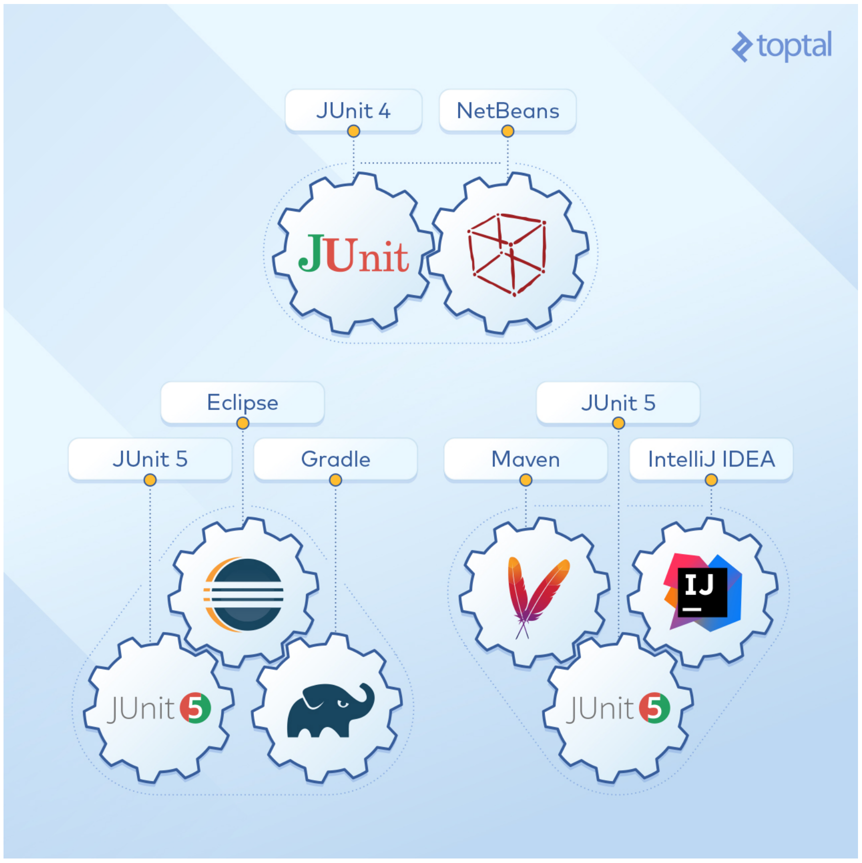


Gradle

As a sample Gradle project for JUnit 5, see the Gradle section of the JUnit user guide and the junit5-samples.git repository. Note that it can also run tests that use the JUnit 4 API (referred to as *"vintage"*).

The project can be created in IntelliJ via the menu option File > Open… > navigate to the `junit-gradle-consumer sub-directory` > OK > Open as Project > OK to import the project from Gradle.

For Eclipse, the Buildship Gradle plugin can be installed from Help > Eclipse Marketplace… The project can then be imported with File > Import… > Gradle > Gradle Project > Next > Next > Browse to the `junit-gradle-consumer` sub-directory > Next > Next > Finish.

After setting up the Gradle project in IntelliJ or Eclipse, running the Gradle `build` task will include running all of the JUnit tests with the `test` task. Note that the tests might be skipped on subsequent executions of `build` if no changes were made to the code.

For JUnit 4, see JUnit's use with Gradle wiki.

Maven

For JUnit 5, refer to the Maven section of the user guide and the junit5-samples.git repository for an example of a Maven project. This can also run vintage tests (ones that use the JUnit 4 API).

In IntelliJ, use File > Open… > navigate to `junit-maven-consumer/pom.xml` > OK > Open as Project. The tests can then be run from Maven Projects > junit5-maven-consumer > Lifecycle > Test.

In Eclipse, use File > Import… > Maven > Existing Maven Projects > Next > Browse to the `junit-maven-consumer` directory > With the `pom.xml` selected > Finish.

The tests can be executed by running the project as Maven build… > specify goal of `test` > Run.

For JUnit 4, see JUnit in the Maven repository.

Development Environments

In addition to running tests through build tools like Gradle or Maven, many IDEs can directly run JUnit tests.

IntelliJ IDEA

IntelliJ IDEA 2016.2 or later is required for JUnit 5 tests, while JUnit 4 tests should work in older IntelliJ versions.

For the purposes of this article, you may want to create a new project in IntelliJ from one of my GitHub repositories (JUnit5IntelliJ.git or JUnit4IntelliJ.git), which include all the files in the simple `Person` class example and use the built-in JUnit libraries. The test can be run with Run > Run 'All Tests'. The test can also be run in IntelliJ from the `PersonTest` class.

These repositories were created with new IntelliJ Java projects and build out the directory structures `src/main/java/com/example` and `src/test/java/com/example`. The `src/main/java` directory was specified as a source folder while `src/test/java` was specified as a test source folder. After creating the `PersonTest` class with a test method annotated with `@Test`, it may fail to compile, in which case IntelliJ offers the suggestion to add JUnit 4 or JUnit 5 to the class path which can be loaded from the IntelliJ IDEA distribution (see these answers on Stack Overflow for more details). Finally, a JUnit run configuration was added for All Tests.

See also the IntelliJ Testing How-to Guidelines.

Eclipse

An empty Java project in Eclipse will not have a test root directory. This has be added from project Properties > Java Build Path > Add Folder… > Create New Folder… > specify the Folder name > Finish. The new directory will be selected as a source folder. Click OK in both remaining dialogs.

JUnit 4 tests can be created with File > New > JUnit Test Case. Select "New JUnit 4 test" and the newly created source folder for tests. Specify a "class under test" and a "package," making sure the package matches the class under test. Then, specify a name for the test class. After finishing the wizard, if prompted, choose to "Add JUnit 4 library" to the build path. The project or individual test class can then be run as a JUnit Test. See also Eclipse Writing and Running JUnit tests.

NetBeans

NetBeans only supports JUnit 4 tests. Test classes can be created in a NetBeans Java project with File > New File… > Unit Tests > JUnit Test or Test for Existing Class. By default, the test root directory is named `test` in the project directory.

Simple Production Class and its JUnit Test Case

Let's take a look at a simple example of production code and its corresponding unit test code for a very simple `Person` class. You can download the sample code from my github project and open it via IntelliJ.

src/main/java/com/example/Person.java

```
package com.example;

class Person {
   private final String givenName;
   private final String surname;

   Person(String givenName, String surname) {
       this.givenName = givenName;
       this.surname = surname;
   }

   String getDisplayName() {
       return surname + ", " + givenName;
   }
}
```

The immutable `Person` class has a constructor and a `getDisplayName()` method. We want to test that `getDisplayName()` returns the name formatted as we expect. Here is test code for a single unit test (JUnit 5):

src/test/java/com/example/PersonTest.java

```
package com.example;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PersonTest {

   @Test
   void testGetDisplayName() {
       Person person = new Person("Josh", "Hayden");
       String displayName = person.getDisplayName();
       assertEquals("Hayden, Josh", displayName);
   }
}
```

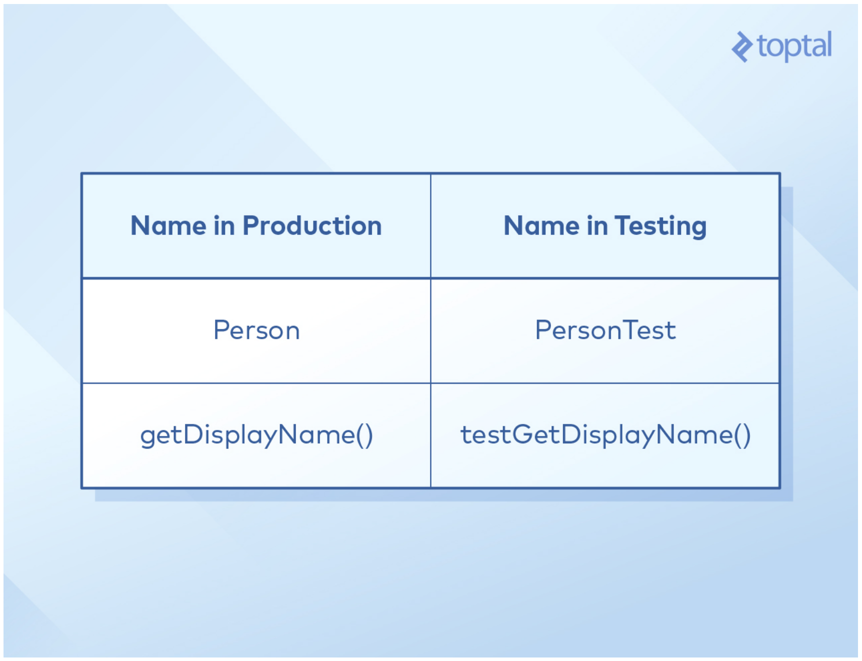
`PersonTest` uses JUnit 5's `@Test` and assertion. For JUnit 4, the `PersonTest` class and method need to be public and different imports should be used. Here's the JUnit 4 example Gist.

Upon running the `PersonTest` class in IntelliJ, the test passes and the UI indicators are green.

Common JUnit Conventions

Naming

Though not required, we use common conventions in naming the test class; specifically, we start with the name of the class being tested (`Person`) and append "Test" to it (`PersonTest`). Naming the test method is similar, starting with the method being tested (`getDisplayName()`) and prepending "test" to it (`testGetDisplayName()`). While there are many other perfectly acceptable conventions for naming test methods, it's important to be consistent across the team and project.



Name in Production	Name in Testing
Person	PersonTest
getDisplayName()	testGetDisplayName()

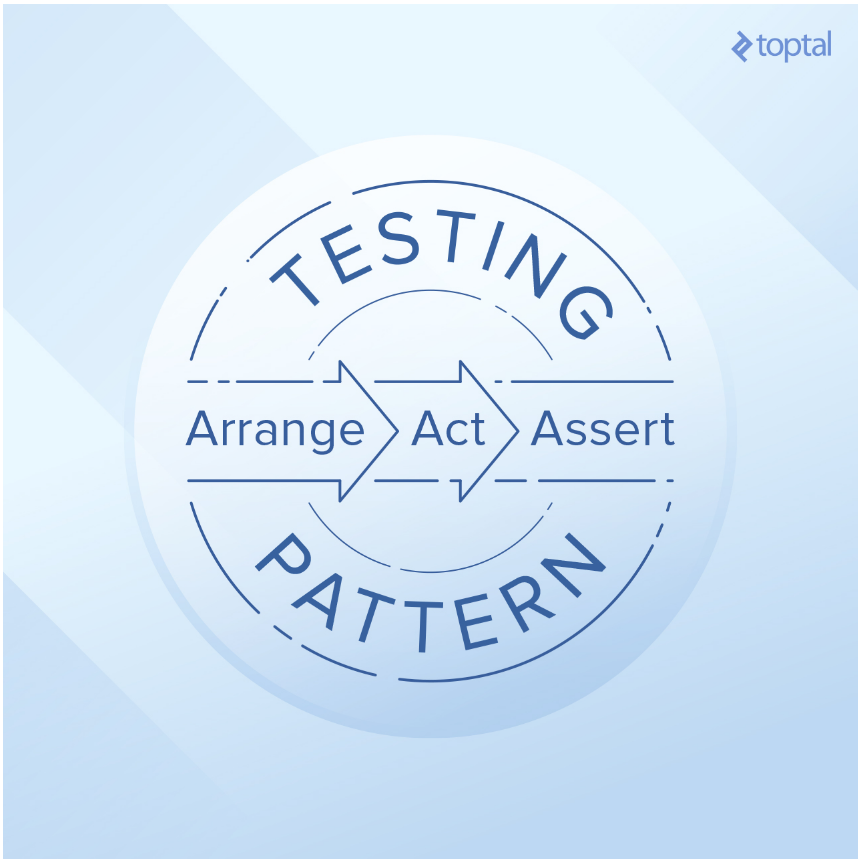
Packages

We also employ the convention of creating the test code `PersonTest` class in the same package (`com.example`) as the production code's `Person` class. If we used a different package for tests, we would be required to use the public access modifier in production code classes, constructors, and methods referenced by unit tests, even where it's not appropriate, so it's better to just keep them in the same package. We do, however, use separate source directories (`src/main/java` and `src/test/java`) as we generally do not want to include test code in released production builds.

Structure and Annotation

The `@Test` annotation (JUnit 4/5) tells JUnit to execute the `testGetDisplayName()` method as a test method and report whether it passes or fails. As long as all assertions (if any) pass and no exceptions are thrown, the test is considered to pass.

Our test code follows the structure pattern of Arrange-Act-Assert (AAA). Other common patterns include Given-When-Then and Setup-Exercise-Verify-Teardown (Teardown is typically not explicitly needed for unit tests), but we use AAA in this article.



Let's take a look at how our test example follows AAA. The first line, the "arrange" creates a `Person` object that will be tested:

```
        Person person = new Person("Josh", "Hayden");
```

The second line, the "act," *exercises* the production code's `Person.getDisplayName()` method:

```
        String displayName = person.getDisplayName();
```

The third line, the "assert," verifies that the result is as expected.

```
        assertEquals("Hayden, Josh", displayName);
```

Internally, the `assertEquals()` call uses the "Hayden, Josh" String object's equals method to verify the actual value returned from the production code (`displayName`) matches. If it didn't match, the test would have been marked failed.

Note that tests often have more than one line for each of these AAA phases.

Unit Tests and Production Code

Now that we've covered some testing conventions, let's turn our attention to making production code testable.

We return to our `Person` class, where I've implemented a method to return a person's age based on his or her date of birth. The code examples require Java 8 to take advantage of new date and functional APIs. Here's what the new `Person.java` class looks like:

Person.java

```
// ...
class Person {
    // ...
    private final LocalDate dateOfBirth;

    Person(String givenName, String surname, LocalDate dateOfBirth) {
        // ...
        this.dateOfBirth = dateOfBirth;
    }

    // ...

    long getAge() {
        return ChronoUnit.YEARS.between(dateOfBirth, LocalDate.now());
    }

    public static void main(String... args) {
        Person person = new Person("Joey", "Doe", LocalDate.parse("2013-01-12"));
        System.out.println(person.getDisplayName() + ": " + person.getAge() + " years");
        // Doe, Joey: 4 years
    }
}
```

Running this class (at the time of writing) announces that Joey is 4 years old. Let's add a test method:

PersonTest.java

```
// ...
class PersonTest {
    // ...

    @Test
    void testGetAge() {
        Person person = new Person("Joey", "Doe", LocalDate.parse("2013-01-12"));
        long age = person.getAge();
        assertEquals(4, age);
    }
}
```

It passes today, but what about when we run it one year from now? This test is non-deterministic and brittle as the expected result depends on the current date of the system running the test.

Stubbing and Injecting a Value Supplier

When running in production we want to use the current date, `LocalDate.now()`, for calculating the person's age, but to make a deterministic test even in a year from now, tests need to supply their own `currentDate` values.

This is known as dependency injection. We don't want our `Person` object to determine the current date itself, but instead we want to pass in this logic as a dependency. Unit tests will use a known, stubbed value, and production code will allow the actual value to be provided by the system at runtime.

Let's add a `LocalDate` supplier to `Person.java`:

Person.java

```
// ...
class Person {
    // ...
    private final LocalDate dateOfBirth;
    private final Supplier<LocalDate> currentDateSupplier;

    Person(String givenName, String surname, LocalDate dateOfBirth) {
        this(givenName, surname, dateOfBirth, LocalDate::now);
    }
    // Visible for testing
    Person(String givenName, String surname, LocalDate dateOfBirth, Supplier<LocalDate> currentDateSupplier) {
        // ...
        this.dateOfBirth = dateOfBirth;
        this.currentDateSupplier = currentDateSupplier;
    }

    // ...

    long getAge() {
        return ChronoUnit.YEARS.between(dateOfBirth, currentDateSupplier.get());
    }

    public static void main(String... args) {
        Person person = new Person("Joey", "Doe", LocalDate.parse("2013-01-12"));
        System.out.println(person.getDisplayName() + ": " + person.getAge() + " years");
        // Doe, Joey: 4 years
    }
}
```

To make it easier to test the `getAge()` method, we changed it to use `currentDateSupplier`, a `LocalDate` supplier, for retrieving the current date. If you don't know what a supplier is, I recommend reading about Lambda Built-in Functional Interfaces.

We also added a dependency injection: The new testing constructor allows tests to supply their own current date values. The original constructor calls this new constructor, passing a static method reference of `LocalDate::now`, which supplies a `LocalDate` object, so our main method still works as before. What about our test method? Let's update `PersonTest.java`:

PersonTest.java

```
// ...
class PersonTest {
    // ...

    @Test
    void testGetAge() {
        LocalDate dateOfBirth = LocalDate.parse("2013-01-02");
        LocalDate currentDate = LocalDate.parse("2017-01-17");
        Person person = new Person("Joey", "Doe", dateOfBirth, ()->currentDate);
        long age = person.getAge();
        assertEquals(4, age);
    }
}
```

The test now injects its own `currentDate` value, so our test will still pass when run next year, or during any year. This is commonly referred to as **stubbing**, or providing a known value to be returned, but we first had to change `Person` to allow this dependency to be injected.

Note the lambda syntax (`()->currentDate`) when constructing the `Person` object. This is treated as a supplier of a `LocalDate`, as required by the new constructor.

Mocking and Stubbing a Web Service

We are ready for our `Person` object—whose entire existence has been in JVM memory—to communicate with the outside world. We want to add two methods: the `publishAge()` method, which will post the person's current age, and the `getThoseInCommon()` method, which will return names of famous people that share the same birthday or are the same age as our `Person`. Suppose there is a RESTful service we can interact with called "People Birthdays." We have a Java client for it that consists of the single class, `BirthdaysClient`.

com.example.birthdays.BirthdaysClient

```
package com.example.birthdays;

import java.io.IOException;
import java.util.Arrays;
import java.util.Collection;

public class BirthdaysClient {

    public void publishRegularPersonAge(String name, long age) throws IOException {
        System.out.println("publishing " + name + "'s age: " + age);
        // HTTP POST with name and age and possibly throw an exception
    }

    public Collection<String> findFamousNamesOfAge(long age) throws IOException {
        System.out.println("finding famous names of age " + age);
        return Arrays.asList(/* HTTP GET with age and possibly throw an exception */);
    }

    public Collection<String> findFamousNamesBornOn(int month, int dayOfMonth) throws IOException {
        System.out.println("finding famous names born on day " + dayOfMonth + " of month " + month);
        return Arrays.asList(/* HTTP GET with month and day and possibly throw an exception */);
    }
}
```

Let's enhance our `Person` class. We start by adding a new test method for the desired behavior of `publishAge()`. Why start with the test, rather than the functionality? We're following the principles of test-driven development (also known as TDD), wherein we write the test first, and then the code to make it pass.

PersonTest.java

```
// …
class PersonTest {
    // …

    @Test
    void testPublishAge() {
        LocalDate dateOfBirth = LocalDate.parse("2000-01-02");
        LocalDate currentDate = LocalDate.parse("2017-01-01");
        Person person = new Person("Joe", "Sixteen", dateOfBirth, ()->currentDate);
        person.publishAge();
    }
}
```

At this point, the test code fails to compile because we haven't created the `publishAge()` method that it's calling. Once we create an empty `Person.publishAge()` method, everything passes. We are now ready for the test to verify that the person's age actually gets published to the `BirthdaysClient`.

Adding a Mocked Object

As this is a unit test, it should run fast and in memory, so the test will construct our `Person` object with a mock `BirthdaysClient` so it doesn't actually make a web request. The test will then use this mock object to verify that it was called as expected. To do this, we'll add a dependency on the Mockito framework (MIT license) for creating mock objects, and then create a mocked `BirthdaysClient` object:

PersonTest.java

```
// ...
import com.example.birthdays.BirthdaysClient;
// ...
import static org.mockito.Mockito.mock;

class PersonTest {
    private BirthdaysClient birthdaysClient = mock(BirthdaysClient.class);

    // ...

    @Test
    void testPublishAge() {
        // ...
        Person person = new Person("Joe", "Sixteen", dateOfBirth, ()->currentDate, birthdaysClient);
        // ...
    }
}
```

We furthermore augmented the signature of the `Person` constructor to take a `BirthdaysClient` object, and changed the test to inject the mocked `BirthdaysClient` object.

Adding a Mock Expectation

Next, we add to the end of our `testPublishAge` an expectation that the `BirthdaysClient` is called. `Person.publishAge()` should call it, as shown in our new `PersonTest.java`:

PersonTest.java

```
// ...
class PersonTest {
    // ...

    @Test
    void testPublishAge() throws IOException {
        // ...
        Person person = new Person("Joe", "Sixteen", dateOfBirth, ()->currentDate, birthdaysClient);
        verifyZeroInteractions(birthdaysClient);
        person.publishAge();
        verify(birthdaysClient).publishRegularPersonAge("Joe Sixteen", 16);
    }
}
```

Our Mockito-enhanced `BirthdaysClient` keeps track of all calls that have been made to its methods, which is how we verify that no calls have been made to `BirthdaysClient` with the `verifyZeroInteractions()` method before calling `publishAge()`. Though arguably not necessary, by doing this we ensure the constructor is not making any rogue calls. On the `verify()` line, we specify how we expect the call to `BirthdaysClient` to look.

Note, that because publishRegularPersonAge has the IOException in its signature, we add it to our test method signature, too.

At this point, the test fails:

```
Wanted but not invoked:
birthdaysClient.publishRegularPersonAge(
    "Joe Sixteen",
    16L
);
-> at com.example.PersonTest.testPublishAge(PersonTest.java:40)
```

This is expected, given that we haven't yet implemented the required changes to `Person.java`, since we're following test-driven development. We will now make this test pass by making the necessary changes:

Person.java

```
// ...
class Person {
    // ...
    private final BirthdaysClient birthdaysClient;

    Person(String givenName, String surname, LocalDate dateOfBirth) {
        this(givenName, surname, dateOfBirth, LocalDate::now, new BirthdaysClient());
    }

    // Visible for testing
    Person(String givenName, String surname, LocalDate dateOfBirth, Supplier<LocalDate> currentDateSupplier, BirthdaysClient birthdaysClient)
        // ...
        this.birthdaysClient = birthdaysClient;
    }

    // ...

    void publishAge() {
        String nameToPublish = givenName + " " + surname;
        long age = getAge();
        try {
            birthdaysClient.publishRegularPersonAge(nameToPublish, age);
        }
        catch (IOException e) {
            // TODO handle this!
            e.printStackTrace();
        }
    }
}
```

Testing for Exceptions

We made the production code constructor instantiate a new `BirthdaysClient`, and `publishAge()` now calls the `birthdaysClient`. All tests pass; everything is green. Great! But notice that `publishAge()` is swallowing the IOException. Instead of letting it bubble out, we want to wrap it with our own PersonException in a new file called `PersonException.java`:

PersonException.java

```
package com.example;

public class PersonException extends Exception {
    public PersonException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

We implement this scenario as a new test method in `PersonTest.java`:

PersonTest.java

```
// ...
class PersonTest {
    // ...

    @Test
    void testPublishAge_IOException() throws IOException {
        LocalDate dateOfBirth = LocalDate.parse("2000-01-02");
        LocalDate currentDate = LocalDate.parse("2017-01-01");

        Person person = new Person("Joe", "Sixteen", dateOfBirth, ()->currentDate, birthdaysClient);

        IOException ioException = new IOException();
        doThrow(ioException).when(birthdaysClient).publishRegularPersonAge("Joe Sixteen", 16);

        try {
            person.publishAge();
            fail("expected exception not thrown");
        }
        catch (PersonException e) {
            assertSame(ioException, e.getCause());
            assertEquals("Failed to publish Joe Sixteen age 16", e.getMessage());
        }
    }
}
```

The Mockito `doThrow()` call stubs `birthdaysClient` to throw an exception when the `publishRegularPersonAge()` method is called. If the `PersonException` is not thrown, we fail the test. Otherwise we assert that the exception was properly chained with the IOException and verify that the exception message is as expected. Right now, because we haven't implemented any handling in our production code, our test fails because the expected exception was not thrown. Here's what we need to change in `Person.java` to make the test pass:

Person.java

```
// ...
class Person {
    // ...

    void publishAge() throws PersonException {
        // ...
        try {
            // ...
        }
        catch (IOException e) {
            throw new PersonException("Failed to publish " + nameToPublish + " age " + age, e);
        }
    }
}
```

Stubs: Whens and Assertions

We now implement the `Person.getThoseInCommon()` method, making our `Person.Java` class look like this.

Our `testGetThoseInCommon()`, unlike `testPublishAge()`, does not verify that particular calls were made to `birthdaysClient` methods. Instead it uses `when` calls to stub return values for calls to `findFamousNamesOfAge()` and `findFamousNamesBornOn()` that `getThoseInCommon()` will need to make. We then assert that all three of the stubbed names we provided are returned.

Wrapping multiple assertions with the `assertAll()` JUnit 5 method allows all assertions to be checked as a whole, rather than stopping after the first failed assertion. We also include a message with `assertTrue()` to identify particular names that are not included. Here's what our "happy path" (an ideal scenario) test method looks like (note, this is not a robust set of tests by nature of being "happy path," but we'll talk about why later.

PersonTest.java

```
// ...
class PersonTest {
    // ...

    @Test
    void testGetThoseInCommon() throws IOException, PersonException {
        LocalDate dateOfBirth = LocalDate.parse("2000-01-02");
        LocalDate currentDate = LocalDate.parse("2017-01-01");
        Person person = new Person("Joe", "Sixteen", dateOfBirth, ()->currentDate, birthdaysClient);

        when(birthdaysClient.findFamousNamesOfAge(16)).thenReturn(Arrays.asList("JoeFamous Sixteen", "Another Person"));
        when(birthdaysClient.findFamousNamesBornOn(1, 2)).thenReturn(Arrays.asList("Jan TwoKnown"));

        Set<String> thoseInCommon = person.getThoseInCommon();

        assertAll(
                setContains(thoseInCommon, "Another Person"),
                setContains(thoseInCommon, "Jan TwoKnown"),
                setContains(thoseInCommon, "JoeFamous Sixteen"),
                ()-> assertEquals(3, thoseInCommon.size())
        );
    }

    private <T> Executable setContains(Set<T> set, T expected) {
        return () -> assertTrue(set.contains(expected), "Should contain " + expected);
    }

    // ...
}
```

Keep Test Code Clean

Though often overlooked, it's equally important to keep test code free of festering duplication. Clean code and principles like "don't repeat yourself" are very important for maintaining a high quality codebase, production and test code alike. Notice that the most recent PersonTest.java has some duplication now that we have several test methods.

To fix this, we can do a handful of things:

- Extract the IOException object into a private final field.
- Extract the `Person` object creation into its own method (`createJoeSixteenJan2()`, in this case) since most of the Person objects are being created with the same parameters.
- Create an `assertCauseAndMessage()` for the various tests that verify thrown `PersonExceptions`.

The clean code results can be seen in this rendition of the PersonTest.java file.

Test More Than the Happy Path

What should we do when a `Person` object has a date of birth that's later than the current date? Defects in applications are often due to unexpected input or lack of foresight into corner, edge, or boundary cases. It is important to try to anticipate these situations as best we can, and unit tests are often an appropriate place to do so. In building our `Person` and `PersonTest`, we included a few tests for expected exceptions, but it was by no means complete. For example, we use `LocalDate` which does not represent or store time zone data. Our calls to `LocalDate.now()`, however, return a `LocalDate` based on the system's default time zone, which could be a day earlier or later than that of the user of a system. These factors should be considered with appropriate tests and behavior implemented.

Boundaries should also be tested. Consider a `Person` object with a `getDaysUntilBirthday()` method. Testing should include whether or not the person's birthday has already passed in the current year, whether the person's birthday is today, and how a leap year affects the number of days. These scenarios can be covered by checking one day before the person's birthday, the day of, and one day after the person's birthday where the next year is a leap year. Here is the pertinent test code:

PersonTest.java

```
// ...
class PersonTest {
    private final Supplier<LocalDate> currentDateSupplier = ()-> LocalDate.parse("2015-05-02");
    private final LocalDate ageJustOver5 = LocalDate.parse("2010-05-01");
    private final LocalDate ageExactly5 = LocalDate.parse("2010-05-02");
    private final LocalDate ageAlmost5 = LocalDate.parse("2010-05-03");

    // ...

    @Test
    void testGetDaysUntilBirthday() {
        assertAll(
            createPersonAndAssertValue(ageAlmost5, 1, Person::getDaysUntilBirthday),
            createPersonAndAssertValue(ageExactly5, 0, Person::getDaysUntilBirthday),
            createPersonAndAssertValue(ageJustOver5, 365, Person::getDaysUntilBirthday)
        );
    }

    private Executable createPersonAndAssertValue(LocalDate dateOfBirth, long expectedValue, Function<Person, Long> personLongFunction) {
        Person person = new Person("Given", "Sur", dateOfBirth, currentDateSupplier);
        long actualValue = personLongFunction.apply(person);
        return () -> assertEquals(expectedValue, actualValue);
    }
}
```

Integration Tests

We have mostly focused on unit tests, but JUnit can also be used for integration, acceptance, functional, and system tests. Such tests often require more setup code, e.g., starting servers, loading databases with known data, etc. While we can often run thousands of unit tests in seconds, large integrations test suites might take minutes or even hours to run. Integration tests should generally not be used to try to cover every permutation or path through the code; unit tests are more appropriate for that.

Creating tests for web applications that drive web browsers in filling forms, clicking buttons, waiting for content to load, etc., is commonly done using Selenium WebDriver (Apache 2.0 license) coupled with the 'Page Object Pattern' (see the SeleniumHQ github wiki and Martin Fowler's article on Page Objects).

JUnit is effective for testing RESTful APIs with the use of an HTTP client like Apache HTTP Client or Spring Rest Template (HowToDoInJava.com provides a good example).

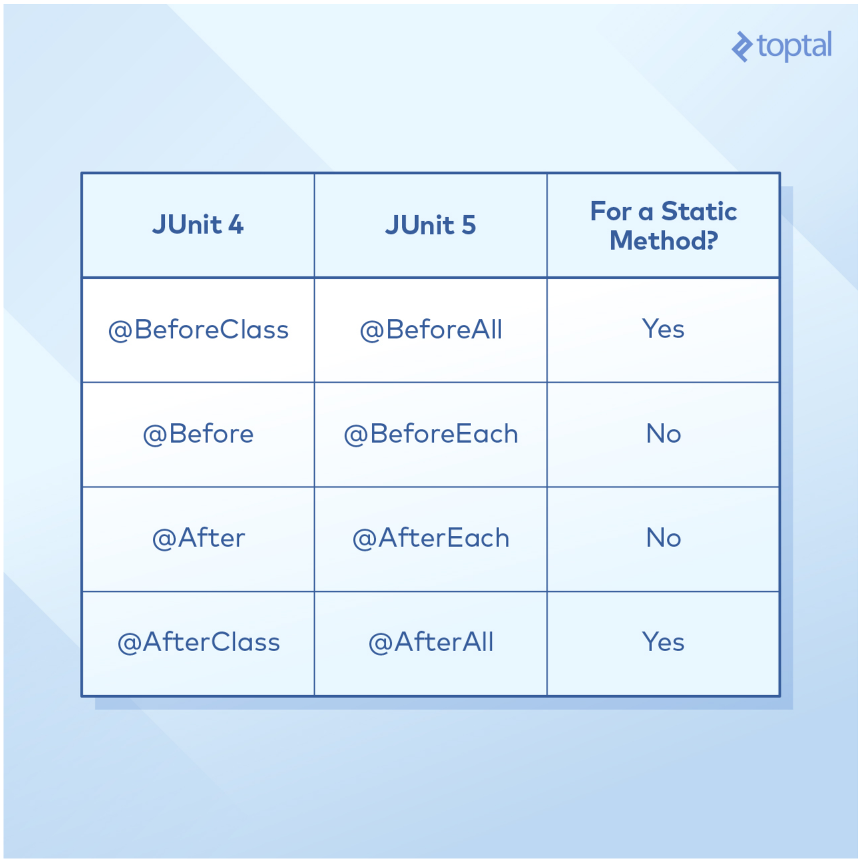
In our case with the `Person` object, an integration test could involve using the real `BirthdaysClient` rather than a mock one, with a configuration specifying the base URL of the People Birthdays service. An integration test would then use a test instance of such a service, verify that the birthdays were published to it and create famous people in the service that would be returned.

Other JUnit Features

JUnit has many additional features we have not yet explored in the examples. We will describe some and provide references for others.

Test Fixtures

It should be noted that JUnit creates a new instance of the test class for running each `@Test` method. JUnit also provides annotation hooks to run particular methods before or after all or each of the `@Test` methods. These hooks are often used for setting up or cleaning up database or mock objects, and differ between JUnit 4 and 5.



JUnit 4	JUnit 5	For a Static Method?
@BeforeClass	@BeforeAll	Yes
@Before	@BeforeEach	No
@After	@AfterEach	No
@AfterClass	@AfterAll	Yes

In our `PersonTest` example, we chose to configure the `BirthdaysClient` mock object in the `@Test` methods themselves, but sometimes more complex mock structures need to be built out involving multiple objects. `@BeforeEach` (in JUnit 5) and `@Before` (in JUnit 4) is often appropriate for this.

The `@After*` annotations are more common with integration tests than unit tests as the JVM garbage collection handles most objects created for unit tests. The `@BeforeClass` and `@BeforeAll` annotations are most commonly used for integration tests that need to perform costly setup and teardown actions once, rather than for each test method.

For JUnit 4, please refer to the test fixtures guide (the general concepts do still apply to JUnit 5).

Test Suites

Sometimes you want to run multiple related tests, but not all the tests. In this case, groupings of tests can be composed into test suites. For how to do this in JUnit 5, check out HowToProgram.xyz's JUnit 5 article, and in the JUnit team's documentation for JUnit 4.

JUnit 5's @Nested and @DisplayName

JUnit 5 adds the capability to use non-static nested inner classes to better show the relationship between tests. This should be very familiar to those who have worked with nested describes in test frameworks like Jasmine for JavaScript. The inner classes are annotated with `@Nested` to use this.

The `@DisplayName` annotation is also new to JUnit 5, allowing you to describe the test for reporting in string format, to be shown in addition to the test method identifier.

Although `@Nested` and `@DisplayName` can be used independently of each other, together they can provide for clearer test results that describe the behavior of the system.

Hamcrest Matchers

The Hamcrest framework, though not itself part of the JUnit codebase, provides an alternative to using traditional assert methods in tests, allowing for more expressive and readable test code. See the following verification using both a traditional assertEquals and a Hamcrest assertThat:

```
//Traditional assert
assertEquals("Hayden, Josh", displayName);

//Hamcrest assert
assertThat(displayName, equalTo("Hayden, Josh"));
```

Hamcrest can be used with both JUnit 4 and 5. Vogella.com's tutorial on Hamcrest is quite comprehensive.

Additional Resources

- The article Unit Tests, How to Write Testable Code and Why it Matters covers more specific examples of writing clean, testable code.
- The JUnit 4 Wiki and JUnit 5 User Guide are always an excellent reference point.
- The Mockito documentation provides information on additional functionality and examples.