Question

- How do we know this method is correct?

```
static int sumArray(int[] a)
{
    int s = 0;
    for (int x : a)
        s += x;
    return s;
}
```

IT 179
12
Testing and Debugging

Have you read Chapter 3 (Testing & Debugging)?

Not at all

A few pages

Most of it

All of it

Chapter Objectives

- To understand different testing strategies
- To learn to test using the JUnit test framework

Types of Testing

Section 3.1

Types of Testing

- Testing is exercising a program under controlled conditions.

Why do we need program testing?

Types of Testing

- Testing is exercising a program under controlled conditions.
- More thorough testing increases the likelihood of finding defects.
- However, in a complex program, no amount of testing can guarantee the absence of defects.

Levels of Testing

- Unit testing
  - Tests the smallest testable pieces of the software
  - In OOD, this may be a class or a method
- Integration testing
  - Tests interaction among units
  - If the unit is a method, this tests the interaction of methods within a class
  - More commonly, tests the interaction between several classes
- System testing
  - checks whether the software meets the specified functional requirements or not.
- Acceptance testing
  - checks whether the software meets the customer requirements or not

Types of Testing

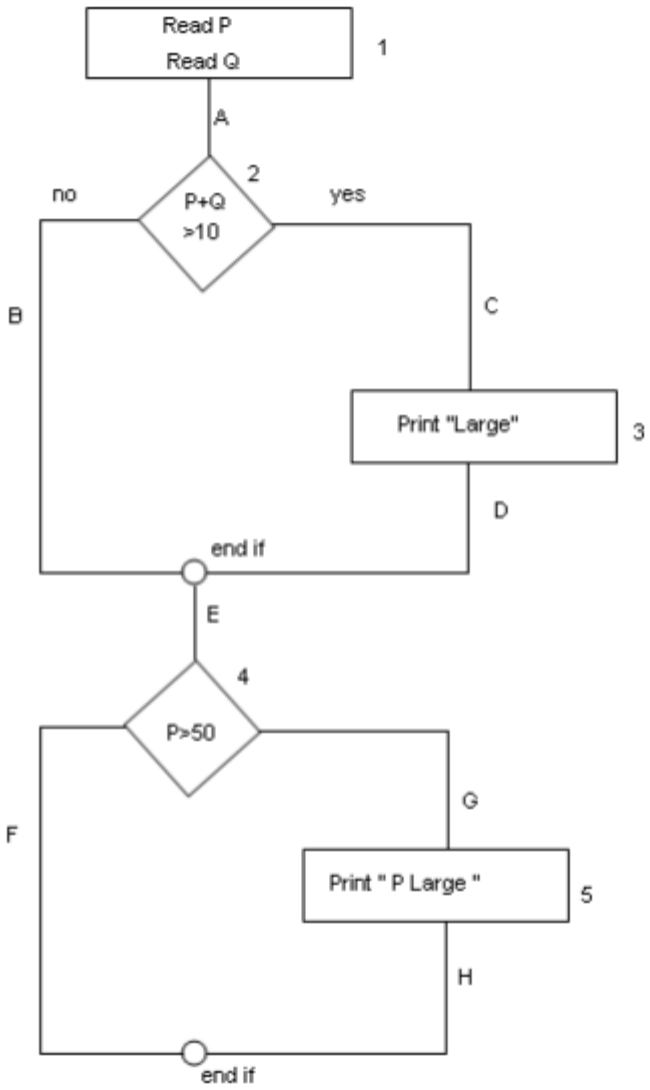
- Black-box testing
  - Tests the item based on its interfaces and functional requirements
  - Input values are varied over allowable ranges and outputs compared to independently calculated values.
  - Input values outside of allowed ranges are also tested to see if the unit responds according to specifications

Types of Testing (cont.)

- White-box testing
  - Tests the unit with knowledge of its internal structure
  - Attempts to exercise as many paths through the unit as possible
  - Statement coverage ensures that each statement is executed at least once
  - Branch coverage ensures that every choice at each branch is tested
  - Path coverage tests each path through a method

Example

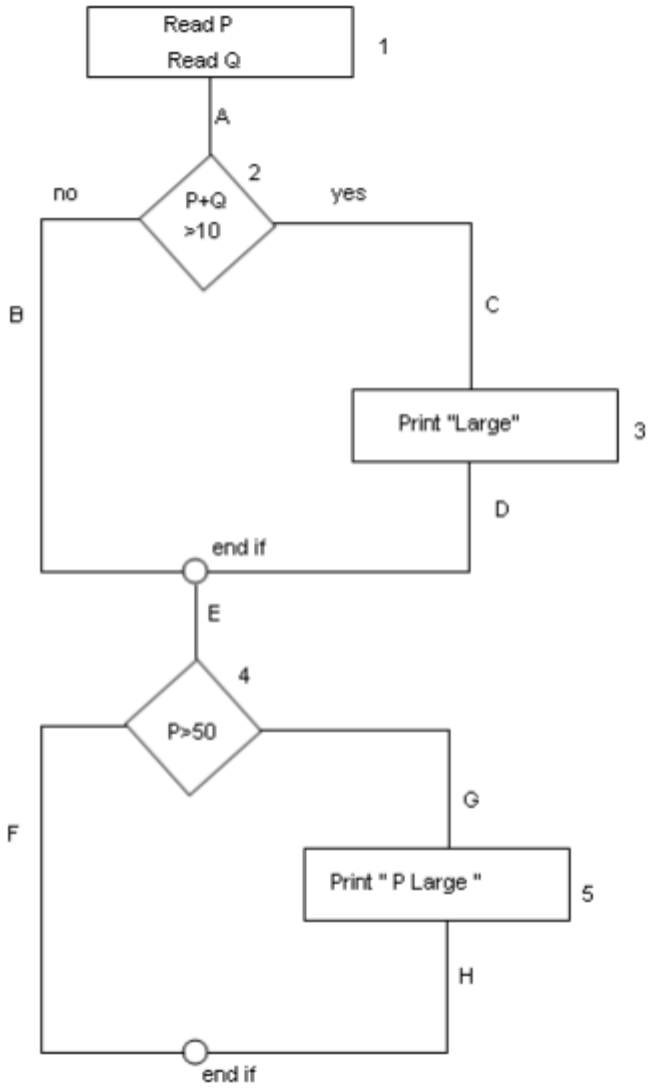

```
Read P
Read Q
IF P+Q > 100 THE
Print “Large”
ENDIF
If P > 50 THEN
Print “P Large”
ENDIF
```



Example

Statement Coverage (SC):

- Find out the shortest number of paths following which all the nodes will be covered.
- By traversing through path 1A-2C-3D-E-4G-5H all the nodes are covered.
- By traveling through only one path, all the nodes 12345 are covered
- So, statement coverage in this case is 1.



Example

Branch Coverage (BC):

Find out the minimum number of paths which will ensure covering of all the edges.

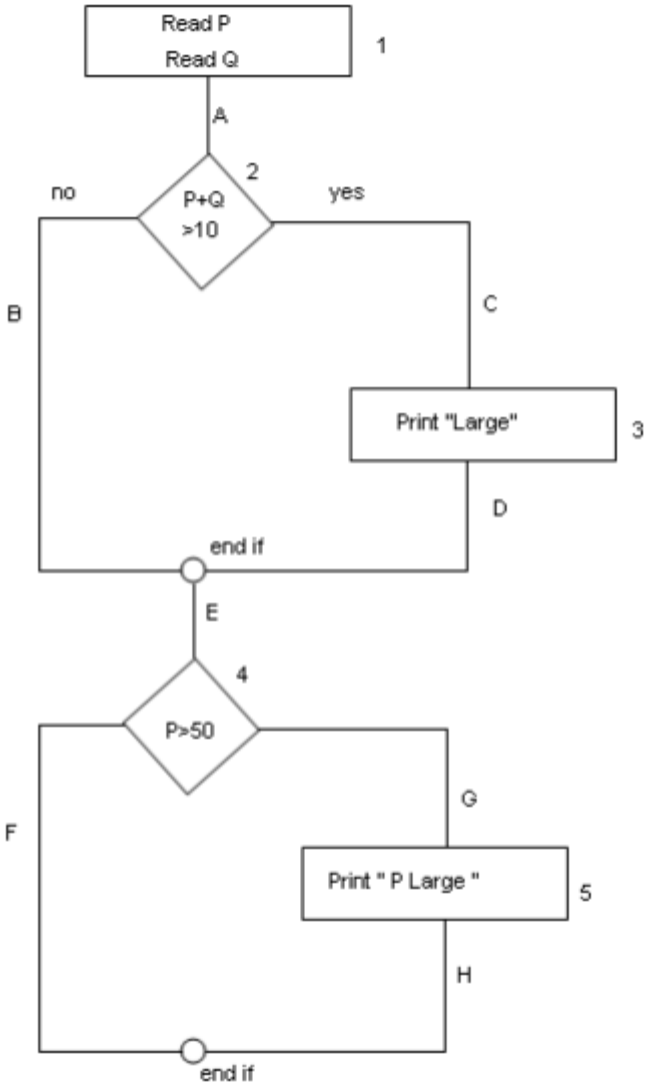
In this case, no single path will cover all the edges at one go.

By following paths 1A-2C-3D-E-4G-5H, maximum numbers of edges (A, C, D, E, G an H) are covered but edges B and F are left.

To covers these edges we can follow 1A-2B-E 4F.

By combining the above two paths, we can ensure of traveling through all the paths.

Branch Coverage = 2.



Example

Path Coverage (PC):

Path Coverage ensures covering of all the paths from start to end.

All possible paths are:

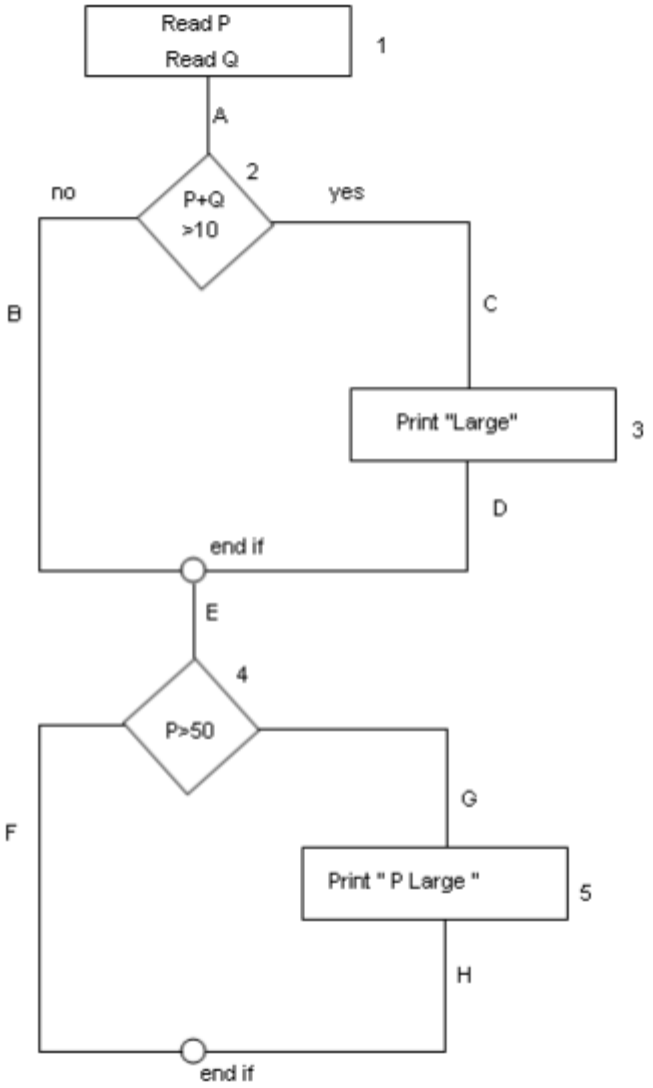
1A-2B-E-4F

1A-2B-E-4G-5H

1A-2C-3D-E-4G-5H

1A-2C-3D-E-4F

Path coverage is 4.



Testing All Paths

- We want to test all of the paths of the method:

```
public void testMethod(char a, char b) {
  if (a < 'M') {
    if (b < 'X') {
      System.out.println("path 1");
      ...
    } else {
      System.out.println("path 2");
      ...
    }
  } else {
    if (b < 'C') {
      System.out.println("path 3");
      ...
    } else {
      System.out.println("path 4");
      ...
    }
  }
}
```

Table 3.1 Testing all Paths (cont.)

- The following table shows possible input values to exercise all four possible paths:

a	b	Message
'A'	'A'	path 1
'A'	'Z'	path 2
'Z'	'A'	path 3
'Z'	'Z'	path 4

- These are the smallest and largest allowable values
- A more complete test should use additional valid combinations of values and with non-letter values

Preparations for Testing

- Planning for testing should begin early and include consideration of:
  - How will the program be tested?
  - When will it be tested?
  - By whom will it be tested?
  - What test data will be used?
- Early planning can help programmers prepare for testing as they write their code.
  - For instance, validating input data and throwing appropriate exceptions.

Testing Tips

- Document all class attributes and method parameters using comments.
- Trace execution by displaying each method name as it is entered.
- Display values of all input parameters as a method is entered, and also any class attributes used.
- After a method returns, display all its outputs including its return value and the values of any class attributes it modified.

Testing Tips (cont.)

- It is useful to include code like

```
if (TESTING) {
   //code that you wish to "remove" after testing
}
```

- Then you can add the following to your class when you want to enable testing

```
private static final boolean TESTING = true;
```

- And change it when you want to disable testing

```
private static final boolean TESTING = false;
```

Specifying the Tests

Section 3.2

Specifying the Tests—General Principles

- Black-box testing
  - Test all expected input values
  - Test unexpected input values
  - Specify anticipated results of each set of values tested

Specifying the Tests—General Principles (cont.)

- White-box testing
  - Exercise every branch of every if statement
  - Test switch statements for all valid selector values and some invalid values
  - Loops—test behavior if
    - The body is never executed
    - The body is executed once
    - The body is executed the maximum number of times
  - Assure that loops eventually will terminate

Boundary Conditions

- Boundary conditions are special cases that should be explicitly tested.
- For instance, in a method designed to find a specific value within an array, you would test cases where
  - The target is the first element in the array
  - The target is the last element in the array
  - The target is somewhere in the middle of the array
  - The target is not in the array

Demo in a minute:

```
static int findValueInArray(int x, int[] a)
```

Boundary Conditions (cont.)

- More boundary conditions for a method that finds a specific target value in an array
  - There is more than one occurrence of the target value
  - The array has but one element and it is not the target
  - The array has but one element and it is the target
  - The array has no elements

Testing Using the JUnit Platform

Section 3.4

The JUnit5 Test Framework

- A test class is a class that contains one or more test methods that test a single method or a class.
  - It provides known inputs for a series of tests
  - It compares the results of each test with known results and reports whether the test was passed or failed
- A test framework is a software product that facilitates writing and running test classes.
- We will demonstrate how to use the JUnit5 test platform next.

Example


```java
static int sumArray(int[] a)
{
    int s = 0;
    for (int x : a)
        s += x;
    return s;
}

@Test
public void testSumArray()
{
    int t1[] = { 2, 4, 1, 1, 51 };
    int t2[] = { 5, 1, 3, 1, 7, 3 };

    assertEquals(59, sumArray(t1));
    assertEquals(20, sumArray(t2));
}
```

The JUnit5 Test Framework

- Example: **Boundary Conditions**

```
Demo: static int findValueInArray(int x, int[] a)
```

Example: Boundary Conditions


```
static int findValueInArray(int x, int[] a)
{
    if (a.length == 0)
        return -1;
    for (int i = 0; i < a.length; i++)
        if (a[i] == x)
            return i;
    return -1;
}

@Test
public void testfindValueInArray()
{
    assertEquals(-1, findValueInArray(19, t1));
    assertEquals(0, findValueInArray(2, t1));
    assertEquals(5, findValueInArray(9, t1));
    assertEquals(t1.length - 1, findValueInArray(32, t1));
}
```

Imports needed for JUnit5

- Each test class in JUnit5 begins with two import statements:

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

- These allow us to use JUnit's **`assert`** methods
- The **`assert`** methods allow us to specify pass/fail behavior for tests.
- They are summarized in Table 3.2 in the book as shown on the next 2 slides.

Table 3.2 Methods of org.junit.jupiter.api.Assertions

Method	Parameters	Description
assertArrayEquals	*expected, actual[,message]*	Tests whether the contents of the two array parameters *expected* and *actual* are equal. This method is overloaded for arrays of the primitive types and Object. Arrays of Objects are tested with the equals method applied to the corresponding elements. The test fails if an unequal pair is found, and an AssertionError is thrown. If the optional *message* is included, the AssertionError is thrown with this *message* followed by the default message; otherwise it is thrown with just a default message
assertEquals	*expected, actual[, message]*	Tests whether *expected* and *actual* are equal.
assertFalse	*condition[, message]*	Tests whether the *condition* is false
assertNotEquals	*expected, actual[, message]*	Tests whether *expected* and *actual* are not equal. This method is overloaded for the primitive types and Object. To test Objects, the equals method is used.
assertNotNull	*object[, message]*	Tests whether the *object* is not null

Table 3.2 (cont.)

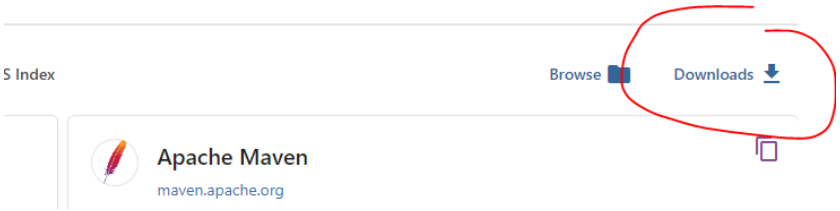
Method	Parameters	Description
assertNotSame	*expected, actual[, message]*	Tests whether *expected* and *actual* are not the same object. (Applies the != operator.)
assertNul1	*object[, message]*	Tests whether the *object* is null
assertSame	*expected, actual[, message]*	Tests to see whether *expected* and *actual* are the same object.
assertThrows	*expected, executable [, message]*	Tests whether the code in the *executable* block throws the expected *exception*.
assertTrue	*condition[, message]*	Tests whether the *condition* is true
fail	*[message]*	Always throws AssertionError

Downloading JUnit

- Go to:

https://search.maven.org/artifact/org.junit.platform/junit-platform-console-standalone/1.7.0-M1/jar

- Click on Download



- Downloaded .jar file is:

junit-platform-console-standalone-1.7.0-M1

JUnit5 Example

- We will create a JUnit5 test class called `ArraySearchTest` to test method search in class `ArraySearch`.
- We wish to test the following:
  - The target is the first element in the array
  - The target is the last element in the array
  - The target is somewhere in the middle
  - The target is not in the array
  - There is more than one occurrence of the target and we find the first

JUnit5 Example (cont.)

- More `ArraySearch.search` tests
  - The array has only one element and it is the NOT the target
  - The array has only one element and it is the target
  - The array has no elements

JUnit5 Example (cont.)

- Test Class `ArraySearchTest` begins with the lines shown below
- The private data field `x` is the common array used for most of the tests

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
/**
* JUnit test of ArraySearch.search
*/
public class ArraySearchTest {
    // Common array for test methods
    private final int[] x = {5, 12, 15, 4, 8, 12, 7};
```

JUnit5 Example (cont.)

- Testing the case where the target is the first element

```
@Test
public void firstElementTest() {
    // Test for target as first element.
    assertEquals("5 not at position 0",
            0, ArraySearch.search(x, 5));
}
```

- The `assertEquals` method specifies the message to print on failure, the expected result (0), and the method call to test. We expect a return value of 0 because 5 is indeed the first array element.

JUnit5 Example (cont.)

- Testing the case where the target is the last element

```
@Test
public void lastElementTest() {
    // Test for target as last element.
    assertEquals("7 not at position 6",
            6, ArraySearch.search(x, 7));
}
```

- In this case, the target value was 7, and we expect to find it in location 6 (the last element of the array)

JUnit5 Example (cont.)

- Testing the case where the target is somewhere in the middle:

```
@Test
public void inMiddleTest() {
    // Test for target somewhere in middle.
    assertEquals("4 is not found at position 3",
            3, ArraySearch.search(x, 4));
}
```

- Here, the target value was 4 and we expect to find it in location 3.

JUnit5 Example (cont.)

- Testing the case where the target is not in the array

```
@Test
public void notInArrayTest() {
    // Test for target not in array.
    assertEquals(-1, ArraySearch.search(x, -5));
}
```

- Here, the target value was -5 and we expect a return value of -1 indicating “not found.”
- The first parameter to `assertEquals` is omitted which would result in a default failure message.

JUnit5 Example (cont.)

- Testing the case where the target is present in multiple locations, we find the first

```
@Test
public void multipleOccurencesTest() {
    // Test for multiple occurrences of target.
    assertEquals(1, ArraySearch.search(x, 12));
}
```

- The target is 12, which occurs at locations 1 and 5. We expect the program to return 1.

JUnit5 Example (cont.)

- Testing a one-element array which does contain the target. The array, `y`, being tested in declared in the method.

```
@Test
public void oneElementArrayTestItemPresent() {
    // Test for one-element array
    int[] y = {10};
    assertEquals(0, ArraySearch.search(y, 10));
}
```

- We expect to find the 10 in location 0.

JUnit5 Example (cont.)

- Testing a one-element array which does not contain the target value

```
@Test
public void oneElementArrayTestItemAbsent() {
    // Test for 1-element array
    int[] y = {10};
    assertEquals(-1, ArraySearch.search(y, -10));
}
```

- Array `y` does not contain -10, so we expect a return value of -1 meaning “not found.”

JUnit5 Example (cont.)

- Testing with an empty array

```
@Test
public void emptyArrayTest() {
    // Test for an empty array
    int[] y = new int[0];
    assertEquals(-1, ArraySearch.search(y, 10));
}
```

- Array `y` does not contain anything, so we expect a return value of -1 meaning “not found.”

JUnit5 Example (cont.)

- The last test case is testing for a null array **y**

```
@Test
public void nullArrayTest() {
    int[] y = null;
    assertThrows(NullPointerException.class,
        ()-> ArraySearch.search(y, 10));
}
```

- The **assertThrows** method states that we expect method search to throw a **NullPointerException** when it is applied to a null array.
- The last line is a lambda expression which causes method search to execute on a null array.

JUnit5 Example (cont.)

- The line shown below is a lambda expression which causes method search to execute on null array y.

```
()-> ArraySearch.search(y, 10));
```

- We discuss lambda expressions in detail in Section 6.4, but we will introduce them as arguments passed to method **`assertThrows`**.
- This statement creates an anonymous instance of class **`ArraySearch`** and applies static method **`search`** to this object, passing y and 10 as its arguments.
- Since y is null, the expected result is that **`search`** will throw a **`NullPointerException`**.

JUnit5 Example (cont.)

- The test results are shown in the window below. The check marks indicate that all tests passed as expected.