If you are using IDEA 14 and higher, you will be asked for an **Artifact ID**. Type `tdd-java-ch03-tic-tac-toe` and click on **Next** twice. Type `tdd-java-ch03-tic-tac-toe` as the project name. Then, click on the **Finish** button:


New Project

- [ ] Use auto-import
- [ ] Create directories for empty content roots automatically
- [✓ Use default gradle wrapper (recommended)]
- Use customizable gradle wrapper — Gradle wrapper customization in script, works with Gradle 1.7 or later
- Use local gradle distribution

Gradle home:

Gradle JVM: 1.8 (java version "1.8.0_45", path: /usr/lib/jvm/java-8-oracle)

In the **New Project** dialog, we can observe that IDEA has already created the `build.gradle` file. Open it and you'll see that it already contains the JUnit dependency. Since this is our framework of choice in this chapter, there is no additional configuration that we should do. By default, `build.gradle` is set to use Java 1.5 as a source compatibility setting. You can change it to any version you prefer. The examples in this chapter will not use any of the Java features that came after Version 5, but that doesn't mean that you cannot solve the exercise using, for example, JDK 8.

Our `build.gradle` file should look like the following:

```
apply plugin: 'java'

version = '1.0'

repositories {
  mavenCentral()
```

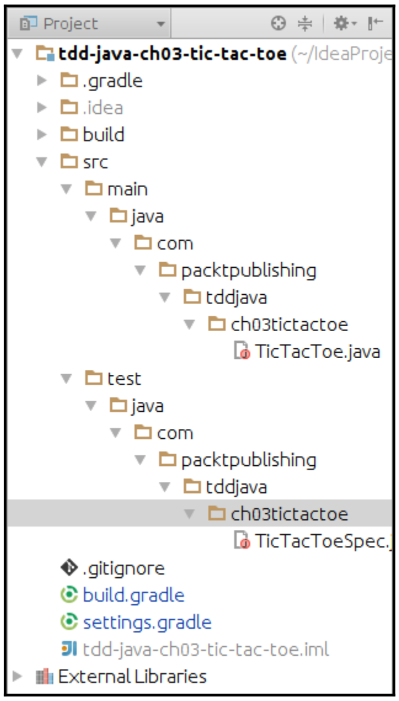
```
    }

    dependencies {
      testCompile group: 'junit', name: 'junit', version: '4.11'
    }
```

Now, all that's left to do is to create packages that we'll use for tests and the implementation. From the **Project** dialog, right-click to bring up the **Context** menu and select **New|Directory**. Type `src/test/java/com/packtpublishing/tddjava/ch03tictactoe` and click on the **OK** button to create the tests package. Repeat the same steps with the `src/main/java/com/packtpublishing/tddjava/ch03tictactoe` directory to create the implementation package.

Finally, we need to the make test and implementation classes. Create the `TicTacToeSpec` class inside the `com.packtpublishing.tddjava.ch03tictactoe` package in the `src/test/java` directory. This class will contain all our tests. Repeat the same for the `TicTacToe` class in the `src/main/java` directory.

Your **Project** structure should be similar to the one presented in the following screenshot:


```
Project
▼ tdd-java-ch03-tic-tac-toe (~/IdeaProje
    ▶ .gradle
    ▶ .idea
    ▶ build
    ▼ src
        ▼ main
            ▼ java
                ▼ com
                    ▼ packtpublishing
                        ▼ tddjava
                            ▼ ch03tictactoe
                                TicTacToe.java
        ▼ test
            ▼ java
                ▼ com
                    ▼ packtpublishing
                        ▼ tddjava
                            ▼ ch03tictactoe
                                TicTacToeSpec.
      .gitignore
      build.gradle
      settings.gradle
      tdd-java-ch03-tic-tac-toe.iml
▶ External Libraries
```

The source code can be found in the `00-setup` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at
`https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/00-setup`.

Always separate tests from the implementation code.

The benefits are as follows: this avoids accidentally packaging tests together with production binaries; many build tools expect tests to be in a certain source directory.

A common practice is to have at least two source directories. The implementation code should be located in `src/main/java` and the test code in `src/test/java`. In bigger projects, the number of source directories can increase, but the separation between implementation and tests should remain.

Build tools such as Maven and Gradle expect source directories, separation, as well as naming conventions.

That's it. We're set to start working on our Tic-Tac-Toe application using JUnit as the testing framework of choice and Gradle for compilation, dependencies, testing, and other tasks. In `Chapter 1`, *Why Should I Care for Test-Driven Development?*, you first encountered the Red-Green-Refactor procedure. Since it is the cornerstone of TDD and is the main objective of the exercise in this chapter, it might be a good idea to go into a bit more detail before we start the development.

The Red-Green-Refactor process

The Red-Green-Refactor process is the most important part of TDD. It is the main pillar, without which no other aspect of TDD will work.

The name comes from the states our code is in within the cycle. When in the red state, code does not work; when in the green state, everything is working as expected, but not necessarily in the best possible way. Refactor is the phase when we know that features are well covered with tests and thus gives us the confidence to change it and make it better.

Writing a test

Every new feature starts with a test. The main objective of this test is to focus on requirements and code design before writing the code. A test is a form of an executable documentation and can be used later on to get an understanding of what the code does or what are the intentions behind it.

At this point, we are in the red state since the execution of tests fails. There is a discrepancy between what tests expect from the code and what the implementation code actually does. To be more specific, there is no code that fulfills the expectation of the last test; we haven't written it yet. It is possible that at this stage all the tests are actually passing, but that's the sign of a problem.

Running all the tests and confirming that the last one is failing

Confirming that the last test is failing, confirms that the test would not, mistakenly, pass without the introduction of a new code. If the test is passing, then the feature already exists or the test is producing a false positive. If that's the case and the test actually always passes independently of the implementation, it is, in itself, worthless and should be removed.

A test must not only fail, but must fail for the expected reason. In this phase, we are still in the red stage. Tests were run and the last one failed.

Writing the implementation code

The purpose of this phase is to write code that will make the last test pass. Do not try to make it perfect, nor try to spend too much time with it. If it's not well-written or is not optimum, that is still okay. It'll become better later on. What we're really trying to do is to create a safety net in the form of tests that are confirmed to pass. Do not try to introduce any functionality that was not described in the last test. To do that, we are required to go back to the first step and start with a new test. However, we should not write new tests until all the existing ones are passing.

In this phase, we are still in the red stage. While the code that was written would probably pass all the tests, that assumption is not yet confirmed.

Running all the tests

It is very important that all the tests are run and not only the last test that was written. The code that we just wrote might have made the last test pass while breaking something else. Running all the tests confirms not only that the implementation of the last test is correct, but also that it did not break the integrity of the application as a whole. This slow execution of the whole test suite is a sign of poorly written tests or having too much coupling in the code. Coupling prevents the easy isolation of external dependencies, thus increasing the time required for the execution of tests.

In this phase, we are in the green state. All the tests are passing and the application behaves as we expect it to behave.

Refactoring

While all the previous steps are mandatory, this one is optional. Even though refactoring is rarely done at the end of each cycle, sooner or later it will be desired, if not mandatory. Not every implementation of a test requires refactoring. There is no rule that tells you when to refactor and when not to. The best time is as soon as one gets a feeling that the code can be rewritten in a better or more optimum way.

What constitutes a candidate for refactoring? This is a hard question to answer since it can have many answers—it's hard to understand code, the illogical location of a piece of code, duplication, names that do not clearly state a purpose, long methods, classes that do too many things, and so on. The list can go on and on. No matter what the reasons are, the most important rule is that refactoring cannot change any existing functionality.

Repeating

Once all the steps (with refactor being optional) are finished, we repeat them. At first glance, the whole process might seem too long or too complicated, but it is not. Experienced TDD practitioners write one to ten lines of code before switching to the next step. The whole cycle should last anything between a couple of seconds and no more than a few minutes. If it takes more than that, the scope of a test is too big and should be split into smaller chunks. Be fast, fail fast, correct, and repeat.

With this knowledge in mind, let us go through the requirements of the application we're about to develop using the Red-Green-Refactor process.

Tic-Tac-Toe game requirements

Tic-Tac-Toe is most often played by young children. The rules of the game are fairly simple.

Tic-Tac-Toe is a paper-and-pencil game for two players, *X* and *O*, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row, wins the game.

For more information about the game, please visit Wikipedia (`http://en.wikipedia.org/wiki/Tic-tac-toe`).

More detailed requirements will be presented later on.

The exercise consists of the creation of a single test that corresponds to one of the requirements. The test is followed by the code that fulfills the expectations of that test. Finally, if needed, the code is refactored. The same procedure should be repeated with more tests related to the same requirement. Once we're satisfied with tests and the implementation of that requirement, we'll move to the next one until they're all done.

In real-world situations, you wouldn't get such detailed requirements, but dive right into tests that would act as both requirements and validation. However, until you get comfortable with TDD, we'll have to define requirements separately from tests.

Even though all the tests and the implementation are provided, try to read only one requirement at a time and write tests and implementation code yourself. Once done, compare your solution with the one from this book and move to the next requirement. There is no one and only one solution; yours might be better than the ones presented here.

Developing Tic-Tac-Toe

Are you ready to code? Let's start with the first requirement.

Requirement 1 – placing pieces

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
- When a piece is placed on an occupied space, then `RuntimeException` is thrown

As you can see, the tests related to this first requirement are all about validations of the input argument. There is nothing in the requirements that says what should be done with those pieces.

Before we proceed with the first test, a brief explanation of how to test exceptions with JUnit is in order.

Starting from Release 4.7, JUnit introduced a feature called `Rule`. It can be used to do many different things (more information can be found at `https://github.com/junit-team/junit/wiki/Rules`), but in our case we're interested in the `ExpectedException` rule:

```
public class FooTest {
  @Rule
  public ExpectedException exception = ExpectedException.none();
  @Test
  public void whenDoFooThenThrowRuntimeException() {
    Foo foo = new Foo();
    exception.expect(RuntimeException.class);
    foo.doFoo();
  }
}
```

In this example, we defined that the `ExpectedException` is a rule. Later on, in the `doFooThrowsRuntimeException` test, we're specifying that we are expecting the `RuntimeException` to be thrown after the `Foo` class is instantiated. If it is thrown before, the test will fail. If the exception is thrown after, the test is successful.

`@Before` can be used to annotate a method that should be run before each test. It is a very useful feature with which we can, for example, instantiate a class used in tests or perform some other types of actions that should be run before each test:

```
private Foo foo;

@Before
public final void before() {
  foo = new Foo();
}
```

In this example, the `Foo` class will be instantiated before each test. This way, we can avoid having repetitive code that would instantiate `Foo` inside each test method.

Each test should be annotated with `@Test`. This tells `JunitRunner` which methods constitute tests. Each of them will be run in a random order so make sure that each test is self-sufficient and does not depend on the state that might be created by other tests:

```
@Test
public void whenSomethingThenResultIsSomethingElse() {
  // This is a test method
}
```

With this knowledge, you should be able to write your first test and follow it with the implementation. Once done, compare it with the solution provided.

Use descriptive names for test methods.

One of the benefits is that it helps to understand the objective of tests.

Using method names that describe tests is beneficial when trying to figure out why some tests failed or when the coverage should be increased with more tests. It should be clear what conditions are set before the test, what actions are performed, and what the expected outcome is.

There are many different ways to name test methods. My preferred method is to name them using the given/when/then syntax used in BDD scenarios. `Given` describes (pre)conditions, `When` describes actions, and `Then` describes the expected outcome. If a test does not have preconditions (usually set using the `@Before` and `@BeforeClass` annotations), `Given` can be skipped.

Do not rely only on comments to provide information about test objectives. Comments do not appear when tests are executed from your favorite IDE, nor do they appear in reports generated by the CI or build tools.

Besides writing tests, you'll need to run them as well. Since we are using Gradle, they can be run from the command prompt:

```
$ gradle test
```

IntelliJ IDEA provides a very good Gradle tasks model that can be reached by clicking on **View|Tool Windows|Gradle**. It lists all the tasks that can be run with Gradle (`test` being one of them).

The choice is yours—you can run tests in any way you see fit, as long as you run all of them.

Test – board boundaries I

We should start by checking whether a piece is placed within the boundaries of the 3x3 board:

```
package com.packtpublishing.tddjava.ch03tictactoe;

import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

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);
  }
}
```

When a piece is placed anywhere outside the *x*-axis, then `RuntimeException` is thrown.

In this test, we are defining that `RuntimeException` is expected when the `ticTacToe.play(5, 2)` method is invoked. It's a very short and easy test, and making it pass should be easy as well. All we have to do is create the `play` method and make sure that it throws `RuntimeException` when the `x` argument is smaller than 1 or bigger than 3 (the board is 3x3). You should run this test three times. The first time, it should fail because the `play` method doesn't exist. Once it is added, it should fail because `RuntimeException` is not thrown. The third time, it should be successful because the code that corresponds with this test is fully implemented.

Implementation

Now that we have a clear definition of when an exception should be thrown, the implementation should be straightforward:

```
package com.packtpublishing.tddjava.ch03tictactoe;

public class TicTacToe {
  public void play(int x, int y) {
    if (x < 1 || x > 3) {
      throw new RuntimeException("X is outside board");
    }
  }
}
```

As you can see, this code does not contain anything else, but the bare minimum required for the test to pass.

Some TDD practitioners tend to take minimum as a literal meaning. They would have the `play` method with only the `throw new RuntimeException();` line. I tend to translate minimum to as little as possible within reason.

We're not adding numbers, nor are we returning anything. It's all about making small changes very fast. (Remember the game of ping pong?) For now, we're doing red-green steps. There's not much we can do to improve this code so we're skipping the refactoring.

Let's move on to the next test.

Test – board boundaries II

This test is almost the same as the previous one. This time we should validate the *y*-axis:

```
@Test
public void whenYOutsideBoardThenRuntimeException() {
  exception.expect(RuntimeException.class);
  ticTacToe.play(2, 5);
}
```

When a piece is placed anywhere outside the *y*-axis, then `RuntimeException` is thrown.

Implementation

The implementation of this specification is almost the same as the previous one. All we have to do is throw an exception if `y` does not fall within the defined range:

```
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");
  }
}
```

In order for the last test to pass, we had to add the `else` clause that checks whether `Y` is inside the board.

Let's do the last test for this requirement.

Test – occupied spot

Now that we know that pieces are placed within the board's boundaries, we should make sure that they can be placed only on unoccupied spaces:

```
@Test
public void whenOccupiedThenRuntimeException() {
  ticTacToe.play(2, 1);
  exception.expect(RuntimeException.class);
  ticTacToe.play(2, 1);
}
```

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

That's it; this was our last test. Once the implementation is finished, we can consider the first requirement as done.

Implementation

To implement the last test, we should store the location of the placed pieces in an array. Every time a new piece is placed, we should verify that the place is not occupied, or else throw an exception:

```
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';
  }
}
```

We're checking whether a place that was played is occupied and, if it is not, we're changing the array entry value from empty (`\0`) to occupied (`X`). Keep in mind that we're still not storing who played (`X` or `O`).

Refactoring

While the code that we have done so far fulfills the requirements set by the tests, it looks a bit confusing. If someone read it, it would not be clear as to what the `play` method does. We should refactor it by moving the code into separate methods. The refactored code will look like the following:

```
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';
  }
}
```

With this refactoring, we did not change the functionality of the `play` method. It behaves exactly the same as it behaved before, but the new code is a bit more readable. Since we had tests that covered all the existing functionality, there was no fear that we might do something wrong. As long as all tests are passing all the time and refactoring did not introduce any new behavior, it is safe to make changes to the code.

The source code can be found in the `01-exceptions` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at `https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/01-exceptions`.

Requirement 2 – adding two-player support

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 player `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`

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.*;
```

In their essence, methods inside the `Assert` class are very simple. Most of them start with `assert`. For example, `assertEquals` compares two objects—`assertNotEquals` verifies that two objects are not the same and `assertArrayEquals` verifies that two arrays are the same. Each of those asserts has many overloaded variations so that almost any type of Java object can be used.

In our case, we'll need to compare two characters. The first is the one we're expecting and the second one is the actual character retrieved from the `nextPlayer` method.

Now it's time to write those tests and the implementation.

Write the test before writing the implementation code .

The benefits of doing this are as follows—it ensures that testable code is written and ensures that every line of code gets tests written for it.

By writing or modifying the test first, the developer is focused on requirements before starting to work on a code. This is the main difference when compared to writing tests after the implementation is done. An additional benefit is that with tests first, we are avoiding the danger that the tests work as quality checking instead of quality assurance.

Test – X plays first

Player `X` has the first turn:

```
@Test
public void givenFirstTurnWhenNextPlayerThenX() {
  assertEquals('X', ticTacToe.nextPlayer());
}
```

The first turn should be played by Player `X`.

This test should be self-explanatory. We are expecting the `nextPlayer` method to return `X`. If you try to run this, you'll see that the code does not even compile. That's because the `nextPlayer` method does not even exist. Our job is to write the `nextPlayer` method and make sure that it returns the correct value.

Implementation

There's no real need to check whether it really is the player's first turn or not. As it stands, this test can be fulfilled by always returning X. Later tests will force us to refine this code:

```
public char nextPlayer() {
  return 'X';
}
```

Test – O plays right after X

Now, we should make sure that players are changing. After X is finished, it should be O's turn, then again X, and so on:

```
@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

In order to track who should play next, we need to store who played last:

```
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';
}
```

You are probably starting to get the hang of it. Tests are small and easy to write. With enough experience, it should take a minute, if not seconds, to write a test and as much time or less to write the implementation.

Test – X plays right after O

Finally, we can check whether `X`'s turn comes after `O` played.

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

There's nothing to do to fulfill this test and, therefore, the test is useless and should be discarded. If you write this test, you'll discover that it is a false positive. It would pass without changing the implementation; try it out. Write this test and if it is successful without writing any implementation code, discard it.

The source code can be found in the `02-next-player` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at `https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/02-next-player`.

Requirement 3 – adding winning conditions

It's time to work on winning according to 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 – by default there's no winner

Let's start by defining the default response of the `play` method:

```
@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

The default return values are always easiest to implement and this one is no exception:

```
public String play(int x, int y) {
  checkAxis(x);
  checkAxis(y);
  setBox(x, y);
  lastPlayer = nextPlayer();
  return "No winner";
}
```

Test – winning condition I

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 on 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;
  }
}
```

Refactoring

The preceding code satisfies the tests, but is not necessarily the final version. It served its purpose of getting code coverage as quickly as possible. Now, since we have tests that guarantee the integrity of the expected behavior, we can refactor the code:

```
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;
}
```

This refactored solution looks better. The `play` method keeps being short and easy to understand. Winning logic is moved to a separate method. Not only have we kept the `play` method's purpose clear, but this separation also allows us to grow the winning condition's code in separation from the rest.

Test – winning condition II

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 (board[i][0] + board[i][1] + board[i][2] ==
playerTotal) {
      return true;
    }
  }
  return false;
}
```

Test – winning condition III

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); // X
  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

Since there is only one line that can constitute with the requirement, we can check it directly without any loops:

```
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 (board[i][0] + board[i][1] + board[i][2] ==
playerTotal) {
      return true;
    }
  }
  if (board[0][0] + board[1][1] + board[2][2] == playerTotal) {
    return true;
  }
  return false;
}
```

Test – winning condition IV

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); // X
  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 (board[i][0] + board[i][1] + board[i][2] ==
playerTotal) {
      return true;
    }
  }
  if (board[0][0] + board[1][1] + board[2][2] == playerTotal) {
    return true;
  } else if (board[0][2] + board[1][1] + board[2][0] ==
playerTotal) {
    return true;
  }
  return false;
}
```

Refactoring

The way we're handling possible diagonal wins, the calculation doesn't look right. Maybe the reutilization of the existing loop would make more sense:

```
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 (board[i][0] + board[i][1] + board[i][2] ==
playerTotal) {
      return true;
    }
  }
  if (diagonal1 == playerTotal || diagonal2 == playerTotal) {
    return true;
  }
  return false;
```

```
            }
```

The source code can be found in the `03-wins` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at
`https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/03-wins`.

Now, let's go through the last requirement.

Requirement 4 – tie conditions

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

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

Test – handling a tie situation

We can test the draw result by filling all the board's boxes:

```
@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);
}
```

Implementation

Checking whether it's a draw is fairly straightforward. All we have to do is check whether all the board's boxes are filled. We can do that by iterating through the board array:

```
public String play(int x, int y) {
  checkAxis(x);
  checkAxis(y);
  lastPlayer = nextPlayer();
  setBox(x, y, lastPlayer);
  if (isWin()) {
    return lastPlayer + " is the winner";
  } else if (isDraw()) {
    return "The result is draw";
  } else {
    return "No winner";
  }
}

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;
}
```

Refactoring

Even though the `isWin` method is not the scope of the last test, it can still be refactored even more. For once, we don't need to check all the combinations, but only those related to the position of the last piece played. The final version could look like the following:

```
private boolean isWin(int x, int y) {
  int playerTotal = lastPlayer * 3;
  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;
      }
```

Refactoring can be done on any part of the code at any time, as long as all the tests are successful. While it's often easiest and fastest to refactor the code that was just written, going back to something that was written the other day, previous month, or even years ago, is more than welcome. The best time to refactor something is when someone sees an opportunity to make it better. It doesn't matter who wrote it or when; making the code better is always a good thing to do.

The source code can be found in the `04-draw` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at `https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/04-draw`.

Code coverage

We did not use code coverage tools throughout this exercise. The reason is that we wanted you to be focused on the Red-Green-Refactor model. You wrote a test, saw it fail, wrote the implementation code, saw that all the tests were executed successfully, refactored the code whenever you saw an opportunity to make it better, and then you repeated the process. Did our tests cover all cases? That's something that code coverage tools such as JaCoCo can answer. Should you use those tools? Probably, only in the beginning. Let me clarify that. When you are starting with TDD, you will probably miss some tests or implement more than what the tests defined. In those cases, using code coverage is a good way to learn from your own mistakes. Later on, the more experienced you become with TDD, the less of a need you'll have for such tools. You'll write tests and just enough of the code to make them pass. Your coverage will be high with or without tools such as JaCoCo. There will be a small amount of code not covered by tests because you'll make a conscious decision about what is not worth testing.

Tools such as JaCoCo were designed mostly as a way to verify that the tests written after the implementation code are providing enough coverage. With TDD, we are taking a different approach with the inverted order (tests before the implementation).

Still, we suggest you use JaCoCo as a learning tool and decide for yourself whether to use it in the future.

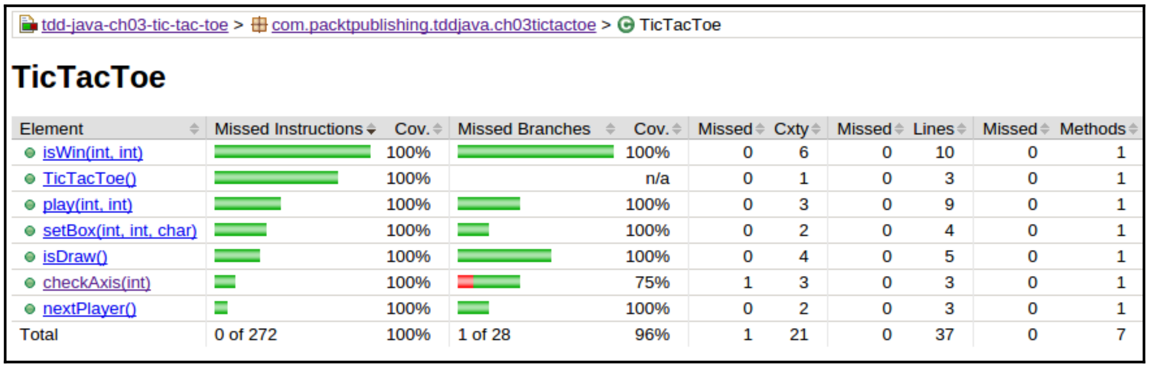
To enable JaCoCo within Gradle, add the following to `build.gradle`:

```
apply plugin: 'jacoco'
```

From now on, Gradle will collect JaCoCo metrics every time we run tests. Those metrics can be transformed into a nice report using the `jacocoTestReport` Gradle target. Let's run our tests again and see what the code coverage is:

```
$ gradle clean test jacocoTestReport
```

The end result is the report located in the `build/reports/jacoco/test/html` directory. Results will vary depending on the solution you made for this exercise. My results say that there is a 100% of instructions coverage and 96% of branches coverage; 4% is missing because there was no test case where the player played on a box 0 or negative. The implementation of that case is there, but there is no specific test that covers it. Overall, this is a pretty good coverage:


tdd-java-ch03-tic-tac-toe > com.packtpublishing.tddjava.ch03tictactoe > TicTacToe

TicTacToe

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isWin(int, int)	🟩	100%	🟩	100%	0	6	0	10	0	1
TicTacToe()	🟩	100%		n/a	0	1	0	3	0	1
play(int, int)	🟩	100%	🟩	100%	0	3	0	9	0	1
setBox(int, int, char)	🟩	100%	🟩	100%	0	2	0	4	0	1
isDraw()	🟩	100%	🟩	100%	0	4	0	5	0	1
checkAxis(int)	🟩	100%	🟥🟩	75%	1	3	0	3	0	1
nextPlayer()	🟩	100%	🟩	100%	0	2	0	3	0	1
Total	0 of 272	100%	1 of 28	96%	1	21	0	37	0	7

JaCoCo will be added in the source code. This is found in the `05-jacoco` branch of the `tdd-java-ch03-tic-tac-toe` Git repository at `https://bitbucket.org/vfarcic/tdd-java-ch03-tic-tac-toe/branch/05-jacoco`.

More exercises

We just developed one (most commonly used) variation of the Tic-Tac-Toe game. As an additional exercise, pick one or more variations from Wikipedia (`http://en.wikipedia.org/wiki/Tic-tac-toe`) and implement it using the Red-Green-Refactor procedure. When finished, implement a kind of AI that would play `O`'s turns. Since Tic-Tac-Toe usually leads to a draw, AI can be considered finished when it successfully reaches a draw for any combination of `X`'s moves.

While working on those exercises, remember to be fast and play ping pong. Also, most of all, remember to use the Red-Green-Refactor procedure.

Summary

We managed to finish our Tic-Tac-Toe game using the Red-Green-Refactor process. The examples themselves were simple and you probably didn't have a problem following them.

The objective of this chapter was not to dive into something complicated (that comes later), but to get into the habit of using the short and repetitive cycle called Red-Green-Refactor.

We learned that the easiest way to develop something is by splitting it into very small chunks. The design was emerging from tests instead of using a big upfront approach. No line of the implementation code was written without writing a test first and seeing it fail. By confirming that the last test fails, we are confirming that it is valid (it's easy to make a mistake and write a test that is always successful) and the feature we are about to implement does not exist. After the test failed, we wrote the implementation of that test. While writing the implementation, we tried to make it a minimal one with the objective being to make the test pass, not to make the solution final. We repeated this process until we felt that there was a need to refactor the code. Refactoring did not introduce any new functionality (we did not change what the application does), but made the code more optimal and easier to read and maintain.

In the next chapter, we'll elaborate in more detail about what constitutes a unit within the context of TDD and how to approach the creation of tests based on those units.