Object-Oriented Software Development

JUnit 3.8

An Automated Java Unit Testing Tool

JUnit

JUnit:

JUnit is an open source testing framework for Java. It provides a very simple way to express the way you intend your code to work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions.

JUnit is a program used to perform unit testing of virtually any software. JUnit testing is accomplished by writing test cases using Java, compiling these test cases and running the resultant classes with a JUnit Test Runner.

Unit Testing:

A unit test case is a collection of tests designed to verify the behavior of a single unit within your program. In Java, the single unit is almost always a class. A Java unit test case, then, tests a single class.

JUnit:

Unit Testing (cont.):

Most programmers do not write tests.

We all know that we should write them, but for whatever reason, most of us don't. This is unfortunate because testing is the most powerful tool we know of to improve software quality.

Tests reduce bugs, provide accurate documentation, and improve design.

Eric M. Burke and Brian M. Coyner, authors of Java Extreme Programming Cookbook, list the following as the top 12 reasons to write unit tests:

1. **Tests Reduce Bugs in New Features**

   We advocate writing tests as you write new code. Tests do not eliminate bugs, but they dramatically reduce the number of bugs as you add new features.

2. **Tests Reduce Bugs in Existing Features**

   With well-tested code, introducing new features rarely breaks existing functionality. If a new feature breaks existing functionality, existing tests fail immediately, allowing you to pinpoint the problem and fix it. Without the tests, you may introduce a bug that is not found for days or weeks.

JUnit:

Unit Testing (cont.):

3. **Tests Are Good Documentation**

   A concise code example is better than many paragraphs of documentation. We see this time after time in our consulting work. Far too often, teams produce boilerplate documents that are of little practical value. When programmers need to learn an API, they search for code examples. Tests are among the best code examples because they are concise snippets of code that exercise public APIs.

4. **Tests Reduce the Cost of Change**

   Tests make it easier to change software because you have confidence that changes do not break existing functionality. When you have good test coverage, you have confidence to explore new design ideas without fear of introducing new bugs.

   Poorly-tested software becomes increasingly expensive to change as time goes on. Risk increases dramatically as the system becomes more complex because it becomes more and more likely that changes inadvertently break things that used to work.

5. **Tests Improve Design**

   Writing tests forces you to make your code testable. Specifically, you tend to rely less on dubious patterns like singletons and global variables, instead making your classes loosely-coupled and easier to use. Code that is tightly-coupled or requires complex initialization is hard to test

JUnit:

Unit Testing (cont.):

6. **Tests Allow Refactoring**

   With tests, you are more able to change code throughout the lifetime of an application. Tests provide a safety net, allowing you to refactor at any time without fear of breaking existing code, so you can constantly improve the design of your program.

7. **Tests Constrain Features**

   Far too often, programmers build fancy frameworks rather than deliver features customers want. When you adopt a test-first approach, you start by writing tests for the current feature. You then implement the feature. When the tests pass, you know you can stop and move to the next feature. Well-tested applications are more easily extended; therefore, you don't have to anticipate what the customer will eventually request.

8. **Tests Defend Against Other Programmers**

   Textbook code is simple, but real-world problems are hard. We find that in real applications, you often encounter very subtle bugs due to Java bugs, quirky business rules, operating system differences, etc. These bugs may only manifest themselves under very peculiar scenarios.

   Let's suppose you find that a payroll calculation routine removes a zero from everyone's salary, but only if the routine runs at 11:59 PM on New Year's Eve. Now, suppose that the bug fix involves a single-line code change.

   Without a test, another programmer may come in and change that code. Unless they run the application at 11:59 PM on New Year's Eve, they won't know that they just re-introduced the bug and will cause countless bounced checks next year. With a test, however, you can ensure that when the programmer changes the code, the test breaks and informs the programmer of the problem.

JUnit:

Unit Testing (cont.):

9. Testing Is Fun

If you thrive on challenges, then testing is a lot of fun. Coming up with automated tests is difficult, requiring creative solutions for complex problems. Just like coding is an art, testing is an art.

In many organizations, testing is relegated to the least-experienced programmers. We often encounter the misconception that testing consists of people completing written checklists as they manually execute the application. This approach is completely unscalable, because it takes longer and longer for humans (monkeys?) to test every feature as the application grows.

Modern OO languages like Java are complex, particularly when it comes to dependencies between classes. One change can easily introduce bugs in seemingly unrelated classes. Gone are the days when each character-based screen is a standalone program. OO apps are far more complex and demand automated tests.

Writing automated tests is harder than writing the code itself, in many cases. The most expert programmers are the best testers. When faced with seemingly mundane coding tasks, coming up with creative tests provides an intellectual challenge that expert programmers thrive on.

Beginners typically need expert assistance when writing tests. This is where pair-programming helps, because experts work side-by-side with beginners as they learn the art of testing.

10. Testing Forces You to Slow Down and Think

When adding a new feature or refactoring an existing solution, testing forces you to think about what the code is supposed to accomplish. By writing tests first, you think about how to use the public API and what the ultimate outcome should be. Thus you end up with a clean and simple design that does exactly what you expect it to do.

JUnit:

Unit Testing (cont.):

11. **Testing Makes Development Faster**

    On a class-by-class basis, testing slows you down. It takes time to think about and produce good tests. But as time goes on, your overall velocity increases because you are not constantly worrying about breaking existing code as you add new features.

    We have also found that with good tests, we can ignore internal implementation details during the first iteration. Provided that we get the public API right, we can improve internal design and algorithms later, again without fear of breaking something. We have used this specifically for things like sorting algorithms. In the first iteration, we do something quick and dirty, like a bubble sort. We can always come back later and replace it with a better algorithm, if necessary.

12. **Tests Reduce Fear**

    One of the biggest fears that programmers encounter is making a change to a piece of code and not knowing what is going to break. Having a complete test suite allows programmers to remove the fear of making changes or adding new features. We have found that we do not hesitate to change and improve well-tested code, whereas we fear changing untested code.

JUnit:

Unit Testing:

The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules.

Unit testing differs from *integration testing*, which confirms that components work well together, and *acceptance testing*, which confirms that an application does what the customer expects it to do. Unit tests are so named because they test a single unit of code. In the case of Java, a unit usually equates to a single class.

Unit testing provides four main benefits:

1. **Encourages change**
   Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (regression testing). This provides the benefit of encouraging programmers to make changes to the code since it is easy for the programmer to check if the piece is still working properly.

2. **Encourages change**
   Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (regression testing). This provides the benefit of encouraging programmers to make changes to the code since it is easy for the programmer to check if the piece is still working properly.

JUnit:

Unit Testing:

Unit testing provides four main benefits (cont.):

3. **Simplifies Integration**
   Unit testing helps eliminate uncertainty in the pieces themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts will make integration testing easier.

4. **Documents the code**
   Unit testing provides a sort of "living document" for the class being tested. Clients looking to learn how to use the class can look at the unit tests to determine how to use the class to fit their needs.

5. **Separation of Interface from Implementation**
   Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is a class that depends on a database. In order to test the class, the tester writes code that interacts with the database. This is a mistake, because a unit test should never go outside of its own class boundary. As a result, the software developer abstracts an interface around the database connection, and then implements that interface with a Mock Object. This results in loosely coupled code, thus minimizing dependencies in the system.

JUnit:

Unit Testing Limitations

It is important to realize that unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, performance problems and any other system-wide issues. In addition, it may not be trivial to anticipate all special cases of input the program unit under study may receive in reality. Unit testing is only effective if it is used in conjunction with other software testing activities.

Benefits of an Automated Unit Test Suite

An automated unit test suite brings along a number of important, tangible advantages as compared to other testing strategies:

1. **Unit tests find problems early in the development cycle.**
   The common wisdom in software development is that the earlier problems are found, the cheaper it is to fix them. An automated unit test suite finds problems effectively as early as possible, long before the software reaches a customer, and even before it reaches the QA team. Most of the problems in new code are already uncovered before the developer checks the code into source control.

JUnit:

Benefits of an Automated Unit Test Suite (cont.)

2. **An automated unit test suite watches over your code in two dimensions: time and space**.
   It watches over your code in the time dimension because once you’ve written a unit test, it guarantees that the code you wrote works now and in the future. It watches over your code in the space dimension because unit tests written for other features guarantee that new code did not break them; likewise it guarantees that code written for other features does not adversely affect the code you wrote for this feature.

3. **Developers will be less afraid to change existing code.**
   Over time, software systems become more and more change resistant because developers are reluctant to change old code. This is natural because when changing old code, there is always the risk of breaking it or some other part of the system through a side effect.

4. **The development process becomes more flexible.**
   Sometimes it may be necessary to fix a problem and to deploy the fix quickly. Despite best efforts, a bug may slip in and an important feature may stop working. The customers cannot purchase products, the users cannot work and your boss is breathing over your shoulder asking you to fix the problem immediately. Releasing quick fixes makes us feel uneasy because we are not certain what side effects the changes might have. Running the unit tests with the fixes applied saves the day, as they should reveal undesirable side effects. Publishing hotfixes is something we hope we never have to do, and a unit test suite should already decrease the need for such things anyway. But if you ever have to publish a hotfix, a unit test suite improves your chances of doing so without introducing new problems.

JUnit:

Benefits of an Automated Unit Test Suite (cont.)

5. **An automated unit test suite reduces the need for manual testing.**
   Some manual testing will always be needed because humans excel at discovering bugs that involve complex data and workflow processes. Writing a unit test for the most complex cases might be so prohibitively time consuming that it is not cost effective any more. The QA team can concentrate discovering the hard-to-find bugs while the unit tests do most of the mundane testing.

The net effect of the benefits listed above is that ***software development will become more predictable and repeatable*** – in a word, a bit more like a real engineering discipline. The design and coding phases still have a fair amount of ‘art’ in them, and this will not go away. Once the coding is done, the build process builds and tests the software much like physical products are built on an assembly line. This removes much of the ad-hoc nature in software development, which is the underlying reason for many of the problems that plague software projects.

JUnit:

JUnit Test Development Process

Before examining the JUnit testing process, we will start by examining mine the following class. This simple class, part of a small Vehicle Rental application, represents a rental “Customer”. The class contains a String first and last name, and an integer that will hold a count of the number of vehicles this customer has rented. We will then use JUnit to build regression tests for this class.

Customer:

```
package examples;

public class Customer
{
   private String name;
   private int numRentals;

   public Customer(String nameIn, int numRentalsIn) throws Exception
   {
      setName(nameIn);
      setNumRentals(numRentalsIn);
   }


   private String getName()
   {
      return name;
   }
```

JUnit

JUnit:

Customer (cont.):

```
   private void setName(String nameIn) throws Exception
   {
      if (nameIn.length() == 0)
            throw new Exception("Empty Customer Name");
      name = nameIn;
   }


   private int getNumRentals()
   {
      return numRentals;
   }


   private void setNumRentals(int numRentalsIn) throws Exception
   {
      if (numRentalsIn < 0)
            throw new Exception("Negative Number of Customer Rentals: " + numRentalsIn);

      numRentals = numRentalsIn;
   }

}
```

JUnit

JUnit:

JUnit Test Development Process

We will now use JUnit to build regression tests for this class.

To do this, we will create a "static" inner class (we'll call it "CustomerTest") *within* the Customer class that extends the JUnit "TestCase" class.

The "static" inner class "CustomerTest" should have the following features:

a) Should extend the JUnit "TestCase" class.
b) Generally has a data attribute of it's associated class type (i.e., "CustomerTest" has a "Customer" data attribute)
c) Contains constructor that accepts a String (used by JUnit) that must be passed to a call to the parent (TestCase) constructor. (i.e., super(…))
d) Should override the "TestCase" protected void method "setUp()". This method is called before each of your "testXxxx()" methods to initialize the testing environment.
e) Should override the "TestCase" protected void method "tearDown ()". This method is called after each of your "testXxxx()" methods to release any permanent resources you allocated in setup method.
f) Should define a public static method "suite()" that returns a "Test" object. This is provided so that the class' test suite can be included within a larger test suite.
g) Contains a set of public void "testXxxx()" methods that contain your tests. These methods test specific conditions using the JUnit "assertXxxxx(…)" methods.

JUnit:

JUnit Assert Class

JUnit uses static void “assertion” methods to determine success or failure of a conditional statement.
Errors are unanticipated problems that generate java Exceptions. Failures are signaled with an AssertionFailedError error.

These assertions are called within the “testXxxx()” methods of your inner static class.
Should one of these assertions fail, an AssertionFailedError will be thrown. JUnit will interpret this AssertionFailedError as a test method failure and will record the details of the failure.

Assertion Types

Examples:
```
assertEquals(expectedSpeed, realSpeed);
assertEquals(“Vehicle speed values not Equal”, expectedSpeed, realSpeed);
```

Equality:

- **assertEquals(type expected, type actual)** Asserts that two values are equal. (boolean/byte/char/short/int/long/String/Object)
- **assertEquals(type expected, type actual, type delta)** Asserts that two values are equal or within a delta. (double/float)
- **assertEquals(String errorMessage, type expected, type actual)** Asserts that two values are equal. If not it throws an AssertionFailedError with the message provided. (boolean/byte/char/short/int/long/String/Object)
- **assertEquals(String errorMessage, type expected, type actual, type delta)** Asserts that two values are equal or within a delta. If not it throws an AssertionFailedError with the message provided. (double/float)

JUnit:

JUnit Assert Class – Assertion Types (cont.)

Examples:

```
assertTrue(“Test Vehicle is Null”, (myVehicle != null));
assertFalse(“Test Vehicle is Null”, (myVehicle == null));
assertNull(customer);
assertNotNull(“Customer is NOT null”, customer);
```

Conditional True/False:

- **assertTrue(boolean condition)** Asserts that a condition is true.
- **assertTrue(String errorMessage, boolean condition)** Asserts that a condition is true. If it is false, it throws an AssertionFailedError with the message provided.
- **assertFalse(boolean condition)** Asserts that a condition is false.
- **assertFalse(String errorMessage, boolean condition)** Asserts that a condition is false. If it is true, it throws an AssertionFailedError with the message provided.

Null/Not Null:

- **assertNull(Object object)** Asserts that an object is null.
- **assertNull(String errorMessage, Object object)** Asserts that an object is null. If it is null, it throws an AssertionFailedError with the message provided.
- **assertNotNull(Object object)** Asserts that an object isn't null.
- **assertNotNull(String errorMessage, Object object)** Asserts that an object isn't null. If it is null, it throws an AssertionFailedError with the message provided.

JUnit:

JUnit Assert Class – Assertion Types (cont.)

Examples:

```
assertSame(“Vehicles are NOT the same”, vehicle1, vehicle2);
assertNotSame(vehicle1, vehicle2);
fail();
fail(“This causes a test method failure”);
```

Same/Not Same:

- **assertSame(Object expected, Object actual)** Asserts that two references refer to the same object.
- **assertSame(String errorMessage, Object expected, Object actual)** Asserts that two references refer to the same object. If they do not, it throws an AssertionFailedError with the message provided.
- **assertNotSame(Object expected, Object actual)** Asserts that two references do not refer to the same object.
- **assertNotSame(String errorMessage, Object expected, Object actual)** Asserts that two references do not refer to the same object. If they do, it throws an AssertionFailedError with the message provided.

Failure:

- **fail()** Fails a test with no message.
- **fail(String errorMessage)** Fails a test with the given message.

JUnit:

Customer with Embedded "static" Inner Class

```
   […]

   private void setNumRentals(int numRentalsIn) throws Exception
   {
      if (numRentalsIn < 0)
            throw new Exception("Negative Number of Customer Rentals: " + numRentalsIn);

      numRentals = numRentalsIn;
   }

// This is a static inner class.  It extends JUnit's TestCase class.

   public static class CustomerTest extends TestCase   a
   {

      // This data attribute will hold an instance of Customer
 b    private Customer customer = null;


      // Basic Test Class Constructor
      public CustomerTest(String name) throws Exception
      {
 c          super(name); // Call the parent constructor passing the String parameter
      }

      […]
```

JUnit:

Customer with Embedded “static” Inner Class (cont.):

```
[…]
// Override the TestCase class' implementation of "setUp()" (which does nothing)
// to set up any data values and attrinutes we want to have set when our tests run.

d   protected void setUp() throws Exception
    {
          System.out.println("Setup Customer");
          customer = new Customer("John Smith", 0);
    }



// Override the TestCase class' implementation of "tearDown()"(which does nothing)
// to clean up when our tests are complete.

e   protected void tearDown()
    {
          System.out.println("TearDown Customer");
          customer = null;
    }



// The "suite()" method returns the Customer class’ suite of tests for inclusion
// within a larger test suite.

f   public static Test suite()
    {
          return new TestSuite(CustomerTest.class);
    }
[…]
```

JUnit

JUnit:

Customer with Embedded "static" Inner Class (cont.):

```
[…]

// JUnit will run all test methods that begin with "test...()". This Test method
// tests the get & set methods for our 3 Customer data attributes.
//
// The JUnit "assertTrue(message, condition)" method asserts that a condition is true.
// If it isn't true, it throws an AssertionFailedError with the message provided.


g  public void testCustomerGetSetMethods() throws Exception
   {
         System.out.println("Test Customer Get/Set Methods");

         customer.setName("Jane Doe");
         customer.setNumRentals(3);

         assertTrue("Value: " + customer.getName(),
                    customer.getName().equals("Jane Doe"));

         assertTrue("Value: " + customer.getNumRentals(),
                    customer.getNumRentals() == 3);
   }

[…]
```

JUnit

JUnit:

Customer with Embedded “static” Inner Class (cont.):

g

```
        […]
        // This Test method tests our set method error handling.
        // The JUnit "assertTrue(condition)" method asserts that a condition is true.
        // If it isn't true, it throws an AssertionFailedError.

        public void testCustomerSetMethodErrorHandling()
        {
              System.out.println("Test Customer Set Method Error Handling");

              try
              {
                  // Try to make the "set" method fail and throw an Exception
                  customer.setName("");
                  fail("Allowed empty username!");
              }
              catch (Exception e) // Enter here if the error was detected and thrown
              {
                  System.out.println("OK! Found: " + e.getMessage());
              }
              try
              {
                  // Try to make the "set" method fail and throw an Exception
                  customer.setNumRentals(-2);
                  fail("Allowed negative numRental value");
              }
              catch (Exception e2) // Enter here if the error was detected and thrown
              {
                  // If we get here, all expected Exceptions were properly thrown
                  System.out.println("OK! Found: " + e2.getMessage());
                  return;
              }
        }
     } // End of CustomerTest class
} // End of Customer class
```

JUnit:

JUnit Test Development Process (cont.)

The next class, also part of a Vehicle Rental application, represents a Rental Vehicle. This class is a little larger than the Customer class, and contains a functional method as well as the expected accessor & modifiers. The class contains a String vehicle id, a daily rental rate, a new vehicle flag, and a reference to a Customer object. This class *already* has the static inner test class in place.

RentalVehicle:

```
package examples;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class RentalVehicle
{
      // Identifier for this Vehicle
      private String vehicleId;

      // Daily rental rate in dollars & cents
      private double dailyRentalRate;

      // Flag indicating if this is considered a "new" Vehicle
      private boolean newVehicle;

      // Current Customer
      private Customer customer = null;

      […]
```

JUnit

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

```
[…]

// Constructor accepts parameters for each attribute and uses
// them to set up the attribute initial values
public RentalVehicle(String idIn, double rateIn, boolean newIn, Customer customerIn) throws Exception
{
      setVehicleId(idIn);
      setDailyRentalRate(rateIn);
      setNewVehicle(newIn);
      setCustomer(customerIn);
}

private String getVehicleId()
{
      return vehicleId;
}

private void setVehicleId(String vehicleIdIn) throws Exception
{
      if (vehicleIdIn.length() == 0)
            throw new Exception("Empty Vehicle Id");
      vehicleId = vehicleIdIn;
}

private double getDailyRentalRate()
{
      return dailyRentalRate;
}

[…]
```

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

```
    […]

    private void setDailyRentalRate(double dailyRentalRateIn) throws Exception
    {
          if (dailyRentalRateIn < 0.0)
                throw new Exception("Negative Daily Rental Rate: " + dailyRentalRateIn);
          dailyRentalRate = dailyRentalRateIn;
    }

    private boolean getNewVehicle()
    {
          return newVehicle;
    }

    private void setNewVehicle(boolean newVehicleIn)
    {
          newVehicle = newVehicleIn;
    }

    private Customer getCustomer()
    {
          return customer;
    }

    private void setCustomer(Customer customerIn) throws Exception
    {
          if (customerIn == null)
                throw new Exception("Attempt to Set a Null Customer");
          customer = customerIn;
    }

    […]
```

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

```
[…]

// Calculates the rental cost based on a numver of days.
public double calculateRentalCost(int numDays)
{
      return getDailyRentalRate() * numDays;
}

public void print()
{
      System.out.println(this);
}

public String toString()
{
      return new String("Vehicle:\t\t\t" + getVehicleId() + "\n" +
                                "Daily Rental Rate:\t" + getDailyRentalRate() + "\n" +
                                "New Vehicle:\t\t" + getNewVehicle());
}

//
// This is a static inner class.  It extends JUnit's TestCase class.
//
public static class RentalVehicleTest extends TestCase  a
{
 b    private RentalVehicle rentalVehicle = null;

      private Customer customer = null;

[…]
```

JUnit:

RentalVehicle with embedded "static" inner class (cont.):

```
[…]

// Call the parent constructor passing the String parameter
public RentalVehicleTest(String name) throws Exception
{
      super(name);
}


// OverRide the TestCase class' implementation of "setUp()"
// to set up any data values we want to have set when our tests run.
protected void setUp() throws Exception
{
      System.out.println("Setup Rental Vehicle");
      customer = new Customer("Abe Lincoln", 5);

      rentalVehicle = new RentalVehicle("NONE", 0.00, false, customer);
}


// OverRide the TestCase class' implementation of "tearDown()"
// to set up clean up when our tests are complete.
protected void tearDown()
{
      System.out.println("TearDown Rental Vehicle");
      rentalVehicle = null;
}

[…]
```

c

d

e

JUnit

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

```
        […]

        // The "suite()" method returns the RentalVehicle class’ suite of tests for inclusion
        // within a larger test suite.
  f     public static Test suite()
        {
              return new TestSuite(RentalVehicleTest.class);
        }

        // JUnit will run all test methods that begin with "test...()"
        // This Test method tests our get & set methods.
  g     public void testRentalVehicleGetSetMethods() throws Exception
        {
              System.out.println("Test Rental Vehicle Get/Set Methods");

              rentalVehicle.setVehicleId("A1B2C3");
              rentalVehicle.setDailyRentalRate(19.99);
              rentalVehicle.setNewVehicle(true);
              rentalVehicle.setCustomer(customer);

              assertTrue("Value: " + rentalVehicle.getVehicleId(),
                             rentalVehicle.getVehicleId().equals("A1B2C3"));

              assertTrue("Value: " + rentalVehicle.getDailyRentalRate(),
                             rentalVehicle.getDailyRentalRate() == 19.99);

              assertTrue("Value: " + rentalVehicle.getNewVehicle(),
                             rentalVehicle.getNewVehicle() == true);

              assertNotNull(rentalVehicle.getCustomer());
        }
        […]
```

JUnit

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

g

```
[…]
// JUnit will run all test methods that begin with "test...()"
// This Test method tests our set method error handling.
public void testRentalVehicleSetMethodErrorHandling()
{
      System.out.println("Test Vehicle Set Method Error Handling");

      try {
           rentalVehicle.setVehicleId("");
           fail("Allowed empty vehicleId");
      }
      catch (Exception e) {
           System.out.println("OK! Found: " + e.getMessage());
      }

      try {
           rentalVehicle.setDailyRentalRate(-1.00);
           fail("Allowed negative dailyRentalRate");
      }
      catch (Exception e1) {
           System.out.println("OK! Found: " + e1.getMessage());
      }

      try {
           rentalVehicle.setCustomer(null);
           fail("Allowed null customer");
      }
      catch (Exception e2) {
           System.out.println("OK! Found: " + e2.getMessage());
      }
}
[…]
```

JUnit

JUnit:

RentalVehicle with embedded “static” inner class (cont.):

g

```
            […]

            // JUnit will run all test methods that begin with "test...()"
            // This Test method tests our rental calculations.

            public void testRentalVehicleCalculations() throws Exception
            {
                  System.out.println("Test Rental Vehicle Calculations");

                  rentalVehicle.setVehicleId("A1B2C3");
                  rentalVehicle.setDailyRentalRate(19.99);
                  rentalVehicle.setNewVehicle(true);

                  // Note to avoid "double" calculation precision problems, we use the
                  // "assertEquals" method that takes a "delta" (acceptable difference).

                  assertEquals("Value: " + rentalVehicle.calculateRentalCost(1),
                               rentalVehicle.calculateRentalCost(1), 19.99, 0.0001);

                  assertEquals("Value: " + rentalVehicle.calculateRentalCost(5),
                               rentalVehicle.calculateRentalCost(5), 99.95, 0.0001);
            }
      } // End of RentalVehicleTest class
} // End of RentalVehicle class
```

JUnit:

JUnit Test Development Process (cont.)

We now have created 2 functional classes with unit tests created for use by JUnit. These classes will compile just like any other class. The static inner test class will not affect the class that it is designed to test. Unless used with JUnit for testing, these static inner test classes will remain idle.

We can now write a “main” method for each of these classes to execute their “testXxxx()” methods. These “main” methods look like this:

```
public static void main(String[] args)
{
   junit.swingui.TestRunner.run(CustomerTest.class);
}
```

```
public static void main(String[] args)
{
   junit.swingui.TestRunner.run(RentalVehicleTest.class);
}
```

The ‘junit.swingui.TestRunner” class is the “test runner”. The TestRunner class runs the test suite of the specified class.

To use the textual test runner, use:

junit.**textui**.TestRunner.run(YourTestClass.class);

To use the graphical test runner, use:

junit.**swingui**.TestRunner.run(YourTestClass.class);

JUnit:

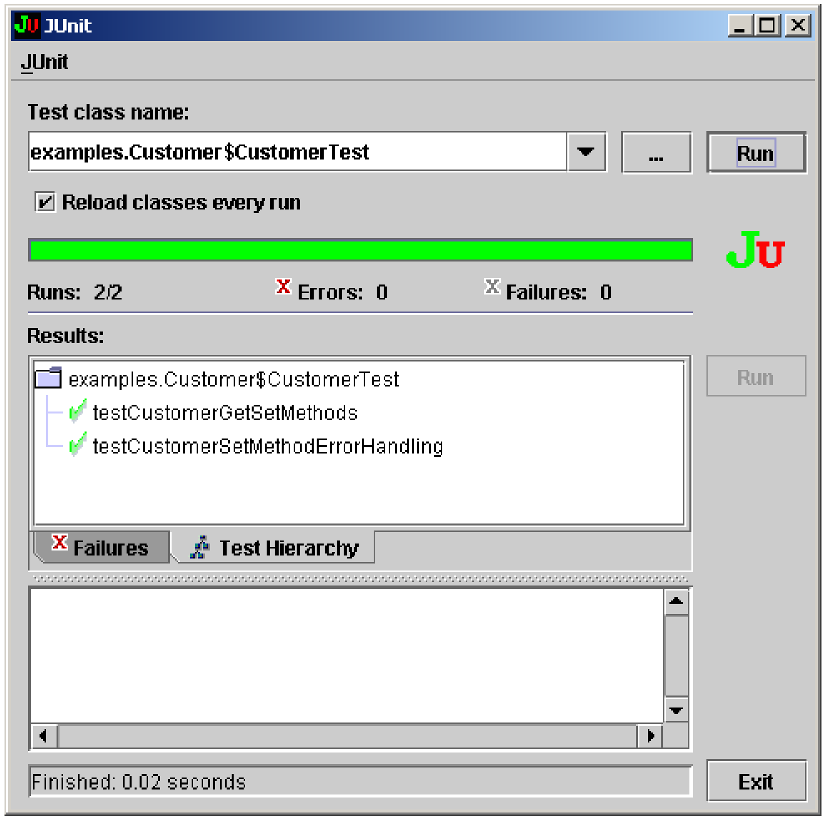
JUnit Test Development Process (cont.)

Your tests will indicate that they have passed by displaying "OK" (when using the text-based test runner) or a green bar indicating successful execution (when using the graphical runner):

Text-Based Test Runner

```
$ java junit.textui.TestRunner examples.Customer

.Setup Customer
Test Customer Get/Set Methods
TearDown Customer
.Setup Customer
Test Customer Set Method Error Handling
TearDown Customer

Time: 0

OK (2 tests)

$
```

Java Swing GUI Graphical Test Runner



Test class name: examples.Customer$CustomerTest

Reload classes every run ✓

Runs: 2/2 Errors: 0 Failures: 0

Results:

```
examples.Customer$CustomerTest
 ✓ testCustomerGetSetMethods
 ✓ testCustomerSetMethodErrorHandling
```

Finished: 0.02 seconds

JUnit:

JUnit Test Development Process (cont.)

Test Suites

As the number of classes grow in a project, it becomes more convenient to execute all the test methods in your test classes at once, rather than individually one class at a time. To do this, we will create a new class that builds and executes a test suite made up of the test suites of your other test classes.

Each of the classes that we added a static inner test classes has a “suite()” method that returns a “TestSuite” object. This object represents the test suites of that class. TestSuite objects can also be used to contain a collection test suites from other classes.

A common technique is to create a new class (we’ll call our new class “AllTests”) designed to contain the test suites of our other test classes. In this way, we can have JUnit run the combined test suite (in the “AllTests” class) which will in turn result in the execution of all test suites associated with it.

JUnit

JUnit:

JUnit Test Development Process (cont.)

Test Suites

Our “AllTest” class, designed to contain the test suites of our other test classes, is designed as follows:

```
package examples;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests
{
      // This method returns a TestSuite object that contains the test suites
      // of our Customer and RentalVehicle classes.
      public static Test suite()
      {
            TestSuite suite = new TestSuite();
            suite.addTest(Customer.CustomerTest.suite()); // Adds the Customer’s test suite
            suite.addTest(RentalVehicle.RentalVehicleTest.suite()); // Adds the RentalVehicle’s test suite
            return suite;
      }

      // Running this “main” method will execute the test suite containing all the other
      // class’ test suites.
      public static void main(String[] args)
      {
            junit.swingui.TestRunner.run(AllTests.class); // Use the graphical test runner
      }
}
```

JUnit:

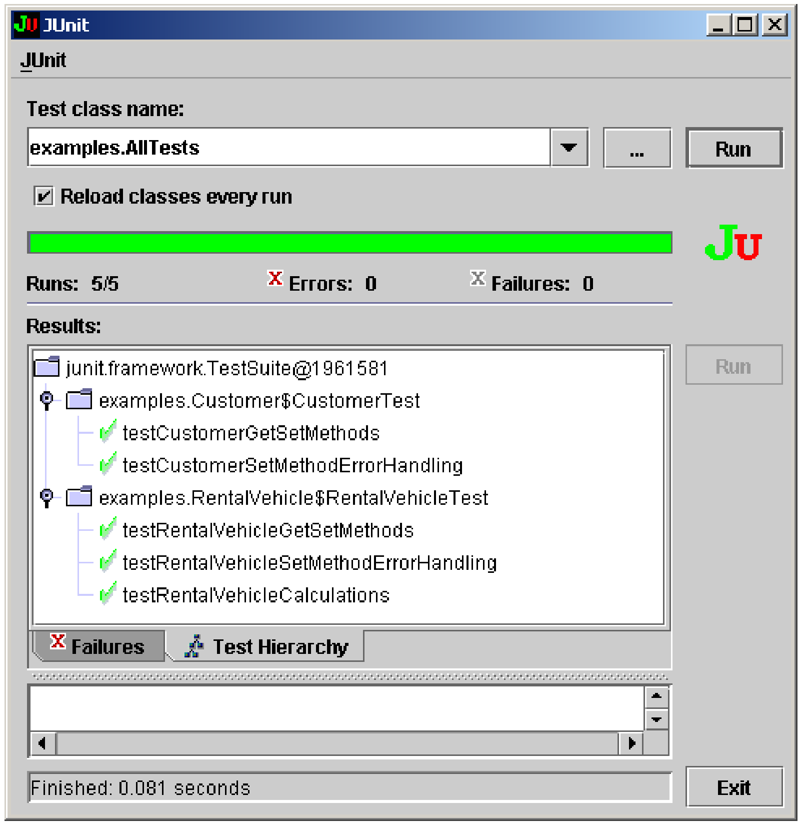
JUnit Test Development Process (cont.)

When run, our “AllTests” class would produce the following results displays, depending upon the test runner used:

Text-Based Test Runner

```
$ java junit.textui.TestRunner examples.AllTests

.Setup Customer
Test Customer Get/Set Methods
TearDown Customer
.Setup Customer
Test Customer Set Method Error Handling
TearDown Customer
.Setup Rental Vehicle
Test Rental Vehicle Get/Set Methods
TearDown Rental Vehicle
.Setup Rental Vehicle
Test Vehicle Set Method Error Handling
TearDown Rental Vehicle
.Setup Rental Vehicle
Test Rental Vehicle Calculations
TearDown Rental Vehicle
Time: 0

OK (5 tests)

$
```

Java Swing GUI Graphical Test Runner



Test class name: examples.AllTests

☑ Reload classes every run

Runs: 5/5 Errors: 0 Failures: 0

Results:

```
junit.framework.TestSuite@1961581
  examples.Customer$CustomerTest
    testCustomerGetSetMethods
    testCustomerSetMethodErrorHandling
  examples.RentalVehicle$RentalVehicleTest
    testRentalVehicleGetSetMethods
    testRentalVehicleSetMethodErrorHandling
    testRentalVehicleCalculations
```

Finished: 0.081 seconds

JUnit:

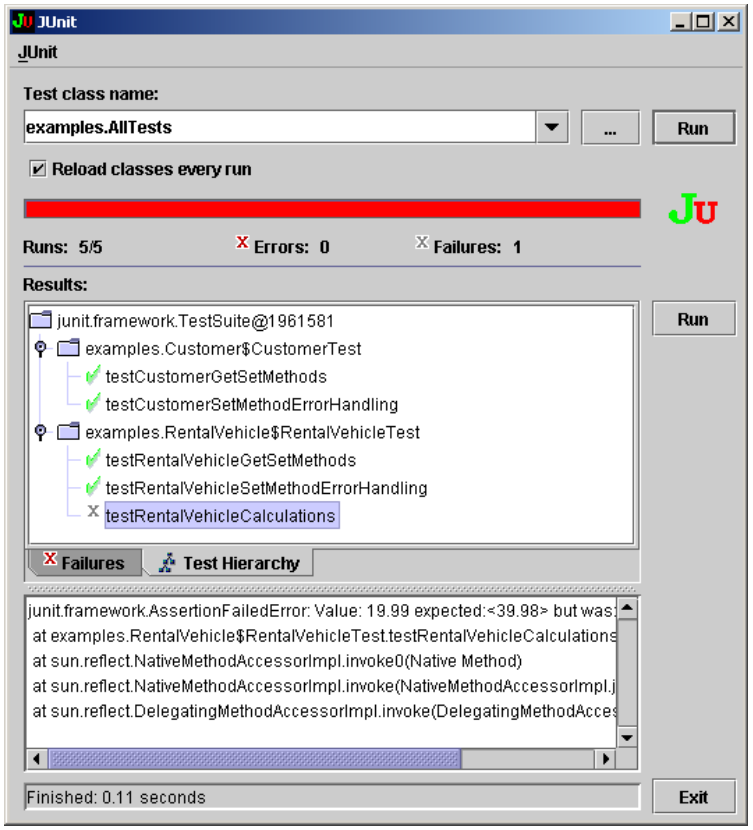
JUnit Test Development Process (cont.)

If we had failures in our test execution, our “AllTests” class would produce the following results displays, depending upon the test runner used:

Text-Based Test Runner

```
$ java junit.textui.TestRunner examples.Customer

.Setup Customer
[…]
TearDown Rental Vehicle


Time: 0.12
There was 1 failure:
1) testIsoDate(IsoDateTest)junit.framework
.AssertionFailedError: This is a test expected:<1> but
was:<2>
      at IsoDateTest.testIsoDate
      (IsoDateTest.java:29)

FAILURES!!!
Tests run: 5,  Failures: 1,  Errors: 0

OK (5 tests)

$
```

Java Swing GUI Graphical Test Runner



```
JUnit
Test class name:
examples.AllTests
[✓] Reload classes every run
Runs: 5/5    Errors: 0    Failures: 1
Results:
junit.framework.TestSuite@1961581
  examples.Customer$CustomerTest
    ✓ testCustomerGetSetMethods
    ✓ testCustomerSetMethodErrorHandling
  examples.RentalVehicle$RentalVehicleTest
    ✓ testRentalVehicleGetSetMethods
    ✓ testRentalVehicleSetMethodErrorHandling
    X testRentalVehicleCalculations
Failures | Test Hierarchy
junit.framework.AssertionFailedError: Value: 19.99 expected:<39.98> but was:
 at examples.RentalVehicle$RentalVehicleTest.testRentalVehicleCalculations
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.j
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
Finished: 0.11 seconds
```

JUnit

JUnit:

Compiling & Running a NetBeans Project with JUnit

The following describes how to incorporate JUnit into your NetBeans projects. The process is a variant of the basic NetBeans process.

Before starting NetBeans, create a directory (folder) in which you plan to store your Java files & other project-related files. It is a good idea to do this for each project (i.e., program) you work on.

Example:

“MyProjectDir” folder on the C: drive:

“C:/MyProjectDir”

Alternatively, you can create a project folder anywhere you want on your system.

The NetBeans IDE (Startup & Project Creation)

Creating/Using/Running a "Project" in the NetBeans IDE

Before starting NetBeans, create a directory (folder) in which you plan to store your Java files & other project-related files. It is a good idea to do this for *each* project (i.e., program) you work on.

Example:

"MyProjectDir" folder on the C: drive:

"C:/MyProjectDir"

Alternatively, you can create a project folder anywhere you want on your system.

To start the NetBeans IDE, either double-click the desktop icon (if one was created)

or select:

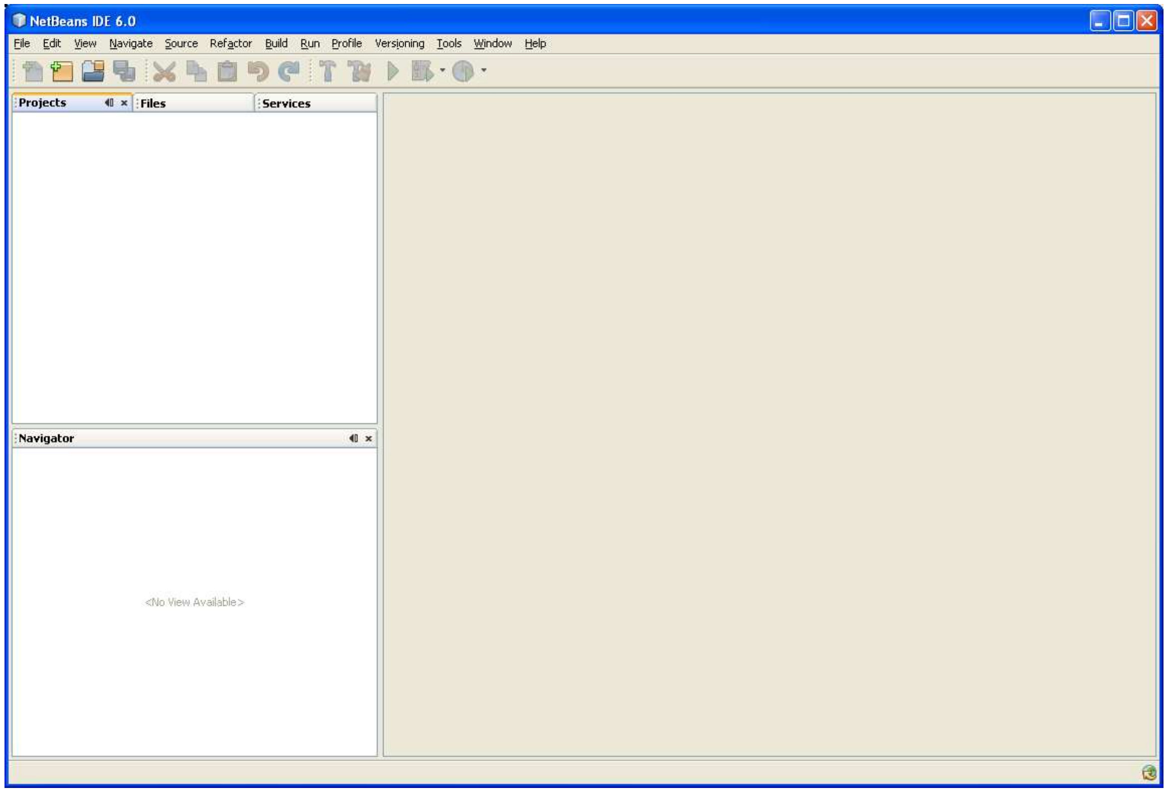
"All Programs" -> "NetBeans" -> "NetBeans IDE 6.0" from the "Start" menu.

JUnit:

Running and Creating/Using a “Project” in NetBeans

If starting a new project, close any currently open projects by right-clicking on the project in the left-side project tree view, and selecting “Close”.

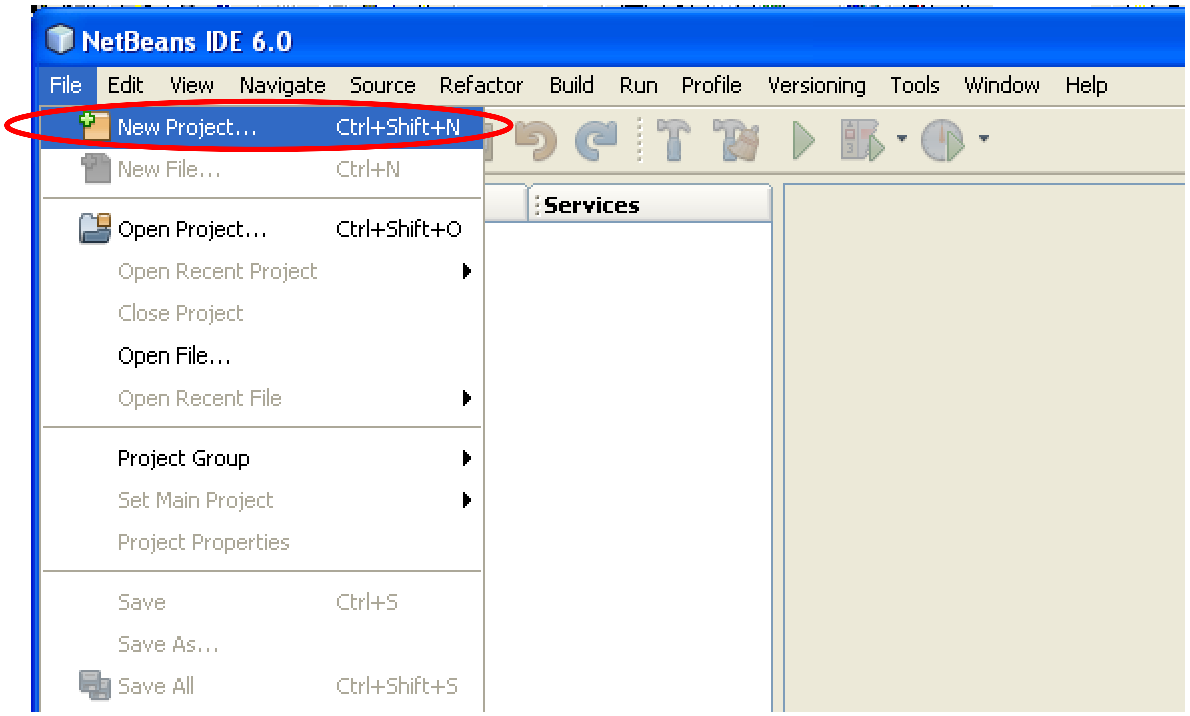
You should then see the following:


NetBeans IDE 6.0
File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help
Projects
Files
Services
Navigator
<No View Available>

The NetBeans IDE (Project Creation)

Creating/Using/Running a "Project" in NetBeans

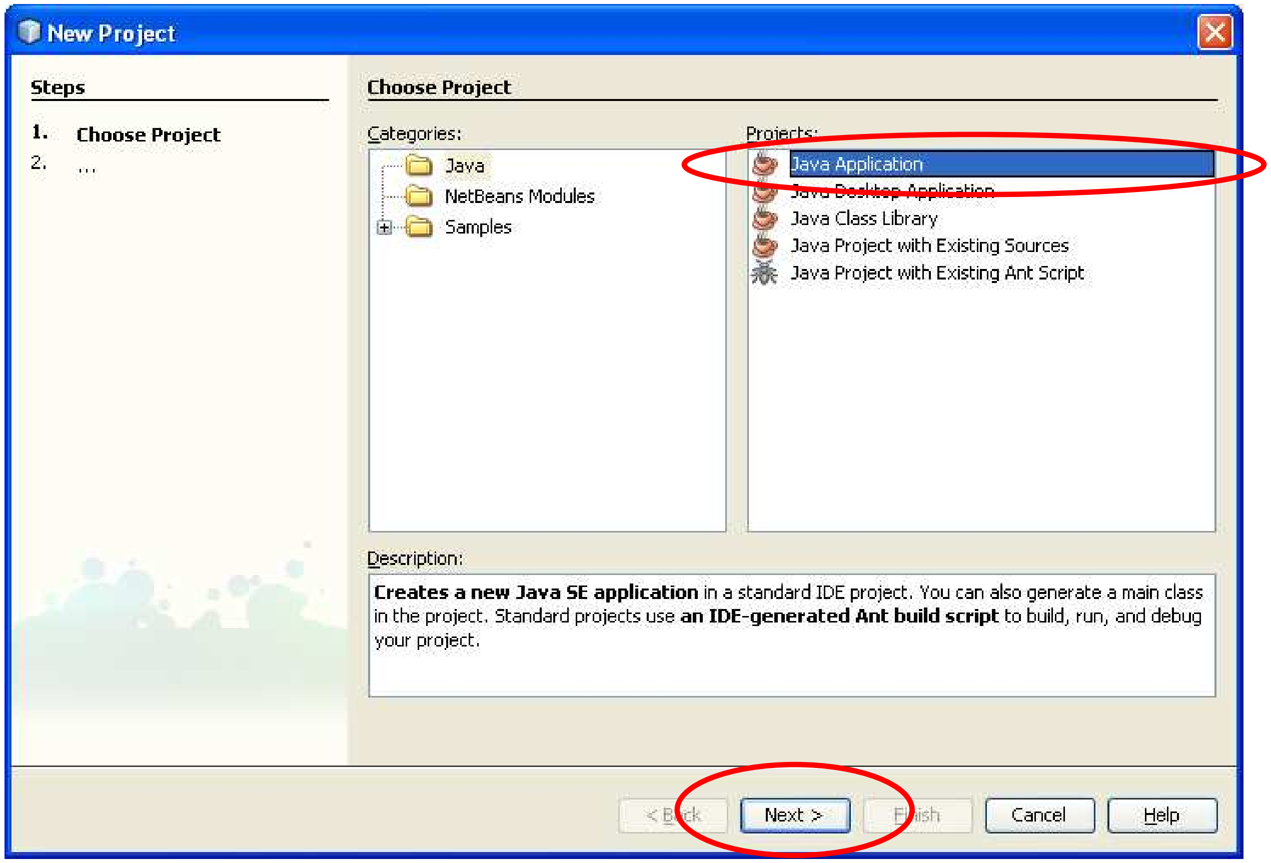
To create a new NetBeans project, select "File" -> "New Project…"



The NetBeans IDE (Project Creation)

Creating/Using/Running a "Project" in NetBeans

In the subsequent "New Project" window, select "Java Application", then click "Next >".



New Project

Steps
1. Choose Project
2. ...

Choose Project

Categories:
- Java
- NetBeans Modules
- Samples

Projects:
- Java Application
- Java Desktop Application
- Java Class Library
- Java Project with Existing Sources
- Java Project with Existing Ant Script

Description:

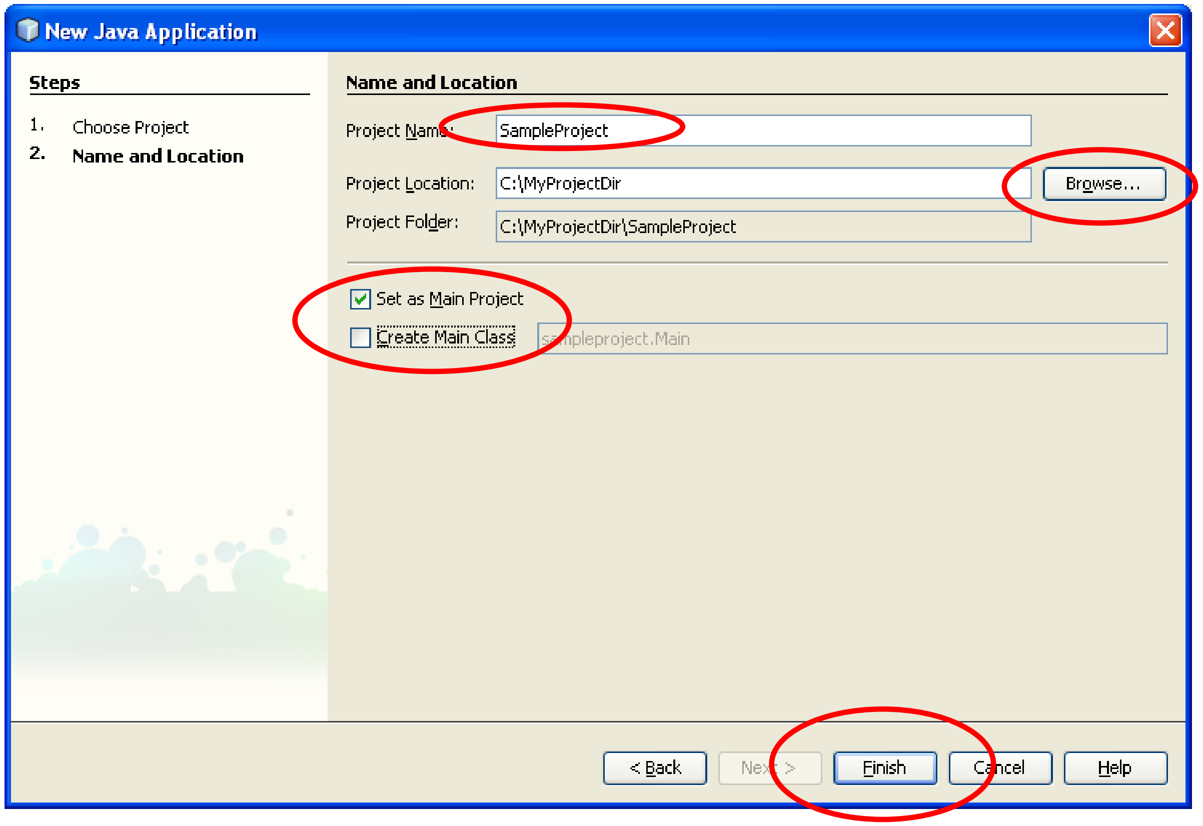
Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

The NetBeans IDE (Project Creation)

Creating/Using/Running a "Project" in NetBeans

Next, in the "New Java Application" window:

- Enter a project name ("SampleProject" in this example)
- Click "Browse…" to select a location for your project ("C:\MyProjectDir" in this example)
- Be sure "Create Main Class" is *not* selected. Then click "Finish".



New Java Application

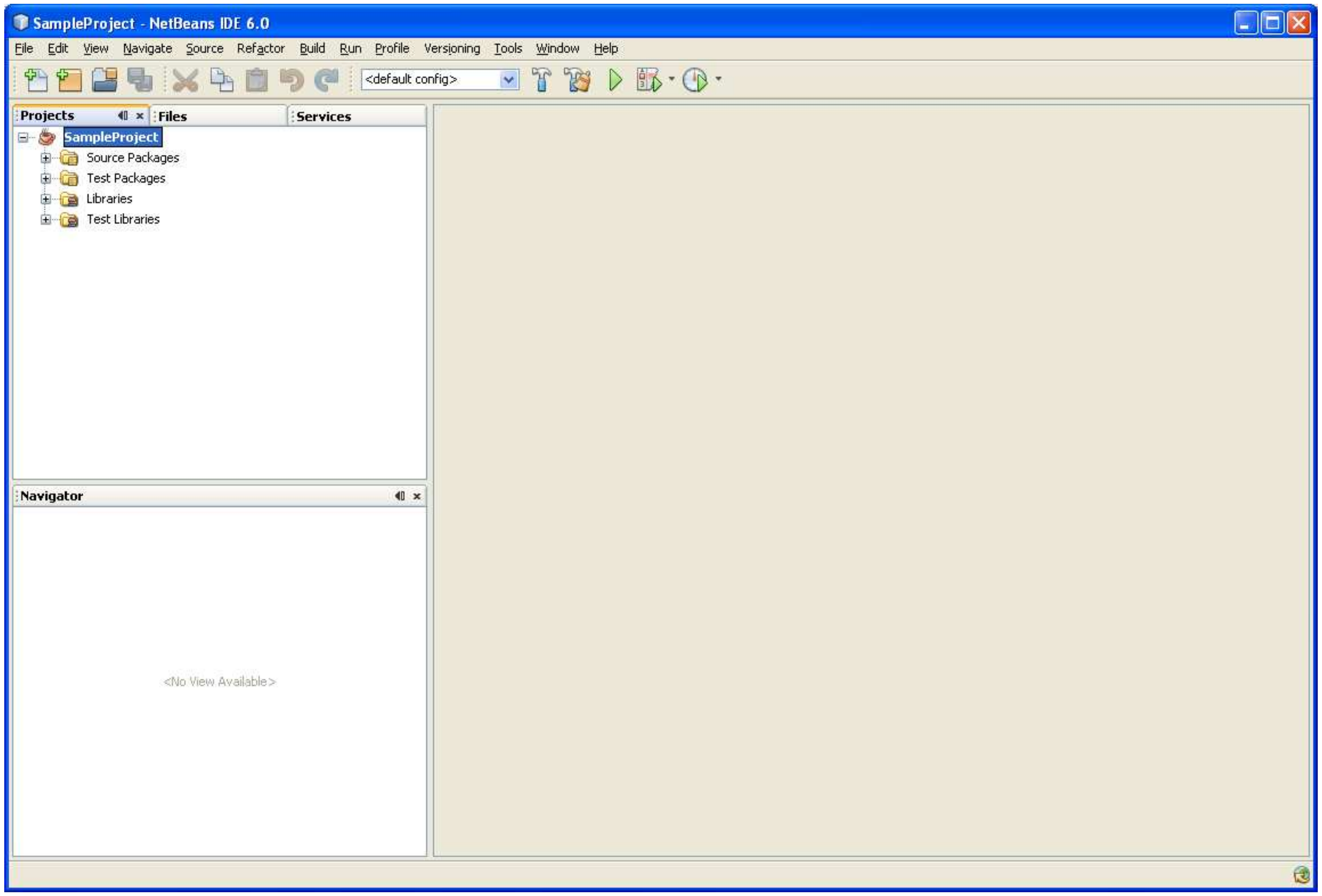
Project Name:	SampleProject
Project Location:	C:\MyProjectDir
Project Folder:	C:\MyProjectDir\SampleProject

[✓] Set as Main Project
[ ] Create Main Class — sampleproject.Main

The NetBeans IDE (Project Creation)

Creating/Using/Running a "Project" in NetBeans

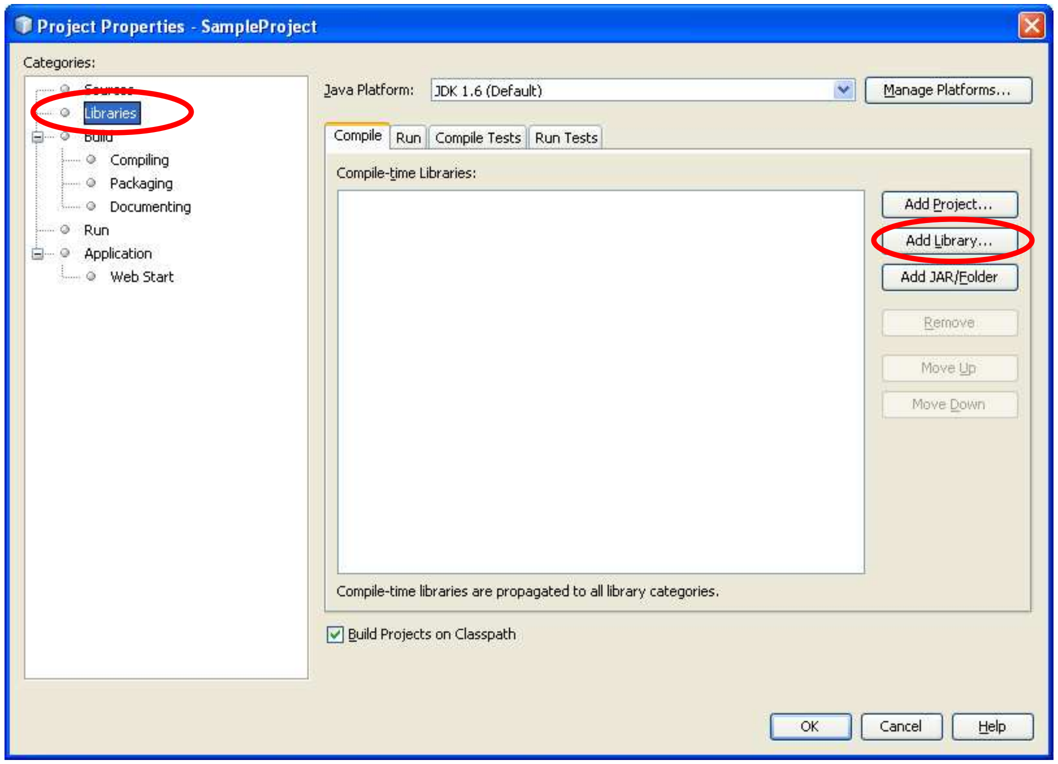
NetBeans will now create and open your new project. The IDE should now look like this:



The NetBeans IDE (Project Creation)

Creating/Using/Running a “Project” in NetBeans

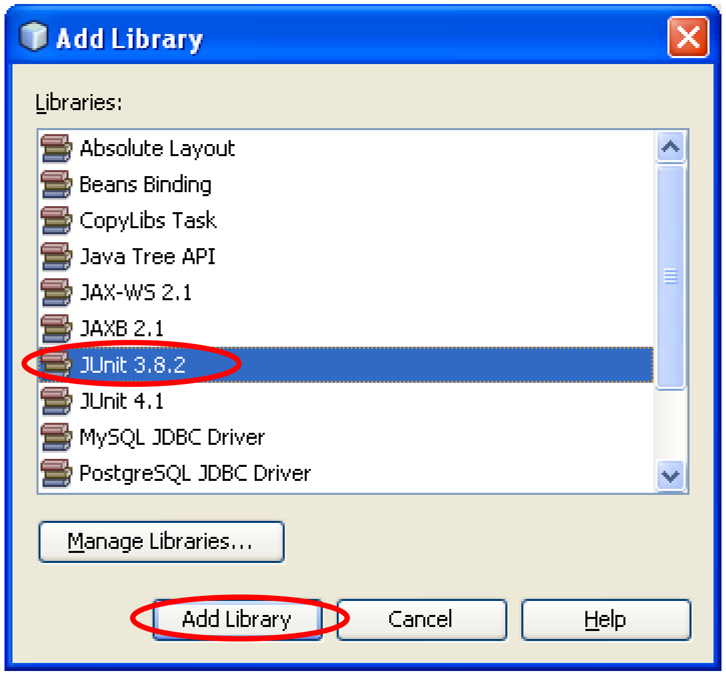
Right-Click on the new project in the left side “Projects” tab, and select “Properties” from the displayed menu. This will open the “Project Properties” window.
Select “Libraries” (left side), then click the “Add Libraries…” button (right side).



The NetBeans IDE (Project Creation)

Creating/Using/Running a "Project" in NetBeans

NetBeans includes the JUnit libraries with its standard installation so it should already appear in the list. Scroll down the list of libraries and select "JUnit 3.8.2". Then click "Add Library", then click "Ok".

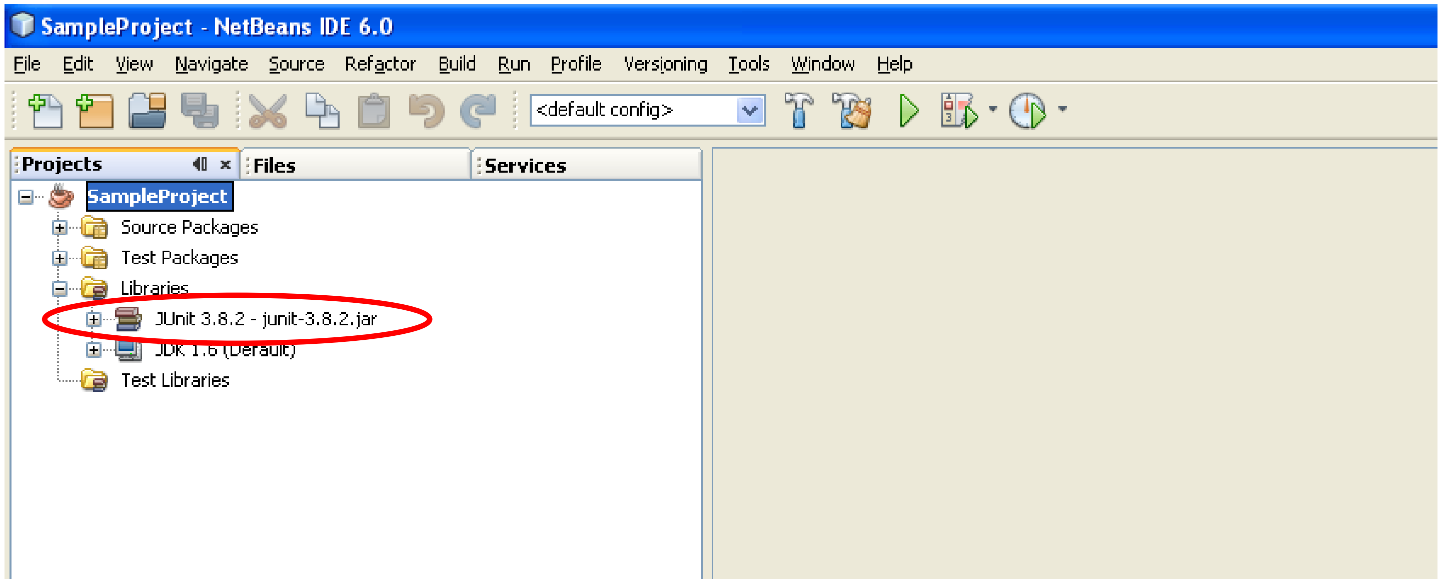


NOTE: If you are not using NetBeans, you can download and install JUnit by going to: http://www.junit.org

The NetBeans IDE (Project Creation)

Creating/Using/Running a “Project” in NetBeans

The JUnit library has now been added to your project.

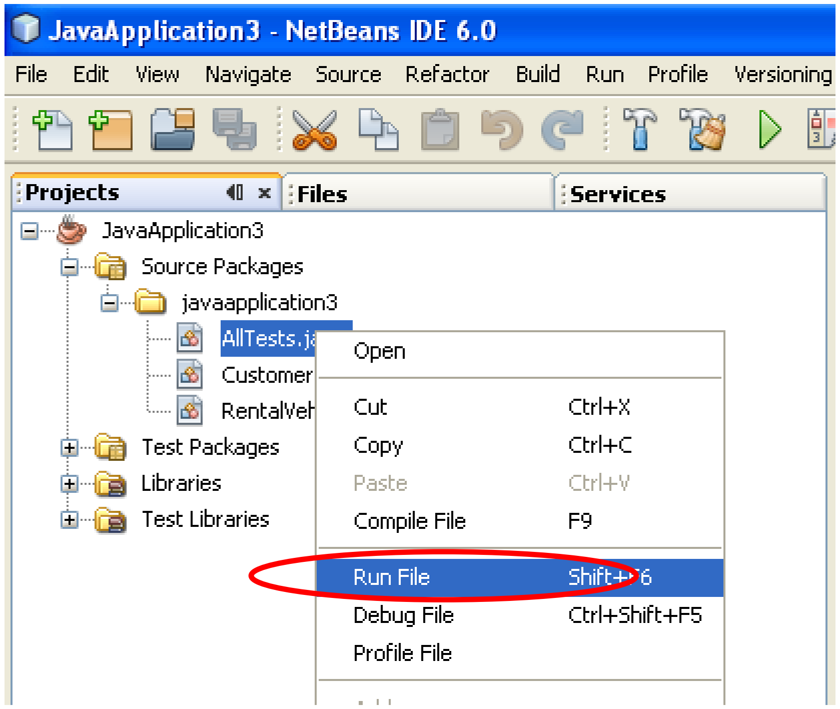

SampleProject - NetBeans IDE 6.0
File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help
<default config>
Projects
Files
Services
SampleProject
Source Packages
Test Packages
Libraries
JUnit 3.8.2 - junit-3.8.2.jar
JDK 1.6 (Default)
Test Libraries

Continue using NetBeans now to create and compile your applications as you normally would.

The NetBeans IDE (Project Creation)

Running and Creating/Using a "Project" in NetBeans

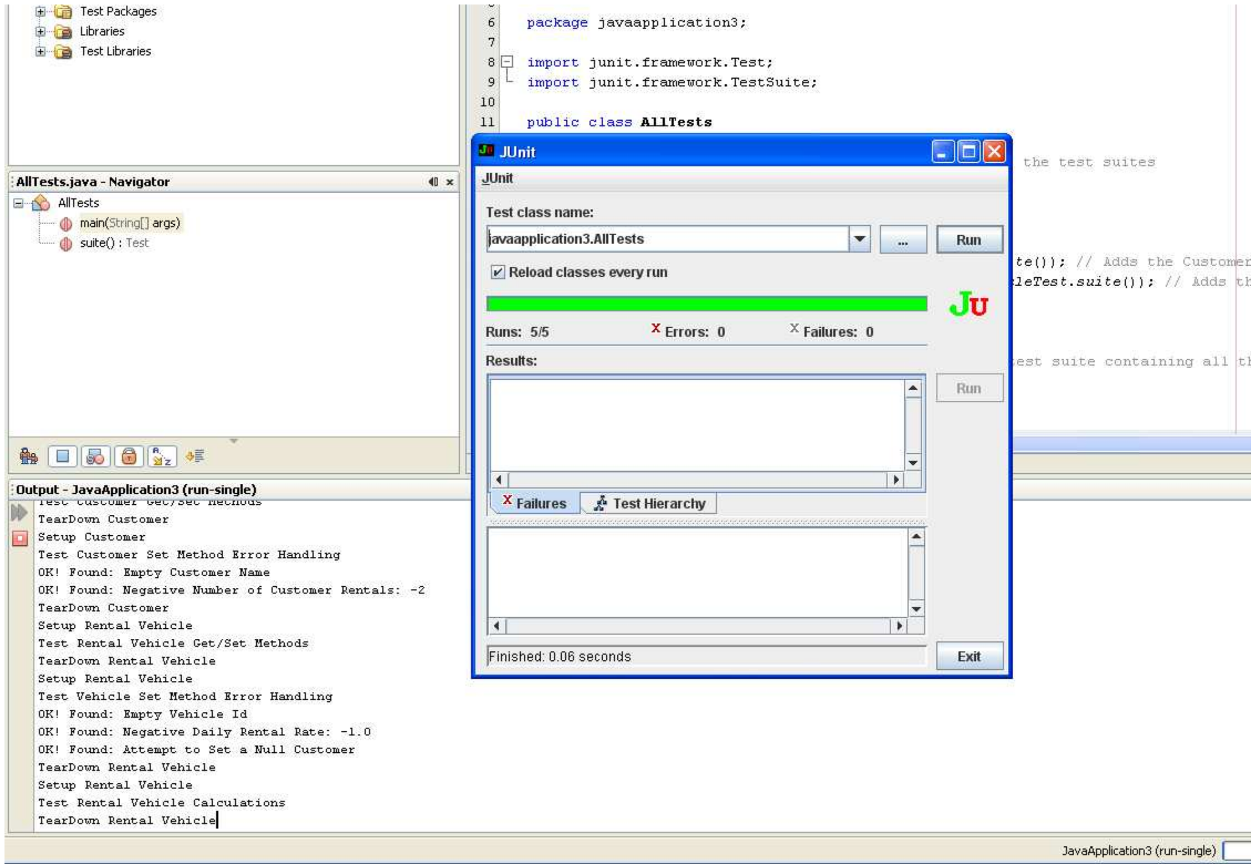
To execute your JUnit tests using NetBeans, right-click on the "AllTests" class and select "Run File"



The NetBeans IDE (Project Creation)

Running and Creating/Using a "Project" in NetBeans

NetBeans will execute all of the test suites that the "suite()" method of your "AllTests" class indicates. Netbeans then displays the results of your JUnit tests in the lower output pane as shown. This image shows that the tests all ran with no errors.



```java
package javaapplication3;

import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests
```

JUnit

Test class name: javaapplication3.AllTests

[✓] Reload classes every run

Runs: 5/5   Errors: 0   Failures: 0

Results:

Finished: 0.06 seconds

AllTests.java - Navigator

AllTests
- main(String[] args)
- suite() : Test

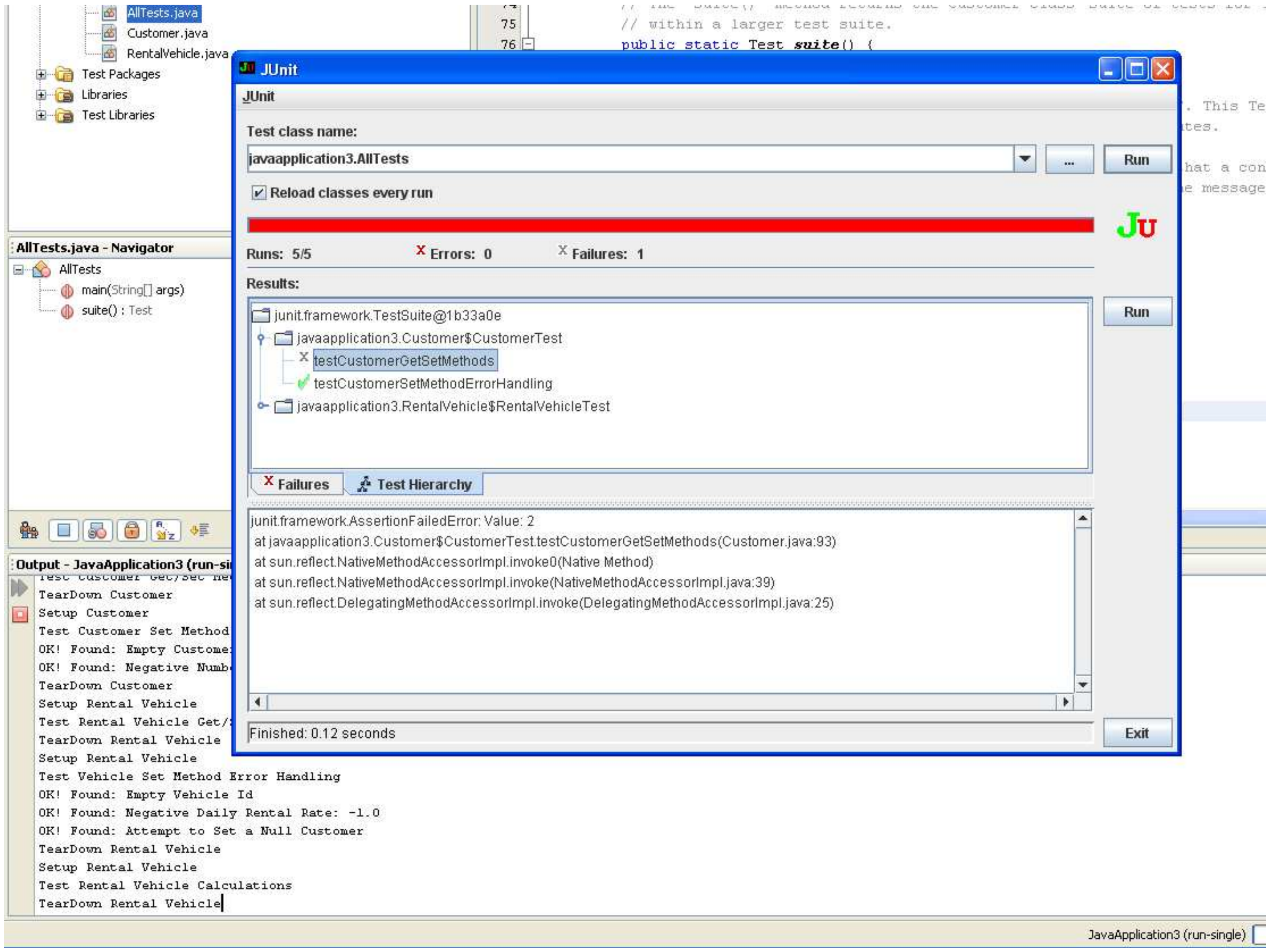
Output - JavaApplication3 (run-single)

```
TearDown Customer
Setup Customer
Test Customer Set Method Error Handling
OK! Found: Empty Customer Name
OK! Found: Negative Number of Customer Rentals: -2
TearDown Customer
Setup Rental Vehicle
Test Rental Vehicle Get/Set Methods
TearDown Rental Vehicle
Setup Rental Vehicle
Test Vehicle Set Method Error Handling
OK! Found: Empty Vehicle Id
OK! Found: Negative Daily Rental Rate: -1.0
OK! Found: Attempt to Set a Null Customer
TearDown Rental Vehicle
Setup Rental Vehicle
Test Rental Vehicle Calculations
TearDown Rental Vehicle
```

JUnit

Running and Creating/Using a “Project” in JBuilder

If any of your test methods failed, the display would reflect this as shown below:



```
// within a larger test suite.
public static Test suite() {
```

Test class name: javaapplication3.AllTests

Reload classes every run: ✓

Runs: 5/5 Errors: 0 Failures: 1

Results:

```
junit.framework.TestSuite@1b33a0e
  javaapplication3.Customer$CustomerTest
    X testCustomerGetSetMethods
    ✓ testCustomerSetMethodErrorHandling
  javaapplication3.RentalVehicle$RentalVehicleTest
```

```
junit.framework.AssertionFailedError: Value: 2
 at javaapplication3.Customer$CustomerTest.testCustomerGetSetMethods(Customer.java:93)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

Finished: 0.12 seconds

```
TearDown Customer
Setup Customer
Test Customer Set Method
OK! Found: Empty Custome
OK! Found: Negative Numb
TearDown Customer
Setup Rental Vehicle
Test Rental Vehicle Get/
TearDown Rental Vehicle
Setup Rental Vehicle
Test Vehicle Set Method Error Handling
OK! Found: Empty Vehicle Id
OK! Found: Negative Daily Rental Rate: -1.0
OK! Found: Attempt to Set a Null Customer
TearDown Rental Vehicle
Setup Rental Vehicle
Test Rental Vehicle Calculations
TearDown Rental Vehicle
```

JUnit Version Differences: JUnit 3.8 vs. JUnit 4.X

Version 4 of the JUnit test framework has been available for use for quite some time. However, many projects have a wealth of JUnit 3-style tests and developers may choose to continue using it. If, on the other hand, you decide to dip your toes in JUnit 4 waters, make it a complete immersion. Don't try to mix and match. Two of the main differences between JUnit 3 and JUnit 4 are the obviation of extending the TestCase class, and the identification of test and lifecycle methods (*setUp* and *tearDown*) using Java 5 **annotations**. For example, the following is a valid JUnit 4-style test case (minus the necessary imports):

```
public class BusinessLogicTests {
  @Before
  public void do_this_before_every_test() {
    // set up logic goes here
  }

  @After
  public void clean_up_after_a_test() {
    // teardown logic goes here
  }

  @Test
  public void addition() {
    assertEquals("Invalid addition", 2, 1+1);
  }
}
```

Observe how:

- The class does not extend TestCase
- The “setup” and “teardown” methods are identified by annotations
- The test method does not require a “test” prefix and is also annotated

JUnit

The following highlights the major differences in the way JUnit 4 testing is set up:

1) “@Test”

Mark your test cases with @Test annotations. You don’t need to prefix your test cases with “test”. In addition, your class does not need to extend from “TestCase” class:

```
@Test
public void addition() {
        assertEquals(12, simpleMath.add(7, 5));
}

@Test
public void subtraction() {
        assertEquals(9, simpleMath.substract(12, 3));
}
```

2) @Before and @After

Use @Before and @After annotations for “setup” and “tearDown” methods respectively. They run before and after every test case.

```
@Before
public void runBeforeEveryTest() {
       simpleMath = new SimpleMath();
}

@After
public void runAfterEveryTest() {
       simpleMath = null;
}
```

JUnit

3) “@BeforeClass” and “@AfterClass”

Use @BeforeClass and @AfterClass annotations for class wide “setup” and “tearDown” respectively. Think them as one time setup and tearDown. They run for one time before and after all test cases.

```
@BeforeClass
public static void runBeforeClass() {
           // run for one time before all test cases
}

@AfterClass
public static void runAfterClass() {
           // run for one time after all test cases
}
```

4) Exception Handling

Use “expected” parameter with @Test annotation for test cases that expect exception. Write the class name of the exception that will be thrown.

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {

        // divide by zero
        simpleMath.divide(1, 0);

}
```

JUnit

5) @Ignore

Put @Ignore annotation for test cases you want to ignore. You can add a string parameter that defines the reason of ignorance if you want.

```
@Ignore("Not Ready to Run")
@Test
public void multiplication() {

        assertEquals(15, simpleMath.multiply(3, 5));

}
```

6) Timeout

Define a timeout period in milliseconds with “timeout” parameter. The test fails when the timeout period exceeds.

```
@Test(timeout = 1000)
public void infinity() {

     while (true)
     ;

}
```

JUnit

7) New Assertions

Compare arrays with new assertion methods. Two arrays are equal if they have the same length and each element is equal to the corresponding element in the other array; otherwise, they're not.

- public static void assertEquals(Object[] expected, Object[] actual);
- public static void assertEquals(String message, Object[] expected, Object[] actual);

```
@Test
public void listEquality() {

       List<Integer> expected = new ArrayList<Integer>();
       expected.add(5);

       List<Integer> actual = new ArrayList<Integer>();
       actual.add(5);

       assertEquals(expected, actual);
}
```

Sample Code JUnit 3.8 & JUnit 4.X

The following shows a simple class “Movie.java”. Following that class are 2 test classes. The fiorst uses JUnit 3.8. the second uses JUnit 4.

Movie.java (This is the class to be tested):

```
package junitsample;

import java.sql.Timestamp;
import java.util.GregorianCalendar;
import junitsample.utils.*;

public class Movie {

    private String title;
    private int year;
    private long dateAdded;
    public static final int OLDEST = 1900;

    public Movie(String titleIn, int yearIn) throws NullValueException, InvalidDataException {
        setTitle(titleIn);
        setYear(yearIn);
        setDateAdded(System.currentTimeMillis());
    }

    public long getDateAdded() {
        return dateAdded;
    }

    public String getDateAddedString() {
        return (new Timestamp(dateAdded)).toString();
    }

    private void setDateAdded(long date) throws InvalidDataException {
        if (date <= 0) {
            throw new InvalidDataException("Invalid date value (<= 0) passed to Movie.setDateAdded: " + date);
        }
```

JUnit

```
        if (date > System.currentTimeMillis()) {
            throw new InvalidDataException("Invalid date value (date > current date) passed to Movie.setDateAdded: " + date);
        }
        dateAdded = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String titleIn) throws NullValueException, InvalidDataException {
        if (titleIn == null) {
            throw new NullValueException("Null value passed to Movie.setTitle");
        }
        if (titleIn.isEmpty()) {
            throw new InvalidDataException("Empty String passed to Movie.setTitle");
        }
        if (titleIn.toLowerCase().startsWith("The ".toLowerCase())) {
            title = titleIn.substring(4) + ", The";
        } else if (titleIn.toLowerCase().startsWith("A ".toLowerCase())) {
            title = titleIn.substring(2) + ", A";
        } else {
            title = titleIn;
        }
    }

    public int getYear() {
        return year;
    }

    public void setYear(int yearIn) throws NullValueException, InvalidDataException {
        int currentYr = (new GregorianCalendar()).get(GregorianCalendar.YEAR);
        if (yearIn < OLDEST || yearIn > currentYr) {
            throw new InvalidDataException("Invalid Movie year passed to Movie.setYear: " + yearIn);
        }
        year = yearIn;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("\n");
        sb.append(String.format("%-12s %s%n", "Title:", getTitle()));
        sb.append(String.format("%-12s %s%n", "Year:", getYear()));
```

```
        sb.append(String.format("%-12s %s%n", "Date Added:", getDateAddedString()));
        return sb.toString();
    }
}
```

MovieTest.java (This is the JUnit test class, using the JUnit 3.8 style setup):

```
package junitsample;

import java.sql.Timestamp;
import java.util.GregorianCalendar;
import java.util.StringTokenizer;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junitsample.utils.InvalidDataException;
import junitsample.utils.NullValueException;


public class MovieTest extends TestCase { // A JUnit 3.8 style test class

    public static String TESTTITLE = "This is a test title";
    public static int TESTYEAR = 2008;
    private Movie testMovie = null;

    public MovieTest(String testName) {
        super(testName);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite(MovieTest.class);
        return suite;
    }

    @Override
    protected void setUp() throws NullValueException, InvalidDataException {
        testMovie = new Movie(TESTTITLE, TESTYEAR);
    }

    @Override
    protected void tearDown() {
```

JUnit

```
        testMovie = null;
    }

    public void testGetDateAdded() {
        System.out.println("getDateAdded");
        long testDate = System.currentTimeMillis();
        long date = testMovie.getDateAdded();
        assertEquals((double) testDate, (double) date, (double) 1000);
    }

    public void testGetDateAddedString() {
        System.out.println("getDateAddedString");
        long date = testMovie.getDateAdded();
        String timeStr = new Timestamp(date).toString();
        String dateStr = testMovie.getDateAddedString();
        assertEquals(timeStr, dateStr);
    }

    public void testSetDateAdded() {
        System.out.println("setDateAdded");
        long testDate = System.currentTimeMillis();
        long date = testMovie.getDateAdded();
        assertEquals((double) testDate, (double) date, (double) 1000);
    }

    public void testGetTitle() {
        System.out.println("getTitle");
        assertEquals(TESTTITLE, testMovie.getTitle());
    }

    public void testSetTitle() {
        System.out.println("setTitle");
        try {
            testMovie.setTitle("This is a different Title");
        } catch (Exception ex) {
            fail(ex.toString());
        }
        assertEquals("This is a different Title", testMovie.getTitle());

        try {
            testMovie.setTitle("The title has The at the beginning");
        } catch (Exception ex) {
            fail(ex.toString());
```

JUnit

```
        }
        assertEquals("title has The at the beginning, The", testMovie.getTitle());

        try {
            testMovie.setTitle("A title has A at the beginning");
        } catch (Exception ex) {
            fail(ex.toString());
        }
        assertEquals("title has A at the beginning, A", testMovie.getTitle());

        try {
            testMovie.setTitle("");
            fail("Accepted Empty String for Title");
        } catch (Exception ex) {
            assertEquals("Empty String passed to Movie.setTitle", ex.getMessage());
        }

        try {
            testMovie.setTitle(null);
            fail("Accepted NULL String for Title");
        } catch (Exception ex) {
            assertEquals("Null value passed to Movie.setTitle", ex.getMessage());
        }
    }

    public void testGetYear() {
        System.out.println("getYear");
        assertEquals(TESTYEAR, testMovie.getYear());
    }

    public void testSetYear() {
        System.out.println("setYear");
        int newYear = 1950;
        try {
            testMovie.setYear(newYear);
        } catch (Exception ex) {
            fail(ex.toString());
        }
        assertEquals(newYear, testMovie.getYear());

        newYear = 1899;
        try {
            testMovie.setYear(newYear);
```

JUnit

```
            fail("Accepted Year before the min. year: " + newYear);
        } catch (Exception ex) {
            assertEquals("Invalid Movie year passed to Movie.setYear: " + newYear, ex.getMessage());
        }

        newYear = (new GregorianCalendar()).get(GregorianCalendar.YEAR) + 1;
        try {
            testMovie.setYear(newYear);
            fail("Accepted Year after the current year: " + newYear);
        } catch (Exception ex) {
            assertEquals("Invalid Movie year passed to Movie.setYear: " + newYear, ex.getMessage());
        }
    }

    public void testToString() {
        System.out.println("toString");
        long addDate = testMovie.getDateAdded();
        String result = testMovie.toString();

        StringTokenizer st = new StringTokenizer(result);

        String label = st.nextToken("\n:");
        String value = st.nextToken("\n:");
        assertEquals(label, "Title");
        assertEquals(value.trim(), "This is a test title");

        label = st.nextToken("\n:");
        value = st.nextToken("\n:");
        assertEquals(label, "Year");
        assertEquals(value.trim(), "2008");

        label = st.nextToken("\n:");
        st.nextToken(" ");
        value = st.nextToken("\n");
        assertEquals(label, "Date Added");
        assertEquals(value.trim(), new Timestamp(addDate).toString());

        label = st.nextToken("\n:");
        value = st.nextToken("\n:");
        assertEquals(label, "Plot");
        assertEquals(value.trim(), "This is a test plot");
    }
}
```

MovieTest.java (This is the JUnit test class, using the JUnit 4 style setup):

```
package junitsample;

import java.util.StringTokenizer;
import java.util.GregorianCalendar;
import java.sql.Timestamp;
import junitsample.utils.InvalidDataException;
import junitsample.utils.NullValueException;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class MovieTest { // A JUnit 3.8 style test class

    public static String TESTTITLE = "This is a test title";
    public static int TESTYEAR = 2008;
    private Movie testMovie = null;

    public MovieTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void runBeforeEach() throws NullValueException, InvalidDataException {
        testMovie = new Movie(TESTTITLE, TESTYEAR);
    }

    @After
    public void runAfterEach() {
        testMovie = null;
```

JUnit

```
}

@Test
public void getDateAddedTest() {
    System.out.println("getDateAdded");
    long testDate = System.currentTimeMillis();
    long date = testMovie.getDateAdded();
    assertEquals((double) testDate, (double) date, (double) 1000);
}

@Test
public void getDateAddedStringTest() {
    System.out.println("getDateAddedString");
    long date = testMovie.getDateAdded();
    String timeStr = new Timestamp(date).toString();
    String dateStr = testMovie.getDateAddedString();
    assertEquals(timeStr, dateStr);
}

@Test
public void getTitleTest() {
    System.out.println("getTitle");
    assertEquals(TESTTITLE, testMovie.getTitle());
}

@Test
public void setTitleTest() throws Exception {
    System.out.println("setTitle");
    try {
        testMovie.setTitle("This is a different Title");
    } catch (Exception ex) {
        fail(ex.toString());
    }
    assertEquals("This is a different Title", testMovie.getTitle());

    try {
        testMovie.setTitle("The title has The at the beginning");
    } catch (Exception ex) {
        fail(ex.toString());
    }
    assertEquals("title has The at the beginning, The", testMovie.getTitle());

    try {
```

JUnit

```
            testMovie.setTitle("A title has A at the beginning");
        } catch (Exception ex) {
            fail(ex.toString());
        }
        assertEquals("title has A at the beginning, A", testMovie.getTitle());

        try {
            testMovie.setTitle("");
            fail("Accepted Empty String for Title");
        } catch (Exception ex) {
            assertEquals("Empty String passed to Movie.setTitle", ex.getMessage());
        }

        try {
            testMovie.setTitle(null);
            fail("Accepted NULL String for Title");
        } catch (Exception ex) {
            assertEquals("Null value passed to Movie.setTitle", ex.getMessage());
        }
    }

    @Test
    public void getYearTest() {
        System.out.println("getYear");
        assertEquals(TESTYEAR, testMovie.getYear());
    }

    @Test
    public void setYearTest() throws Exception {
        System.out.println("setYear");
        int newYear = 1950;
        try {
            testMovie.setYear(newYear);
        } catch (Exception ex) {
            fail(ex.toString());
        }
        assertEquals(newYear, testMovie.getYear());

        newYear = 1899;
        try {
            testMovie.setYear(newYear);
            fail("Accepted Year before the min. year: " + newYear);
        } catch (Exception ex) {
```

JUnit

```
            assertEquals("Invalid Movie year passed to Movie.setYear: " + newYear, ex.getMessage());
        }

        newYear = (new GregorianCalendar()).get(GregorianCalendar.YEAR) + 1;
        try {
            testMovie.setYear(newYear);
            fail("Accepted Year after the current year: " + newYear);
        } catch (Exception ex) {
            assertEquals("Invalid Movie year passed to Movie.setYear: " + newYear, ex.getMessage());
        }
    }

    @Test
    public void toStringTest() {
        System.out.println("toString");
        long addDate = testMovie.getDateAdded();
        String result = testMovie.toString();

        StringTokenizer st = new StringTokenizer(result);

        String label = st.nextToken("\n:");
        String value = st.nextToken("\n:");
        assertEquals(label, "Title");
        assertEquals(value.trim(), "This is a test title");

        label = st.nextToken("\n:");
        value = st.nextToken("\n:");
        assertEquals(label, "Year");
        assertEquals(value.trim(), "2008");

        label = st.nextToken("\n:");
        st.nextToken(" ");
        value = st.nextToken("\n");
        assertEquals(label, "Date Added");
        assertEquals(value.trim(), new Timestamp(addDate).toString());

        label = st.nextToken("\n:");
        value = st.nextToken("\n:");
        assertEquals(label, "Plot");
        assertEquals(value.trim(), "This is a test plot");
    }
}
```

JUnit

Summary:

JUnit:

- JUnit tests allow you to write code faster while increasing quality.
- JUnit is elegantly simple.
- JUnit tests check their own results and provide immediate feedback.
- JUnit tests can be composed into a hierarchy of test suites.
- Writing JUnit tests is inexpensive.
- JUnit tests increase the stability of software.
- JUnit tests are developer tests.
- JUnit tests are written in Java.
- JUnit is free!

JUnit

Summary:

Unit Testing:

- The software does well those things that the tests check.
- Test a little, code a little, test a little, code a little...
- Make sure all tests always run at 100%.
- Run all the tests in the system at least once per day (or night).
- Write tests for the areas of code with the highest probability of breakage.
- Write tests that have the highest possible return on your testing investment.
- If you find yourself debugging using `System.out.println()`, write a test to automatically check the result instead.
- When a bug is reported, write a test to expose the bug.
- The next time someone asks you for help debugging, help them write a test.
- Write unit tests before writing the code and only write new code when a test is failing.