

## Unit Test Worked Example

It is often best to see how something is done from start to finish. This document will provide an example of Test Driven Development (TDD) from specification to finished code.

The example is built with:

- NetBeans 11
- Oracle JDK 11
- Junit 5

Unit testing should always follow a set of steps:

- 1) A specification of some type defines what the class needs to do and/or be.
- 2) You define the public interface of the class (i.e.: only the parts that need to interact with other classes or the outside world). This includes methods, constructors, and any public attributes or constants.
- 3) You list all the publicly known attributes of the class, even though they will be defined as private. There may be other attributes later, but those are probably for internal use only.
- 4) For every attribute, list its constraints. Examples are things like numeric ranges, String lengths, or a set of discrete, allowable values.
- 5) Make a list of every combination of method/constructor, its parameters, and the constraints on those parameters.
- 6) For each item on the list, tell what happens if you supply a good value and what happens if you supply a bad value.
- 7) For each item on the list, write a test method that checks that the constraints are enforced.
- 8) Write the class shell that has all the public methods, constructors, and attributes. Remember, only the public ones! Note that your environment may require that you create the class shell first, and that is acceptable.
- 9) For each test method, do enough coding in the public method being tested so that the public method passes the test.

This looks like a lot, but that's because describing it correctly takes a lot of words. Once you have done this a couple of times, it will be relatively easy, and probably even monotonous.

In the large environments, most of the testing, including the writing of tests, will be automated. In smaller shops, you will do this yourself or someone will write the tests and someone else will code to them. Either way, you will need to understand the process.

## The Specification

Specification document may take many forms. There are two things they have in common: they tell you what the software needs to do and they tie your work to a contract somewhere. The contract part is where your pay comes from, but following the rest will take care of that.

In this example, we will stick to a single class representing items that are available for sale. The format used here is similar to what you may have seen with outlines for your academic writing. The numbering and letters let us refer to parts of the specification to a great amount of detail.

- I. Data records will be kept for sale items. Each item will have the following fields and constraints
  - a. Item ID: The unique identifier for the item. This is an integer not less than zero. This is described in the documentation as 'The ID of the sale item.'
  - b. Item Name: A required alphanumeric value not longer than 256 characters, described as 'The name of the item.'
  - c. Item Description: A required alphanumeric value not longer than 1024 characters, described as 'The human-readable description of the item.'
  - d. Unit Price: A non-negative decimal number to two (2) decimal places, described as 'The current price for which the item should be sold.'
  - e. The code must be able to access or change the values of these fields as needed with the constraints maintained.
  - f. There must be a mechanism to create a record for existing values as well as creating a new record where not all the values are known at the time of the creation.
  - g. There must be a way to display a text representation of the sale item in the format: ITEM: {Item Name} {{Unit Price}}
  - h. There must be a mechanism by which a collection of Sale Items records may be organized alphabetically by Item Name.

#### *1Example Specification Snippet*

This snippet of specification is fairly limited because it only describes the fields and some basic functionality of a single record. In Java terms, a record is represented by a class. This is our first connection between the “spec” and the final code.

### Find the Object-Oriented Parts

We look through the specification and try to find two main things: attributes and operations. Why those? Because that is what OOP is all about.

Obviously we will need a class to manage the Sale Item. To make it Java-friendly, we will call it SaleItem. We can start building the shell of such code easily.

```
/**
 * Represents an item that can be sold.
 *
 * @author Author Name
 */
public class SaleItem {

}
```

#### *2The basic start of a class shell for SaleItem*

Sub-item 'a' tells us we need an integer attribute for the Item ID. This is an attribute for our Sale Item class. The other parts are the description and constraints, which we will document, but won't use until a later step. We can start adding the attributes to the Java class shell. Remember to put the description and constraints into the Javadoc comment above each attribute. Also note, a blank String and an empty String are not the same thing. A "required" String usually means neither empty nor blank.

```
/**
 * The ID of the sale item. An integer not less than zero.
 */
private int itemId;

/**
 * The name of the item. A required String not longer than 256 characters.
 */
private String itemName;

/**
 * The human-readable description of the item. A required String not longer
 * than 1024 characters.
 */
private String description;

/**
 * The current price for which the item should be sold. A non-negative
 * decimal number to two decimal places.
 */
private BigDecimal unitPrice;
```

3The private attributes based on the specification

## Defining the Needed Tests

From here, we can start looking at methods and constructors. Item 'f' in the specification says, "There must be a mechanism to create a record for..." which tells us that there must be constructors. The two cases mentioned are for an "existing record" and for a "new record where not all the values are known". The existing record part is easy; that's the full constructor that requires an argument for every attribute.

Since the second constructor does not go into any details about combinations of values, we must assume that an instance of SaleItem can be instantiated with no arguments. At this point, we will need to do some thinking. The job of the constructor is to initialize the attributes of the object. The attributes of this object have constraints. We will need to see if the default values for the data types are allowed. If they are not, the specification must tell us what values with the no-argument constructor. If the specification does not do so, the business analyst must go back to the customer and get the answers.

**Remember, unassigned primitive variables default to zero while unassigned reference variables default to null.**

At this point, we can start making a list of the tests needed for the constructors. The test for the no-argument constructor is simple; just call the constructor and then call all the accessors, one after the other, to make sure that the constructor used the correct default values.

For the full constructor, it is a little more complex. There are four parameters, each with its own constraints. We have to find all the tests of the constraints and then test each parameter, one at a time, for each of its constraints. For each test method, we provide a known good value for any parameter not being tested by that test.

This combination of factors gives seventeen (17) different tests. This is the list of the tests:

#### *No-Argument Constructor*

- Test SaleItem()
- Expected return value: new SaleItem object
- Related test(s)
  - o getItemId() returns default item ID value
  - o getItemName() returns default item name
  - o getDescription() returns default description
  - o getUnitPrice() returns default unit price

#### *Full Constructor - Item ID = 0*

- Test SaleItem(0, goodItemName, goodDescription, goodUnitPrice)
- Expected return value: new SaleItem object
- Related test(s)
  - o getItemId returns 0

#### *Full Constructor - Item ID < 0 i.e.: ID = -1*

- Test SaleItem(-1, goodItemName, goodDescription, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

#### *Full Constructor - Item ID > 0 i.e.: ID = 500 (arbitrary positive value)*

- Test SaleItem(500, goodItemName, goodDescription, goodUnitPrice)
- Expected return value: new SaleItem object
- Related test(s)
  - o getItemId() returns 500

#### *Full Constructor - Item Name Null*

- Test SaleItem(goodItemId, null, goodDescription, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Item Name Blank i.e.: length = 0*

- Test SaleItem(goodItemId, "", goodDescription, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Item Name Length > 256, length = 257 i.e.: bad boundary high*

- Test SaleItem(goodItemId, longName257, goodDescription, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Item Name Length = 256 i.e.: good boundary high*

- Test SaleItem(goodItemId, longName256, goodDescription, goodUnitPrice)
- Expected return value: new SaleItem
- Related test(s)
  - o getItemName() returns longName256

*Full Constructor - Item Name Length = 1 i.e.: good boundary low*

- Test SaleItem(goodItemId, longName1, goodDescription, goodUnitPrice)
- Expected return value: new SaleItem
- Related test(s)
  - o getItemName() returns longName1

*Full Constructor - Description null*

- Test SaleItem(goodItemId, goodItemName, null, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Description Blank i.e.: length = 0*

- Test SaleItem(goodItemId, goodItemName, "", goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Description Length = 1025 i.e.: bad boundary high*

- Test SaleItem(goodItemId, goodItemName, description1025, goodUnitPrice)
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Description Length = 1024 i.e.: good boundary high*

- Test SaleItem(goodItemId, goodItemName, description1024, goodUnitPrice)
- Expected return value: new SaleItem
- Related test(s)
  - o getDescription() returns description1024

*Full Constructor - Description Length = 1 i.e.: good boundary low*

- Test SaleItem(goodItemId, goodItemName, description1, goodUnitPrice)
- Expected return value: new SaleItem
- Related test(s)
  - o getDescription() returns description1

*Full Constructor - Unit Price negative i.e.: bad boundary low*

- Test SaleItem(goodItemId, goodItemName, goodDescription, new BigDecimal("-1.0"))
- Expected return value: IllegalArgumentException
- Related test(s)
  - o None

*Full Constructor - Unit Price zero i.e.: good boundary low*

- Test SaleItem(goodItemId, goodItemName, goodDescription, new BigDecimal("0.0"))
- Expected return value: new SaleItem
- Related test(s)
  - o getUnitPrice() returns BigDecimal("0.0")

*Full Constructor - Unit Price 100.00 i.e.: arbitrary good value*

- Test SaleItem(goodItemId, goodItemName, goodDescription, new BigDecimal("100.00"))
- Expected return value: new SaleItem
- Related test(s)
  - o getUnitPrice() returns BigDecimal("100.00")

Several patterns emerge from these tests. The biggest is that any sort of bad value for an argument should result in an IllegalArgumentException. This prevents the creation of a bad object. Because these

calls throw an Exception, there are no related tests. If there was no object instantiated, you cannot call an accessor to see what the value is; it simply does not exist.

However, if the values are good, we get a new instance of the object. That is the first step. Since the constructor must initialize the instance attributes, we must check that it actually happened. We call the related accessor and inspect the value returned. If the return value is what we expect, then the test passes. If the returned value does not match the expectation, then the test failed.

The list of tests uses variable names for many of the arguments. This makes it easy to read but it can also show where such variables can be used when we write our test methods later. In some cases, it is more communicative to use values that are not variables, such as when we use BigDecimal. We can still use variables for those values when we write the test methods.

When we have ranges of values, there is usually a boundary at each end, low and high. Tests must cover both sides of the boundary. When the range is very wide or when we have a single boundary, it is common to test an arbitrary good value just to be sure the logic works away from the boundaries. Also note that “required String value” means not null and that the length must be greater than zero (blank String). A String length of greater than zero makes another boundary condition.

## Tests for Accessors

Defining tests for accessors is relatively simple. The test will instantiate the class to form an object that has a value for an attribute. Then, the test calls the accessor and then compares the returned value to an expected value.

Assume that the no-argument constructor assigns zero to the itemId attribute. The test for that accessor would be listed as:

### Accessor for Item ID

- Instance from no-argument constructor
- Expected return value: 0 (constructor default)
- Related test(s)
  - o getItemID() returns 0
  - o assert actual return value == expected value

***Remember, accessor methods, also called “getters” simply return the value of an attribute. Typically, there is no other logic involved, but that is not prohibited.***

It is important to specify what values are going into the constructor that creates the test instance, as well as the expected return value. The test will only pass if the actual return value is equal to the expected value. If they are not equal, the test will fail.

When it comes time to write the actual test methods, the tests for the accessors must coincide with the tests for the no-argument constructor (if there is one).

## Tests for Mutators

The tests for mutators are a little more complex. You will need at least one test for each constraint on the related attribute. For example, the item ID is “an integer not less than zero”. This means we have three tests: less than zero, zero, and more than zero.

For less than zero, we want to pick a value just on the bad side of that boundary. In this case, the value is -1. The good side of the boundary is zero itself. For more than zero, we can pick an arbitrary positive value.

For the allowed values, we call the mutator to set the value, then call the accessor to check that the value matches the intended new value.

***A mutator (setter) is a method that assigns a value to an attribute. Typically, the mutator will validate the value first and throw an exception if the value violates data constraints.***

For prohibited values there are a couple of steps. First, we instantiate the object, then we use the accessor to get the value of the relevant attribute. Next, we use the mutator to attempt to set the bad value. This should throw an Exception; if it does not then the test fails. Finally, we use the accessor again and compare the returned value to the original value. If they are equal, then the test passes. If they are not equal, then the test failed.

Using the item ID example gives us the following tests:

### *Mutator for Item ID = -1 i.e.: bad boundary*

- Instance from no-argument constructor
- Original value = instance.getItemId()
- Expected value = IllegalArgumentException
- Related test(s)
  - o getItemID() equals original value

### *Mutator for Item ID = 0 i.e.: good boundary*

- Instance from no-argument constructor
- Expected value = 0
- Call setItemId() with 0 as the argument
- Related test(s)
  - o getItemID() equals 0

### *Mutator for Item ID = 100 i.e.: arbitrary good value*

- Instance from no-argument constructor
- Expected value = 100
- Call setItemId() with 100 as the argument
- Related test(s)
  - o getItemID() equals 100



The other constraint and mutator combinations would be:

- setName() with a null name
- setName() with a blank name
- setName() with a name with length of one character
- setName() with a name with length of 256 characters
- setName() with a name with length of 257 characters
- setDescription() with a null description
- setDescription() with a blank description
- setDescription() with a description of length of one character
- setDescription() with a description of length of 1024 characters
- setDescription() with a description of length of 1025 characters
- setUnitPrice() with a null value
- setUnitPrice() with a value of -0.01
- setUnitPrice() with a value of 0.00
- setUnitPrice() with a value of 0.01
- setUnitPrice() with an arbitrary positive value

## Testing Other Methods

Most classes will have methods other than constructors, accessors, and mutators. These methods must be tested as well. How to test them will depend on what each method does. You will need to determine the types of inputs, their constraints (if any), and what the output should be for those inputs.

Note that we only test the public methods of a class. The private methods should be part of the logic that makes the public methods work.

Consider the following methods from the SaleItem class:

***Accessors and mutators directly interact with attributes of the object. Other, general methods are allowed as well and, if they are public, must be tested.***

```

/**
 * An override of the toString() method.
 *
 * @return ITEM: {itemName} ({unitPrice})
 */
@Override
public String toString() {
    throw new UnsupportedOperationException("Not supported yet.");
}

/**
 * An implementation of compareTo() from the Comparable interface. Used
 * to sort SaleItem objects by name in alphabetical order.
 *
 * @param other the other SaleItem for comparing
 * @return less than zero, zero, or greater than zero
 */
@Override
public int compareTo(SaleItem other) {
    throw new UnsupportedOperationException("Not supported yet.");
}

```

4The toString() and compareTo() methods of the SaleItem class

There are two methods: toString() and compareTo(). The toString() method is an override of the method from the Object class and just returns a String representation of the SaleItem instance. The compareTo() method is from the Comparable interface; it is used when sorting collections of instances.

### Test toString()

To test the toString() method, you need to know what the String representation should look like. In this case, the Javadoc comment says that the String will have the text "ITEM: ", then the item name, then "(", the unit price, and ")". Note that the curly braces in the description are there to show that you replace "itemName" and "unitPrice" with the actual values. This means that, if you know the values for itemName and unitPrice, then you should be able to figure out what the toString() method returns.

The test definition should look like this:

#### toString() for itemName = "Test" and unitPrice = 1.00

- Instance from constructor using "Test" and 1.00, with other good values
- Expected value = "ITEM: Test (1.00)"
- Result = call toString()
- Compare result == expected value

## Test compareTo()

The Comparable interface only has one method: compareTo(). This method takes another instance of the same type and performs a comparison. The return value is an int with one of three options: less than zero, zero, or more than zero. Each of those means something for purposes of sorting.

Assume that we have two instance of the same class: alpha and beta. We call alpha.compareTo(beta) and get a result. The result can be interpreted as:

Less than zero	Object alpha is "less than" object beta, so alpha should occur first in a sorted collection.
Zero	Objects alpha and beta are equal for sorting purposes and either may come first in a sorted collection.
Greater than zero	Object alpha is "greater than" object beta, so beta should occur first in a sorted collection.

There is a fourth outcome. If you call compareTo() and pass an object that is not the correct data type, the method will throw a ClassCastException.

Because there are four possible outcomes, there are four tests for this method. The Javadoc comment says that SaleItem objects are sorted alphabetically by itemName, so three of the tests must use itemName values that match the desired sorting pattern. The fourth test uses an object of a different class.

### *compareTo() for itemName alpha and itemName beta*

- x = instance from constructor using "alpha" with other good values
- y = instance from constructor using "beta" with other good values
- x.compareTo(y) < 0

### *compareTo() for itemName alpha and itemName alpha*

- x = instance from constructor using "alpha" with other good values
- y = instance from constructor using "alpha" with other good values
- x.compareTo(y) == 0

### *compareTo() for itemName alpha and itemName beta*

- x = instance from constructor using "beta" with other good values
- y = instance from constructor using "alpha" with other good values
- x.compareTo(y) > 0

### *compareTo() for itemName alpha*

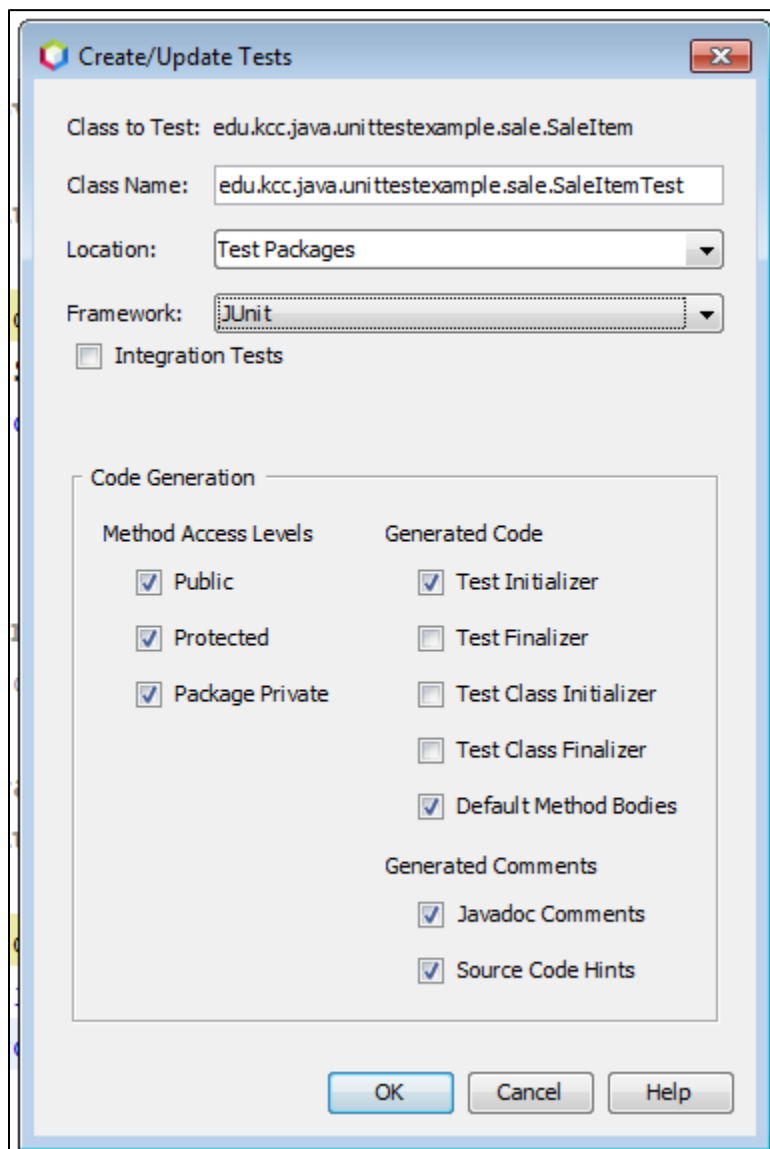
- x = instance from constructor using "alpha" with other good values
- x.compareTo("Some String") throws ClassCastException

## Writing the Tests

So far, we have only made a list of the needed tests. We know the methods to be tested, the inputs, and the expected results. That's the starting point. Writing the tests themselves is next.

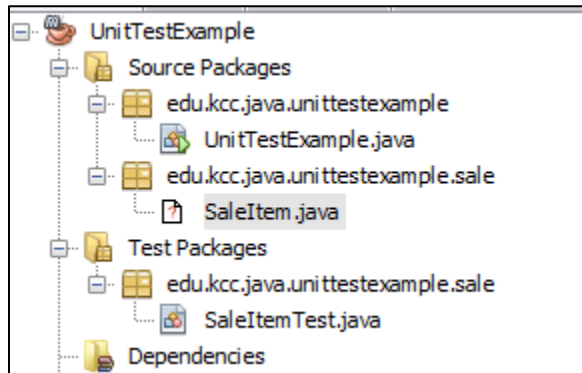
We will use the Junit test framework. This is software that will run inside our NetBeans environment specifically for the purpose of testing our classes. The tests will become part of the NetBeans project, but will not be part of the jar file you would use to distribute your code.

To create the tests, find the class you wish to test in the project explorer pane of NetBeans. Right-click on the class and choose Tools → Create/Update Tests. This will open the Create/Update Tests dialog. It should look something like this:



5The Create/Update Tests dialog

On the dialog, click the OK button to generate the unit tests. The test is a Java class file with the name of your class but with “Test” appended to the end. In the project explorer, you will find a folder labeled TestPackages. This folder contains any test classes generated in this way.



*6The TestPackages folder will hold all of the generated unit test classes*

Be warned, the automatically generated test methods will not cover all of the tests you need. There may even be methods that are underlined in red because there is something wrong with them. That’s okay; over time you will figure out how to comment those out or fix them depending on your needs.

Also note that all of the generated test methods have the following code:

```
// TODO review the generated test code and remove the default call to fail.  
fail("The test case is a prototype.");
```

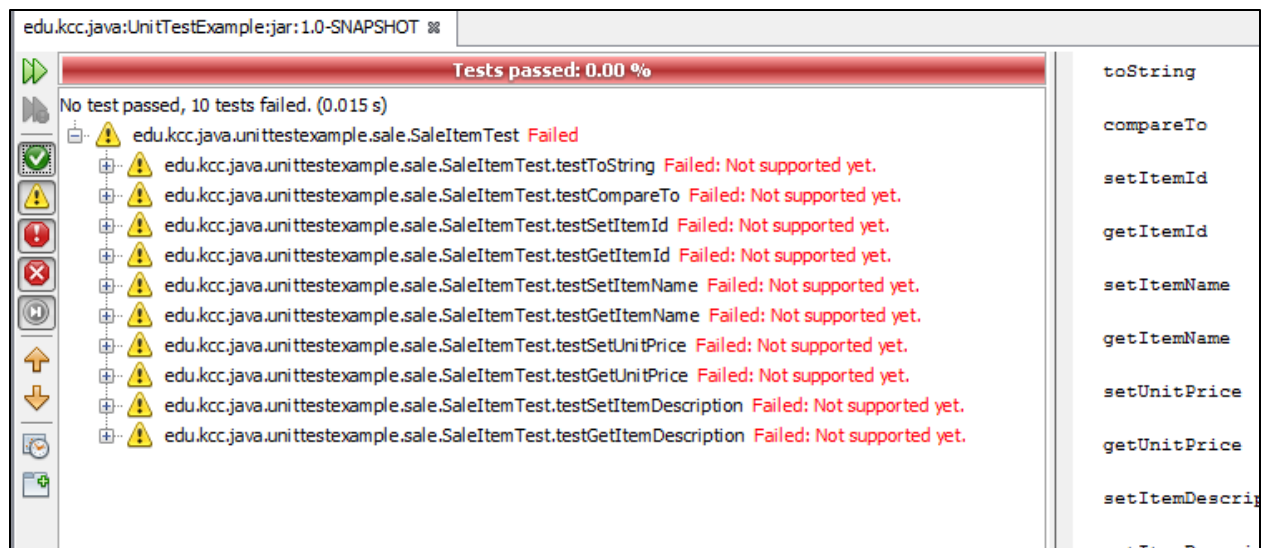
*7The call to the fail() method appears at the end of every automatically generated unit test method*

This is there to remind you not to trust the automatic code. That’s good because you need to change the test methods to match your list of tests anyway.

## A First Run of the Tests

To run the test, make sure the test class has focus and use the key combination Ctrl+F6. There will be a lot of output while the tests run, but we are looking for particular final output.

The output is color codes with little icons showing what happened. When the tests pass, the icon for that method is green. The yellow icon appears next to methods that failed. Note that there is red text next to each failed method with, “Failed: Not supported yet.” It gets that text from the UnsupportedOperationException thrown in each of the SaleItem methods. That will go away as we work on the SaleItem methods later.



*Example output from the initial run. Note that all the tests failed*

If your class has a no-argument constructor, like in this example, it will get tested when you test the accessors. You would need the accessors to make sure the no-argument constructor worked, and you need the no-argument constructor to test that the accessors worked. Because of this, we will start by working on the test methods for the accessors.

## Helper Methods

In the tests, we know that we will need String values of various lengths. It is easy to make an empty String or a String with a length of one. However, we will need Strings with lengths of 257 and 1024, for example. To make this easier, we can make helper methods that build String values for us. If you have tests that need other things built, the same idea applies.

In this example, the method takes a length parameter and builds a String of that length. The String is filled with an arbitrary character 'X' because none of our tests care about the content of the String, only its length. You could build a more complex version of content were a concern.

```

/**
 * Builds and returns a String of characters of the length specified.
 *
 * @param length The desired length of the String
 * @return the String
 */
private String buildString(int length) {
    char[] chars = new char[length];
    for (int i = 0; i < chars.length; i++) {
        chars[i] = 'X';
    }
    return new String(chars);
}

```

9 A helper method to build Strings of a specified length

## Rewriting the Tests for Accessors

The generated test method for getItemId() is called testGetItemId(). It prints the method being tested. It then creates an instance of SaleItem, which it assigns to a variable called instance. The method defines a variable called expectedResult, meaning “expected result” and assigns the value 0. It calls instance.getItemId() and assigns the return value to a variable called result. Finally, it calls the method assertEquals() and passes both expectedResult and result.

The assertEquals() method is part of the Junit framework. If the two items passed in are equal (by whatever way equal is defined for the item type) then the test passes. If the items are not equal, the test fails.

Note that we need to remove the line that calls the fail() method. This method is also supplied by the Junit framework. We will use it in later tests when we definitely want something to fail, such as when a bad value was allowed into a method.

The test for getUnitPrice() is a little different. The expectedResult in the generated method is null. We have to wonder if the no-argument constructor would set the attribute to null or if it would set it to a default value such as 0.00. If the business rules say that null is acceptable, then this is fine. If we want the 0.00, then we will have to alter the test to look for that. The same can be said for the String values for itemName and description, the tests for which are looking for empty Strings.

## Rewriting the Tests for Mutators

The mutators will take more work. The software only generated one method per mutator. Our list of needed tests say we need more than that. We will have to write all the missing tests.

There is another issue with the generated tests; they don’t check to see if the setter worked. The test just calls the setter and assumes that everything worked fine.

Our tests for the mutators will follow one of two patterns. One pattern is for allowed values and the other is for disallowed values.

The pattern for the allowed values is:

- 1) Print the description of the test (method name, good/bad value, etc.)
- 2) Set a variable for the value to be set (different from the default value)
- 3) Create an instance of the object using the no-argument constructor
- 4) Call the setter with the variable
- 5) Call the getter and compare the result with the variable: equal means pass, unequal means fail

The pattern for disallowed values is:

- 1) Print the description of the test (method name, good/bad value, etc.)
- 1) Set a variable for the value to be set (this is a disallowed value)
- 2) Create an instance of the object using the no-argument constructor
- 3) Call the getter and assign the result to a variable
- 4) In a try block
  - a. Call the setter with the variable
  - b. Call the fail() method with a statement saying the bad value was allowed
- 5) In the catch block (usually catching IllegalArgumentException)
  - a. Call the getter and compare the result to the original value (step 3 above)

The pattern for the allowed values is straight forward. The pattern for disallowed values is more complex. There are good reasons for this. For a disallowed value, a test passes when it results in an Exception *and* the value of the attribute does not change.

To do all of that, we instantiate the object. Then we use the accessor method to get the original value of the attribute. Because a bad value must result in an Exception, the call to the setter happens inside of a try statement. When we catch the exception, we call the accessor again and compare that value to the original value (stored in a local variable) to make sure nothing changed.

In the try block, if we do not get an Exception, then we must call fail() for that error. When calling the fail() method, we pass a String argument describing what went wrong, such as, "Allowed to set negative itemId."

From our example, we know that we need three test methods for setItemId(). Two of them use good values, and the other uses a bad value.

Note, even though we have now written the tests for the accessor and mutator for itemId, we are not changing anything in the SaleItem class yet. We must write all of the tests before we begin work on the class itself.

If you look through literature online, you will see Junit handling Exceptions by using the keyword "expecting=" and the name of an Exception class on the same line as the @Test annotation. This practice has a questionable history and it has been debated. The easiest way to deal with expected Exceptions is the pattern defined above.



```

/**
 * Test of setItemId method, of class SaleItem.
 */
@org.junit.jupiter.api.Test
public void testSetItemIdBad() {
    System.out.println("setItemId bad");
    int itemId = -1;
    SaleItem instance = new SaleItem();
    int original = instance.getItemId();
    try{
        instance.setItemId(itemId);
        fail("Allowed to set an invalid itemID.");
    } catch (IllegalArgumentException iae) {
        assertEquals(original, instance.getItemId());
    }
}

```

10The test method for an invalid item ID. Note the try-catch block

### Test for the Other Constructor(s)

The generated code may not have any tests for the constructors. The logic for such tests is complex and may be beyond the ability of test generators at our level of development. There are more comprehensive testing suites in the world, but the cost tends to match the abilities provided, i.e.: they cost a bunch.

We will have to write tests for the constructors ourselves. This can take a little work; we have sixteen constructor tests for the full constructor. This is one of the reasons why we must make a list of tests before starting to write the test methods. We can use the list of tests as a checklist as we go.

Remember from our list of tests that we referred to some items with names like “goodItemId” and “goodDescription”. Now that we are writing the test methods, we can create static variables with these names and values and use them throughout the sixteen constructor test methods. That will save some time as we go.

```

public class SaleItemTest {

    /**
     * An arbitrary good value for itemId for testing constructors.
     */
    private static final int GOOD_ITEM_ID = 10;

    /**
     * An arbitrary good value for itemName for testing constructors.
     */
    private static final String GOOD_ITEM_NAME = "Item Name";

    /**
     * An arbitrary good value for description for testing constructors.
     */
    private static final String GOOD_ITEM_DESCRIPTION = "Item Description";

    /**
     * An arbitrary good value for unitPrice for testing constructors.
     */
    private static final BigDecimal GOOD_UNIT_PRICE = new BigDecimal("12.50");
}

```

11 Static variables for the good values used in the tests of the constructor

As an example, consider the constructor where we want to test the itemId of zero. In this case, we only want to test the itemId, but we still must supply a value for the other parameters. This is where we use the static variables. Those variables have arbitrarily good values, so we shouldn't have to worry about them for this test.

```

/**
 * Test of the full constructor with itemID = 0.
 */
@org.junit.jupiter.api.Test
public void testConstructorItemID0() {
    System.out.println("SaleItem() itemId 0");
    int setValue = 0;
    SaleItem instance = new SaleItem(setValue, GOOD_ITEM_NAME,
                                     GOOD_ITEM_DESCRIPTION, GOOD_UNIT_PRICE);
    int result = instance.getItemId();
    assertEquals(setValue, result);
}

```

12 A test method for the full constructor using the static variables for the parameters not under test

From here, we continue on to build all the other constructor tests. One useful trick: write the related tests together so you can take advantage of copy-paste. For example, all the constructor tests for the `itemId` will have similar constructor calls. Being able to copy that will save time and typing.

### Writing Tests for the Other Methods.

The `toString()` method requires that you know what the String value should be. Since it is based on the item name and price, it is best to explicitly set those values. This uses the full constructor so we can build the string within the test method in the same way that the `toString()` method would build it. This is acceptable in this case because tests are often written by persons who are not the final developer.

```
/**
 * Test of toString method, of class SaleItem. Expects a String in the form
 * of ITEM: {itemName} ({unitPrice})
 *
 */
@org.junit.jupiter.api.Test
public void testToString() {
    System.out.println("toString");
    String itemName = "Test";
    BigDecimal unitPrice = new BigDecimal("1.00");
    SaleItem instance = new SaleItem(GOOD_ITEM_ID, itemName
        , GOOD_ITEM_DESCRIPTION, unitPrice);
    String expResult = "ITEM: " + itemName + " ("
        + unitPrice.toPlainString() + ")";
    String result = instance.toString();
    assertEquals(expResult, result);
}
```

*13 The test method for `toString()` builds the expected result using the same logic that the `toString()` method would use*

The `compareTo()` method has four tests, so it will need four test methods. The method generator only built one, but that can be a base for the other four.

Since `compareTo()` says that it can return values less than or greater than zero, `assertEquals()` is not up to the task. The “less than zero” does not necessarily mean negative one; it means anything that is less than zero. Here we use a new JUnit method `assertTrue()`.

To use `assertTrue()`, pass a Boolean expression and a String failure message. If the expression is true, the test passes. If the expression is false, then the test fails and returns the String as the failure message.

Pay attention to the Boolean expression to make sure that your logic really matches the `compareTo()` logic.

One issue you may encounter is that your IDE may not allow you to build the test where you pass an object of the wrong class type to the `compareTo()` method. That is fine; just skip that test. If you want, you can type the test and then comment it out just to document that you tried.

```

/**
 * Test of compareTo method, of class SaleItem. instance before other
 */
@org.junit.jupiter.api.Test
public void testCompareTo() {
    System.out.println("compareTo instance before other");
    SaleItem other = new SaleItem(GOOD_ITEM_ID, "beta"
        , GOOD_ITEM_DESCRIPTION, GOOD_UNIT_PRICE);
    SaleItem instance = new SaleItem(GOOD_ITEM_ID, "alpha"
        , GOOD_ITEM_DESCRIPTION, GOOD_UNIT_PRICE);
    int result = instance.compareTo(other);
    assertTrue(result < 0, "instance before other failed");
}

```

14 One test method for compareTo() showing the use of assertEquals()

## After Building the Test Methods

Now that the test methods have been built, we can use them to write the code for the SaleItem class. To do this, we will:

- 1) Pick exactly one test method
- 2) Run the test and review the results of that one method (ignore the other for now)
- 3) Find the SaleItem method that is tested by that test method and make a change so the method will pass just that test. (Ignore the effect on related tests for now.)
- 4) Once the SaleItem method can pass that one test, pick another test of the same SaleItem method and do the same thing. On the second and subsequent test methods, you have to make sure you do not make a change to the SaleItem method that will cause it to fail a test it previously passed.
- 5) Keep doing this until the SaleItem method being tested has passed all of its tests. Then move on to another.
- 6) Repeat this process until all the tests have been passed.

It is best to start with accessor methods because they will be needed to validate other tests. If you have not tested them, then you will never know if the other tests really passed.

## Helper Items

When working on the code for SaleItem, there are some patterns that will help. First, create constants for any default values and constraint limits. For example, there will be a default item name, we want to make a constant for that. There will also be a maximum length on item name and description; we can make a constant to list those. The default value constants should be private. The constraint constants may be private or public. The public ones could be used by other software to dynamically build user interfaces with the constraints in place.

It is also useful to build validation methods. These methods take values that are intended to be stored in the attributes and makes sure that the value meets the constraints. Since validation must happen in

the mutators as well as the constructors, putting the validation logic in one place can reduce replication and possible logic errors. The validation methods should have the following characteristics:

- The methods should be private
- They are typically named "validate" and the name of the attribute, i.e.: validateItemName()
- They take the value to be validated as an argument: validateItemName(String itemName)
- They have a return type of void
- If the value is not valid, the validation method must throw an Exception, usually an IllegalArgumentException
- The validation method only validates, it never changes anything inside the class. In particular, it does not change the attribute related to the validator!
- It is permissible, but not required, to place all the validation methods inside a utility class such as SaleItemValidator. If they are in a separate class, each method must be public and static.

### Testing and Building getItemId()

When we first run the tests, the test for getItemId() fails with the message "Failed: Not supported yet." That's obviously because the getItemId() method has the UnsupportedOperationException as its only code. We also know that the getItemId() should only return whatever is in the itemId() attribute. With that in mind, we make the change to SaleItem.getItemId() to do the return.

```
/**
 * The ID of the sale item. An integer not less than zero.
 *
 * @return the current item ID
 */
public int getItemId() {
    return itemId;
}
```

*15 The updated getItemId() method*

When we run the test again, we still get "Failed: Not supported yet." What's the issue?

Remember that the test method attempts to instantiate the SaleItem using the no-argument constructor. Unfortunately, the no-argument constructor also throws an UnsupportedOperationException. We need to fix that. Remember, we only do enough to pass the test, so we will only remove the Exception and run the test again. We may need to return to this constructor later, but that is okay.

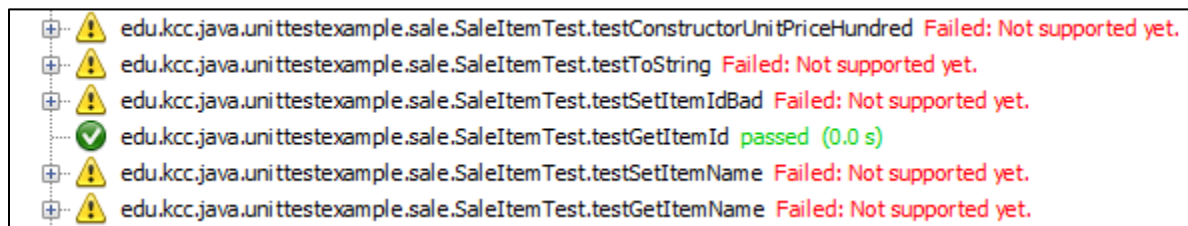
```

/**
 * The default constructor uses default values.
 */
public SaleItem() {
}

```

16 The no-argument constructor without the UnsupportedOperationException and no other code

Now, though the overall test of the class fails, the test for getItemId() has passed. You may have to search through the results, but you should find the green circle with a checkmark and the green text stating “passed”.



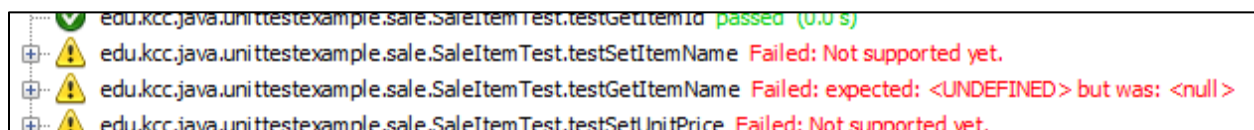
17 The test results showing that the test for getItemId() has passed

## Testing and Building getItemName()

This test has a little more to it. The test for getItemName() specifies some default text. This will show up when we work with the no-argument constructor.

To start, run the test and then alter the getItemName() method to return the itemName attribute. The test will still fail, but with a new failure message. The new message says, “Failed: expected <UNDEFINED> but was: <null>”.

In the test method, we defined the String expectedResult to be “UNDEFINED”. That’s the expected part in the failure message. What getItemName() returned was null. That’s because the no-argument constructor does not set a value for the itemName attribute. The itemName is a String, so its default value is null.



18 The results of the test for getItemName()

The fix for this is to update the no-argument constructor. This is also when we should create a constant for the default item name. If we don’t create a constant then the value used in the constructor would be a “magic value” meaning that it “just appeared”. The constant is both static and final. This means no instance of the class can change the value (final) but all instances of the class can access the single copy of this constant (static).

```

/**
 * A default value for itemName when one is not supplied.
 */
private static final String DEFAULT_ITEM_NAME = "UNDEFINED";

```

19 A constant with the value for itemName to use when none is supplied

With the constant in place, the no-argument constructor can use it to assign a value to itemName. Then the tests for getItemName() should pass.

```

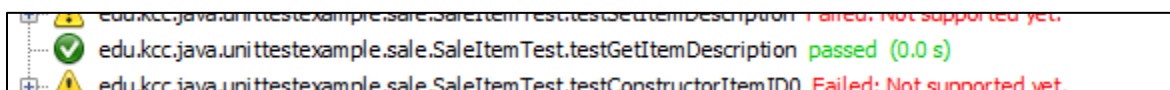
/**
 * The default constructor uses default values.
 */
public SaleItem() {
    itemName = DEFAULT_ITEM_NAME;
}

```

20 The no-argument constructor uses the constant to assign a value to itemName

### Testing and Building for getItemDescription()

The test for getItemDescription() works in the same way as the getItemName() worked. You add a constant to represent the default value. Then you alter the constructor to set the attribute to the default value. When you alter getItemDescription() to return the itemDescription, the test should pass.



21 The test for getItemDescription should pass

### Testing and Building for getUnitPrice()

By now, there should be a pattern emerging. In this case, the constant must be an instance of BigDecimal. The no-argument constructor assigns the constant value to the attribute. The accessor returns the value of the attribute and things are ready to test again.

```

/**
 * A default value for unitPrice when one if not supplied.
 */
private static final BigDecimal DEFAULT_UNIT_PRICE = new BigDecimal("0.00");

```

22 The constant for the unitPrice

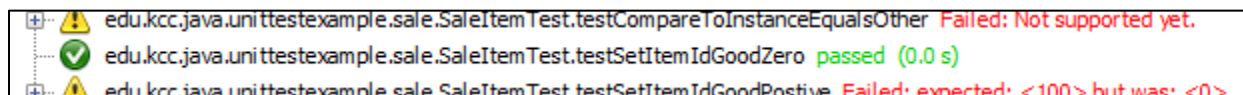
## Testing and Building for setItemId()

There are three test methods for setItemId(). Two use allowed values and one uses disallowed values. Like the other methods in SaleItem, setItemId() throws an UnsupportedOperationException. Start by removing that.

```
/**
 * The ID of the sale item. An integer not less than zero.
 *
 * @param itemId the ID to set
 */
public void setItemId(int itemId) {
}
```

23 The setItemId() method after removing the UnsupportedOperationException

Removing the UnsupportedOperationException lets one of the tests pass. This is the test where we set the itemId to zero. This test is deceptive, though. The default value for the itemId is zero. In the test, we check if setting worked by comparing the expected value to the value returned by getItemId(). The accessor returns the default value, which just happens to be the same as the expected value. Always watch for this when testing.



24 The test for setItemId() with a zero value passes

Next, we will look at the other test with a valid value. In this case, we are setting the itemId to 100. That is valid, so we must change the setter to work. This is why we test with an arbitrary good value. The test with zero passed, but it did not tell the whole story. An arbitrary good value not passing tells even more about what is missing on the method.

We add the code to the setItemId() method and the test should pass.



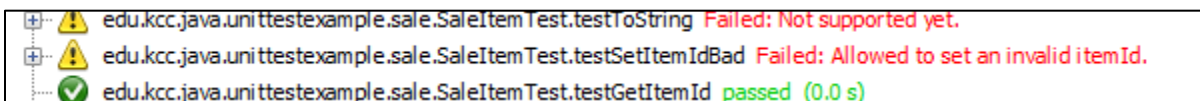
```

/**
 * The ID of the sale item. An integer not less than zero.
 *
 * @param itemId the ID to set
 */
public void setItemId(int itemId) {
    this.itemId = itemId;
}

```

25 The `setItemId()` method after adding code to assign the parameter value to the attribute

One side effect of this code change is that the failure message has changed for the test of `setItemId()` using a negative number. The message now reads, “Failed: Allowed to set an invalid itemId”. This is the failure message we defined ourselves in the test method. In the test method, we perform the call to `setItemId()`, with a negative value, inside a try-block. When the things work correctly, this call should result in an `IllegalArgumentException`. Because it did not, the next statement called the `fail()` method.



```

edu.kcc.java.unittestexample.sale.SaleItemTest.testToString Failed: Not supported yet.
edu.kcc.java.unittestexample.sale.SaleItemTest.testSetItemIdBad Failed: Allowed to set an invalid itemId.
edu.kcc.java.unittestexample.sale.SaleItemTest.testGetItemId passed (0.0 s)

```

26 The failure message for `setItemId()` using a negative value

To remedy this, we must apply validation to the `SaleItem` class to prevent negative values. Since we will need this same validation of `itemId` in both the mutator and the full constructor, we will put the validation logic into a validation method (see above for a description).

The logic checks to see if the supplied `itemId` is less than zero. If it is, the method throws a new `IllegalArgumentException` with an appropriate error message.

```

/**
 * A validator method for itemId. An integer not less than zero. Throws
 * an IllegalArgumentException if the supplied value is not valid.
 *
 * @param itemId the value to test for validity
 */
private void validateItemId(int itemId) {
    if(itemId < 0){
        throw new IllegalArgumentException(
            "The item ID cannot be less than zero");
    }
}

```

27 The `itemId` validator method from the `SaleItem` class

Now that we have the validator method, we can call it inside the mutator. There is a critical detail here: the call to the validator must happen before any other action inside the setter method!

The code in the image below is wrong. In this version of the method, the attribute is changed before the new value is tested. That means that the attribute gets the new value regardless of whether the value is valid.

```
/**
 * The ID of the sale item. An integer not less than zero.
 *
 * @param itemId the ID to set
 */
public void setItemId(int itemId) {
    this.itemId = itemId;
    validateItemId(itemId);
}
```

28 An *incorrect* application of the `validateItemId()` method inside `setItemId()`

Fortunately, the test still fails. This time it has a new message, “Failed: expected<0> but was: <-1>”. This is because the setter was allowed to assign the value to the attribute **before** testing. The test method retrieved the value of the `itemId` before attempting to set the invalid value, and then again at the end of the method. The two values did not match, which is why the test failed.

In the correct version of `setItemId()` calls the validator method first. If the value is allowed, then the mutator will assign the value to the attribute. However, if the validator throws an Exception, then `setItemId()` will also throw the Exception before making any attempt to change the attribute value. This will allow the test to pass.

```
/**
 * The ID of the sale item. An integer not less than zero.
 *
 * @param itemId the ID to set
 */
public void setItemId(int itemId) {
    validateItemId(itemId);
    this.itemId = itemId;
}
```

29 The *correct* usage of `validateItemId()` inside `setItemId()`

## Testing and Building the other Mutators

The other mutators, `setItemName()`, `setItemDescription()` and `setUnitPrice()`, follow the same pattern as `setItemName()`.

- Remove the `UnsupportedOperationException` from the setter
- Add the logic to assign a value to the attribute
- Create the validator method inside `SaleItem`
- Call the validator method as the first operation inside the mutator method

Remember to do these things one at a time, focusing on one set of tests until they all pass. Only then should you move to the next mutator method.

It is also a good time to remember that some constraints should be defined as constants so you can avoid “magic numbers”. If you make them public, the classes outside of `SaleItem` can read them and possibly use them to perform pre-validation such as on a user interface. Making them public is optional; they could also be private.

```
/**
 * The maximum length for an itemName.
 */
public static final int MAX_ITEM_NAME_LENGTH = 256;

/**
 * The maximum length for a description.
 */
public static final int MAX_ITEM_DESCRIPTION_LENGTH = 1024;
```

*30 The constants used to validate the maximum lengths of String values for the attributes*

When checking reference type variables, be sure to check for null before calling any methods to check values. For example, when validating the `itemName`, a required String cannot be null. A required String also cannot be blank. If we call the `isBlank()` method on a null reference, we will get an unexpected Exception.

For `itemName`, we will perform our checks in the following order:

- 1) Does null equal the `itemName`? If so, throw an Exception.
- 2) Is the `itemName` blank (which is different from, but includes empty)? If so, throw an exception.
- 3) Is the length greater than the maximum allowed length? If so, throw an exception.

Two things to note. First, each test throws its own Exception. This allows for each to have its own error message and prevents the following test from happening. Second, there is no final else in the structure. At that point, the `itemName` is valid, so there is nothing left to do.

The validator for the description and unit price are similar.

```

/**
 * A validator for itemName.  A required String not longer than 256
 * characters.
 *
 * @param itemName the itemName to validate
 */
private void validateItemName(String itemName){
    if(null == itemName){
        throw new IllegalArgumentException(
            "The item name cannot be null.");
    } else if(itemName.isBlank()){
        throw new IllegalArgumentException(
            "The item name cannot be blank.");
    } else if(itemName.length() > MAX_ITEM_NAME_LENGTH){
        throw new IllegalArgumentException(
            "The item name cannot be more than "
            + MAX_ITEM_NAME_LENGTH
            + " characters in length.");
    }
}
}

```

31 The validateItemName() method in the SaleItem class

Sometimes, the validation requires more complexity if object being validated does not have an obvious comparison. The unitPrice is a BigDecimal object. Once we have tested to be sure it is not null, we have to see if its value is less than 0.00. Because BigDecimal is a class, the regular less-than and greater-than operators do not work. However, the BigDecimal class implements the Comparable interface, so we can use the compareTo() method in our test.

First, we create an instance of BigDecimal with a value of 0.00. Then we call the compareTo() method on that new instance and pass it the unitPrice as an argument. If the unitPrice is negative, it is less than the new instance. The compareTo() method will then return a number greater than zero. We just test for that condition and throw an Exception if the value is greater than zero.

```

/**
 * A validator for unitPrice. A non-negative decimal number to two decimal
 * places.
 *
 * @param unitPrice the unitPrice to validate
 */
private void validateUnitPrice(BigDecimal unitPrice){
    if(null == unitPrice){
        throw new IllegalArgumentException(
            "The unit price cannot be null.");
    }
    if(0 < new BigDecimal("0.00").compareTo(unitPrice)){
        throw new IllegalArgumentException(
            "The unit price cannot be negative.");
    }
}

```

*32 The validation method for unitPrice using the compareTo() method of BigDecimal*

## Testing and Building the Full Constructor

The full constructor has many tests. The good news is that the process will go smoothly. For each test, the constructor will need to assign the argument value to the related attribute. This will allow the tests with good values to pass. You will still need to add calls to the validation methods to get the other tests to pass. Work with one parameter at a time until all of its tests pass. Then move to the next.

```

/**
 * The full constructor requires a value for all the attributes.
 *
 * @param itemId the ID of the item, not less than zero
 * @param itemName the name of the item, from 1 to 256 characters
 * @param description the description of the item from 1 to 1024 characters
 * @param unitPrice the unit price of the item, not less than zero
 */
public SaleItem(int itemId, String itemName, String description
    , BigDecimal unitPrice) {
    validateItemId(itemId);
    validateItemName(itemName);
    validateItemDescription(description);
    validateUnitPrice(unitPrice);
    this.itemId = itemId;
    this.itemName = itemName;
    this.description = description;
    this.unitPrice = unitPrice;
}

```

33 The full constructor once all the code has been added

## Testing and Building toString()

The toString() method is fairly simple to test. When we built the test, we created two values, an itemName and a unitPrice. Then we created an instance using the full constructor and passed those two values in along with the default good values for the other parameters. Then we built a String that should match the desired output.

In the SaleItem class, we just build the same String, but using the object's attributes and then return it. You may even be able to adapt the code from the test itself. Since the toString() method only returns values that are already in the object, there is no validation required.

```

/**
 * An override of the toString() method.
 *
 * @return ITEM: {itemName} ({unitPrice})
 */
@Override
public String toString() {
    return "ITEM: " + itemName + " (" + unitPrice.toPlainString() + ")";
}

```

34 The completed toString() method in SaleItem

## Testing and Building compareTo()

The compareTo() method has three tests based on sort order possibilities. The natural sort order for SaleItem was alphabetical order by itemName. The good news is that itemName is a String and the String class has an implementation of compareTo() that sorts alphabetically.

This is an important concept when building your classes. Try to see if the classes from which your class is built already has functionality you need. That code reuse is one of the reasons composition is so critical to object oriented programming.

To use the String version of compareTo(), go to the SaleItem version of compareTo(). There, return whatever comes from this.itemName.compareTo(other.getItemName()). This allows the itemName String version of compareTo() to generate a value; the SaleItem just returns it.

**Composition is when one class is built, partially or wholly, from objects of other classes. This allows one class to make use of the functionality of other classes. If any of the class' attributes are reference types, there is an example of**

```
/**
 * An implementation of compareTo() from the Comparable interface. Used
 * to sort SaleItem objects by name in alphabetical order.
 *
 * @param other the other SaleItem for comparing
 * @return less than zero, zero, or greater than zero
 */
@Override
public int compareTo(SaleItem other) {
    return this.itemName.compareTo(other.getItemName());
}
```

35 The completed compareTo() method

## Completed

At this time, the tests are all complete and so is the SaleItem class. If done properly, this means:

- The class is considered tested and reliable, so anyone can use it without wondering about the insides.
- There should be no code that was not required to pass a test. If true, then there is no untested code lurking inside to become an issue later.
- The tests are all documented. The documentation of the tests may be necessary to meet the requirements of a contract or to prove the testing in the case of litigation or other challenge.
- You should have a full understanding of what the class is and what to expect from its public methods.