

Introduction to Software Engineering (ISAD1000)

Lecture 8: White-Box Testing & Test Fixtures

Updated: 16th February, 2022

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Copyright © 2020, Curtin University
CRICOS Provide Code: 00301J

Outline

[Inputs and Outputs](#)

[Files](#)

[Exceptions](#)

[Test Fixtures](#)

[White Box Testing](#)

Testing So Far

- ▶ So far, our discussion of testing has focused on parameters and return values.
- ▶ We see the range of possible parameter values, and divide (partition) them into categories.
 - ▶ Equivalence partitioning and boundary value analysis.
 - ▶ But we *haven't* looked *inside* the method/function. We've treated it as a black box.
- ▶ For the return values, we obtain them the actual value, and compare to what we expected.
 - ▶ We've *assumed* that methods/functions are like mathematical functions: transforming one or more parameters into a single return value.

Generalising

- ▶ Unit testing can get more complicated.
- ▶ We *do still* have the following basic ingredients:
 - ▶ Test cases;
 - ▶ For each test case: test data, production code call, expected results, and actual results.
- ▶ But:
 - ▶ We can also do “white-box” testing, where we find test cases based on the code.
 - ▶ In either white- or black-box testing, we sometimes need additional “setting up” for each test case.
 - ▶ Some test data are *not* provided via parameters.
 - ▶ Some test results are *not* obtained via return values.
 - ▶ There can be multiple results per test case, and hence multiple assertions.

Input (Sources of Data)

- ▶ Methods/functions can get data from various sources:
 - ▶ Parameter values;
 - ▶ User input;
 - ▶ Fields (variables outside the method/function);
 - ▶ Files on disk;
 - ▶ Databases;
 - ▶ The operating system;
 - ▶ Other running programs on the same computer;
 - ▶ Online services.
- ▶ Testing must account for *all* the data a method/function uses.
- ▶ Test code must control and mimick these sources of data.
 - ▶ e.g. We need to fool the production code into accepting fake “user input” that really comes from the test code.

Output (Actions)

- ▶ Methods/functions can also send data to many different places:
 - ▶ Return value;
 - ▶ Exceptions;
 - ▶ The screen/console;
 - ▶ Fields;
 - ▶ Files;
 - ▶ Databases;
 - ▶ Other running programs;
 - ▶ Online services.
- ▶ Testing must check that all of these are as expected.
 - ▶ e.g., if you're writing a file, your test code must ensure that actually happens properly.
- ▶ We need to capture these outputs and run them all through assertion statements.

} Called “*side effects*”, though they're often the main purpose of a method/function.

Input/Output is String-Based

- ▶ We'll consider how to test methods/functions that:
 - ▶ Take user input (console input);
 - ▶ Display output (console output);
 - ▶ Read text files (file input);
 - ▶ Write text files (file output);
 - ▶ Generate exceptions.
- ▶ The first four of these have something in common:
- ▶ Everything is a string!
 - ▶ Reading console input? The user enters *a string*.
 - ▶ If you ask for an integer, the user enters a string containing digit characters. Then you calculate what integer they form.
 - ▶ Writing real numbers to a file? You're writing a string.
 - ▶ You take each number, and generate a string that contains various digit characters and a ".".

New Lines (“\n”) in Input/Output

- ▶ The string-iness of input/output goes a bit further.
- ▶ You can think of *all* console input as just *one big string* put together.
 - ▶ And, similarly, all console output is another big string.
 - ▶ And an entire text file is really just a single string.
- ▶ To make sense of this, you need to understand the “newline” character “\n”.
 - ▶ It’s written backslash-n, though it doesn’t really *look* like anything.
 - ▶ But it can be part of any string, just like a letter or digit.
 - ▶ It represents a line break; a point at which one line of text ends and another begins.
 - ▶ It’s the ENTER key, when pressed by the user.

New Lines (“\n”) in Input/Output

- ▶ The newline character “\n” lets us think of multiple inputs or outputs as all being one string.
- ▶ Say you write a program that asks the user some questions, and they answer like this:

```
Enter an integer: 50
Enter another integer: -71
Enter a word: calculator
```

- ▶ You can represent the input as the string
“50\n-71\ncalculator\n”.
- ▶ That’s the string the user has actually typed in.
- ▶ An aside: to represent an actual backslash, you must write “\\”. (For two backslashes, write “\\\\”, etc.)

GUIs

- ▶ Another aside. . .
- ▶ For programs that display *windows, buttons, scroll-bars*, etc., and respond to mouse clicks or touch-based input, these things are *not* strings.
- ▶ Testing this is outside the scope of this unit.
- ▶ It can be quite tricky:
 - ▶ How many different ways are there to move the mouse?
 - ▶ How many different positions on the screen can windows and buttons occupy?
 - ▶ Nonetheless, there are mechanisms to automate GUI input; e.g. `java.awt.Robot`.

Simulating User Input

- ▶ To test a method/function that takes console input:
 - ▶ Our test data will be a string representing that input.
 - ▶ We have to set things up so that the production code *thinks* this string is actual input.
- ▶ And remember: the production code still has to work as normal.
 - ▶ We can't change it to make this work.
 - ▶ Everything we do to test it must be in the test code.
- ▶ Fortunately, languages like Java and Python provide a couple of tricks to help...

Simulating Console Input

```
import java.io.*; // Test code:
...
String simInput = "abc";
System.setIn(new ByteArrayInputStream(
    simInput.getBytes()));
... // call production code
```



- ▶ We decide what the simulated input should be (e.g. "abc")
- ▶ We create an "object" that behaves like a source of input, but actually just gives you back what you put into it.

 **Java** "ByteArrayInputStream"
 **Python** "io.StringIO"

- ▶ We tell the system to read from this, instead of the console.


Simulating Console Input

```
import sys, io # Test code:
...
simInput = "123"
sys.stdin = io.StringIO(simInput)
... # call production code
```



- ▶ We decide what the simulated input should be (e.g. "abc")
- ▶ We create an "object" that behaves like a source of input, but actually just gives you back what you put into it.

 **Java** "ByteArrayInputStream"

 **Python** "io.StringIO"


- ▶ We tell the system to read from this, instead of the console.


Using Simulated Input

- ▶ Say your test code sets up some simulated input; specifically, a value of "abc".

- ▶ Your test code then calls the production code.

- ▶ The production code tries to read some console input.

Java:  `String val = scanner.nextLine();`


 Python: `val = input()`

- ▶ The production code receives the simulated input; i.e., `val` becomes equal to "abc".
 - ▶ The production code assumes the user has entered this value.
 - ▶ It can't tell the difference between real and simulated input, which is the point.
- ▶ The test code uses this to check what happens for different kinds of input.

Using Simulated Input: Example Test Code

- ▶ Remember “`max()`”? Say we also have “`inputMax()`”.
- ▶ Instead of importing two numbers, it *inputs* them.
- ▶ Here’s how we might test it:

```
public static void testInputMax()  
{  
    System.setIn(  
        new ByteArrayInputStream("10\n15".getBytes());  
    assert 15 == MyUtils.inputMax();  
  
    System.setIn(  
        new ByteArrayInputStream("10\n-10".getBytes());  
    assert 10 == MyUtils.inputMax();  
    ... // One other test case  
}
```



Using Simulated Input: Example Test Code

- ▶ Remember “`max()`”? Say we also have “`inputMax()`”.
- ▶ Instead of importing two numbers, it *inputs* them.
- ▶ Here’s how we might test it:

```
def testInputMax():  
    sys.stdin = io.StringIO("10\n15")  
    assert 15 == MyUtils.inputMax()  
  
    sys.stdin = io.StringIO("10\n-10")  
    assert 10 == MyUtils.inputMax()  
    ... # One other test case
```




Capturing Output



- ▶ A “mirror image” of the problem of simulating input.
- ▶ If production code *displays* something, test code must be able to check it.
 - ▶ But, *normally*, your code cannot see its own output. Only the user can.
 - ▶ We must do something to *capture* it, before it is actually displayed.

Capturing Console Output

```
import java.io.*;  
...  
ByteArrayOutputStream capOut = new ByteArrayOutputStream();  
System.setOut(new PrintStream(capOut));  
... // call production code  
String actualOutput = capOut.toString();
```

 **Java**

- ▶ Create an “object” that can receive output, but *stores* it instead of displaying it.

 **Java** “ByteArrayOutputStream”
 **Python** “io.StringIO” (same as for input)


- ▶ Tell the system to use this, instead of the console.
- ▶ Afterwards, retrieve the text that was “displayed”.

Capturing Console Output

```
import sys, io
...
capOut = io.StringIO()
sys.stdout = capOut
... // call production code
actualOutput = capOut.getvalue()
```



- ▶ Create an “object” that can receive output, but *stores* it instead of displaying it.

 **Java** “ByteArrayOutputStream”

 **Python** “io.StringIO” (same as for input)

- ▶ Tell the system to use this, instead of the console.
- ▶ Afterwards, retrieve the text that was “displayed”.

Capturing Output: Example Test Code

- ▶ Say we have another variation of `max()` called `outputMax()`.
- ▶ Instead of returning the result, it *outputs* it.
- ▶ Here's how we might test it:

```
public static void testOutputMax()  
{  
    ByteArrayOutputStream capOut =  
        new ByteArrayOutputStream();  
    System.setOut(new PrintStream(capOut));  
    MyUtils.outputMax(10, 15); // Production code call  
    assert "15".equals(capOut.toString());  
  
    ... // Other test cases  
}
```



Capturing Output: Example Test Code

- ▶ Say we have another variation of `max()` called `outputMax()`.
- ▶ Instead of returning the result, it *outputs* it.
- ▶ Here's how we might test it:


```
def testOutputMax():  
    capOut = io.StringIO()  
    sys.stdout = capOut  
    MyUtils.outputMax(10, 15) # Production code call  
    assert "15" == capOut.getvalue()  
  
    ... # Other test cases
```



Simulating Input *and* Capturing Output

- ▶ What about `inputOutputMax()`?
- ▶ We can simulate input and capture output at the same time:

```
public static void testInputOutputMax()  
{  
    ByteArrayOutputStream capOut =  
        new ByteArrayOutputStream();  
    System.setOut(new PrintStream(capOut));  
    System.setIn(  
        new ByteArrayInputStream("10\n15".getBytes()));  
  
    MyUtils.inputOutputMax(); // Production code call  
    assert "15".equals(capOut.toString());  
    ... // Other test cases  
}
```



Simulating Input *and* Capturing Output

- ▶ What about `inputOutputMax()`?
- ▶ We can simulate input and capture output at the same time:

```
def testInputOutputMax():  
    capOut = io.StringIO()  
    sys.stdout = capOut  
    sys.stdin = io.StringIO("10\n15")  
  
    MyUtils.inputOutputMax() # Production code call  
    assert "15" == capOut.getvalue()  
    ... # Other test cases
```



Testing with Files

- ▶ If the production code deals reads/writes data files, the test code must also deal with them.
- ▶ *Conceptually* this is quite similar to console IO, but the fine details a bit different.
- ▶ To test a method/function that reads an input file:
 - ▶ The test code must setup (i.e. write) the input file beforehand.
- ▶ To test a method/function that writes an output file:
 - ▶ The test code must verify (i.e. read) the output file afterwards.

File Reading and Writing Syntax

- ▶ The following slides show code for reading/writing files.
 - ▶ In Java, we'll use `PrintWriter` and `Scanner`.
 - ▶ In Python, we'll use the "with" statement, and `open()`, `write()` and `read()`.
- ▶ There are other valid ways to do this.
- ▶ There's nothing particularly special about these particular reading/writing approaches.
 - ▶ Except that they're reasonably short and easy to put into slide form.
- ▶ The most important thing is that you understand the *concept* of what is happening.

Testing with Input Files


```
public static void testInputFileMax()  
{  
    PrintWriter inputFile = new PrintWriter("inputfile.txt");  
    inputFile.println("10\n15"); // <-- The test data  
    inputFile.close();  
  
    assert 15 == MyUtils.inputFileMax("inputfile.txt");  
    ... // Other test cases  
}
```



- ▶ First, we create a file containing test data.
- ▶ Then, we call the production code, which (in theory) reads that file and thus gets the test data.
- ▶ Let's assume the production code takes a filename parameter.
 - ▶ If so, this must be the same as the file we created.

Testing with Input Files


```
def testInputFileMax():  
    with open("inputfile.txt", mode = "w") as inputFile:  
        inputFile.write("10\n15") # <-- The test data  
  
    assert 15 == MyUtils.inputFileMax("inputfile.txt")  
    ... # Other test cases
```



- ▶ First, we create a file containing test data.
- ▶ Then, we call the production code, which (in theory) reads that file and thus gets the test data.
- ▶ Let's assume the production code takes a filename parameter.
 - ▶ If so, this must be the same as the file we created.

Testing with Output Files


```
public static void testOutputFileMax()  
{  
    MyUtils.outputFileMax(10, 15, "outputfile.txt");  
  
    Scanner outputFile = new Scanner("outputfile.txt");  
    String actual = outputFile.nextLine(); // <-- Actual  
    assert "15".equals(actual);           //      result  
  
    ... // Other test cases  
}
```



- ▶ First, we call the production code, which (in theory) creates a file and writes a result to it.
- ▶ Then, the test code reads that file, gets the result, and runs it through an assertion.

Testing with Output Files

```
def testOutputFileMax():  
    MyUtils.outputFileMax(10, 15, "outputfile.txt")  
  
    with open("outputfile.txt") as outputFile:  
        actual = outputFile.read().strip() # <-- Actual  
        assert "15" == actual             #      result  
  
    ... # Other test cases
```





- ▶ First, we call the production code, which (in theory) creates a file and writes a result to it.
- ▶ Then, the test code reads that file, gets the result, and runs it through an assertion.

Exceptions During Testing

Exceptions during testing occur because:

- ▶ The production code gave the wrong result.
- ▶ The production code unexpectedly threw an exception.
 - ▶ The test fails, but before we even get to the `assert` statement.
- ▶ The test code itself is broken.
 - ▶ Maybe the “expected result” is incorrect.
 - ▶ Maybe you didn’t set up a test input file properly.
- ▶ **The production code *expectedly* threw an exception.**
 - ▶ Often the production code *must* throw an exception, under certain circumstances.
 - ▶ So, the test code must check that it does.
 - ▶ Such a test should fail in the *absence* of an exception.

Handling Unexpected Exceptions

- ▶ You could use a `try-catch/except` statement.
- ▶ But *in test code*, it's simpler to just pass-on the exception.
- ▶ This will be interpreted as a test failure, as it should be.
- ▶  Python: nothing additional needs to be done!
- ▶  Java: add a `throws` clause:

```
public static void testThing() throws ExceptionType
{
    ...
}
```

ExceptionType is the particular kind of exception that the production code might generate.

Handling *Expected* Exceptions

- ▶ Remember `formatTime()` ?
- ▶ It returned "error" when passed invalid hours/minutes.
 - ▶ i.e., if we write `actual = MyUtils.formatTime(12, -10)`, we should expect `actual` to be "error".
- ▶ But, in the real world, it's more likely to be designed to throw an exception.
 - ▶ In this case, there is no string value at all, and by default the exception means a test failure.
 - ▶ But *this* exception means the code actually works!
 - ▶ i.e. it successfully identifies invalid values, and takes appropriate action.
 - ▶ The test should only fail if there *isn't* an exception.
 - ▶ How do we test for this?

Handling Expected Exceptions

- ▶ We need to use a `try` statement to sort this out.
- ▶ The test passes only if an exception makes the code jump to the `catch/except` block.

```
public static void testFormatTime()  
{  
    try  
    {  
        String actual = MyUtils.formatTime(12, -10);  
        assert false; // Test fails if it reaches this point  
    }  
    catch (IllegalArgumentException e) {} // Do nothing.  
  
    ... // Other test cases  
}
```



Handling Expected Exceptions

- ▶ We need to use a `try` statement to sort this out.
- ▶ The test passes only if an exception makes the code jump to the `catch/except` block.

```
def testFormatTime():  
    try:  
        actual = MyUtils.formatTime(12, -10)  
        assert False # Test fails if it reaches this point  
  
    except ValueError:  
        pass # Do nothing.  
  
    ... # Other test cases
```



Test Fixtures

- ▶ We're now seen various situations where test cases are written like this:
 1. Perform setting up.
 2. Call production code.
 3. Compare results.
 - ▶ And there's one more step we sometimes have:
 4. Tear down: restore everything to its original state.
- For instance:
- ▶ If the test code (or production code) created a file, we should delete it afterwards.
 - ▶ If the test code redirected console input or output, it should set it back afterwards.
 - ▶ The "setting up" and "tearing down" defines a *test fixture*:
 - ▶ A set of things in-place to make the test case work, and to isolate the test case from external factors.

Common Setting-Up and Tearing-Down

- ▶ Multiple test cases often require at least some of the same setting-up and tearing-down.
 - ▶ Some of the setting-up may be separate from providing the test data.
 - ▶ Perhaps *all* test cases require the same console input, just to make the production code work.
 - ▶ Or perhaps they all require the same file to exist.
- ▶ Reuse applies to test code too! We don't want to repeat ourselves.
- ▶ So, the convention is to have `setUp()` and `tearDown()` helper method/functions.
 - ▶ These do all the *common* work to establish a test fixture.
 - ▶ `setUp()` will be called immediately before *every* test method/function.
 - ▶ `tearDown()` will be called immediately after.

Text Fixture Example - *Without* a Test Framework

```
public class TestSuite
{
    public static void main(String[] args)
    {
        setUp();
        testMethod1();
        tearDown();
        setUp();
        testMethod2();
        tearDown();
    }

    public static void setUp() { ... }
    public static void testMethod1() { ... }
    public static void testMethod2() { ... }
    public static void tearDown() { ... }
}
```



Text Fixture Example - *Without* a Test Framework

```
def setUp(): ...  
def testFunction1(): ...  
def testFunction2(): ...  
def tearDown(): ...  
  
if __name__ == "__main__":  
    testMethod1()  
    testMethod2()
```



Test Fixtures in JUnit/unittest

- ▶ JUnit understands methods that have `@Before` and `@After` annotations.
 - ▶ It doesn't actually care what the names are.
- ▶ Python's `unittest` module looks for methods specifically called `setUp()` and `tearDown()`.
- ▶ In either case, the framework will automatically:
 - ▶ Call the `@Before/setup()` method before *each* test method.
 - ▶ Call the `@After/tearDown()` method after each test method.

Test Fixture Example - *With* a Test Framework

```
@RunWith(JUnit4.class)
```

```
public class TestSuite  
{
```

```
    @Before
```

```
    public void setUp() { ... }
```

```
    @Test
```

```
    public void testMethod1() { ... }
```

```
    @Test
```

```
    public void testMethod2() { ... }
```

```
    @After
```

```
    public void tearDown() { ... }
```

```
}
```



Test Fixture Example - *With* a Test Framework

```
import unittest
```



```
class TestSuite(unittest.TestCase):  
    def setUp(self): ...  
    def testMethod1(self): ...  
    def testMethod2(self): ...  
    def tearDown(self): ...
```

White-Box Testing

- ▶ Let's leave the implementation details now, and go back to *test design*.
 - ▶ i.e., how to decide which test cases we need in the first place.
- ▶ In "Black Box" testing:
 - ▶ We design test cases *without* looking at the code.
 - ▶ We just look at the parameters, return type, and documentation.
 - ▶ This is what Equivalence Partitioning and Boundary Value Analysis are doing.
- ▶ In "White Box" (or "Clear Box") testing, test cases are based on the *paths* through a method/function.

Paths

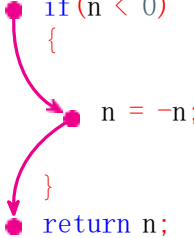
- ▶ A “path” is (roughly speaking) one possible way to “get through” a method/function, from start to end.
- ▶ There’s always at least one path, but there are often more.
- ▶ Different paths are created by conditional statements, like `if`.
 - ▶ When there are multiple choices, this translates to multiple paths.
 - ▶ Multiple paths are also created by `switch`, `while`, `do-while`, `for` and `try-catch/except`.
 - ▶ (Not all of these exist in Python.)
- ▶ White-box testing ensures that we test each path – each possible way through the production code.
 - ▶ Each path becomes a test case!

if Statement Paths

- ▶ if statements have two paths.
- ▶ 1st path:

Java

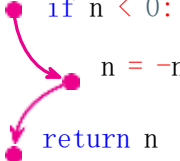
```
double abs(double n)
{
    if(n < 0)
    {
        n = -n;
    }
    return n;
}
```



The control flow graph for the Java `abs` method starts at a pink node before the `if` statement. A curved pink arrow leads to a second pink node after the `n = -n;` statement. From there, another curved pink arrow leads to a third pink node before the `return n;` statement. The graph ends at a final pink node after the closing brace.

Python

```
def myAbs(n):
    if n < 0:
        n = -n
    return n
```



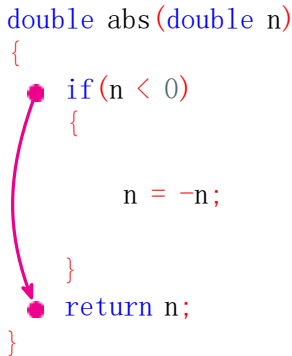
The control flow graph for the Python `myAbs` function starts at a pink node before the `if` statement. A curved pink arrow leads to a second pink node after the `n = -n` statement. From there, another curved pink arrow leads to a third pink node before the `return n` statement. The graph ends at a final pink node after the function definition.

if Statement Paths

- ▶ if statements have two paths.
- ▶ 2nd path:

Java

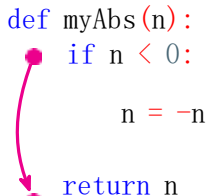
```
double abs(double n)
{
    if(n < 0)
    {
        n = -n;
    }
    return n;
}
```



The diagram shows a control flow graph for the Java `abs` method. It starts at the opening curly brace of the method. A magenta dot marks the entry to the `if(n < 0)` statement. From this dot, a curved magenta arrow points down to another magenta dot located at the `return n;` statement, representing the path taken when the condition is false.

Python

```
def myAbs(n):
    if n < 0:
        n = -n
    return n
```



The diagram shows a control flow graph for the Python `myAbs` function. It starts at the `def` line. A magenta dot marks the entry to the `if n < 0:` block. From this dot, a curved magenta arrow points down to another magenta dot located at the `return n` statement, representing the path taken when the condition is false.

White Box Test Design - if statements

- ▶ In drawing up our test design, we work with paths instead of equivalence categories.
- ▶ We still need to pick test data and expected results:

Test design for abs:

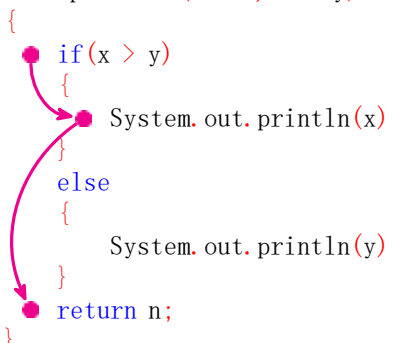
Path	Test Data	Expected Result
1. Enter the if	$n = -5$	5
2. DO NOT enter the if	$n = 10$	10

if-else Paths

- ▶ if-else statements also have two paths.
- ▶ 1st path:

Java

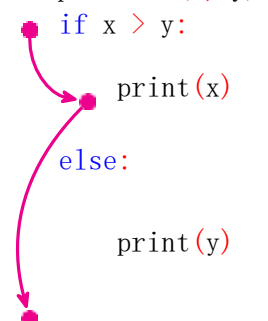
```
void printMax(int x, int y)
{
    if (x > y)
    {
        System.out.println(x);
    }
    else
    {
        System.out.println(y);
    }
    return n;
}
```



The diagram shows a control flow graph for the Java code. It starts with a pink dot at the beginning of the function. A curved arrow leads to another pink dot at the start of the 'if' statement. From there, a straight arrow leads to a pink dot before 'System.out.println(x);'. Another straight arrow leads to a pink dot before the 'else' block. A curved arrow then leads to a pink dot before 'return n;'. Finally, a straight arrow leads to the end of the function.

Python

```
def printMax(x, y):
    if x > y:
        print(x)
    else:
        print(y)
```



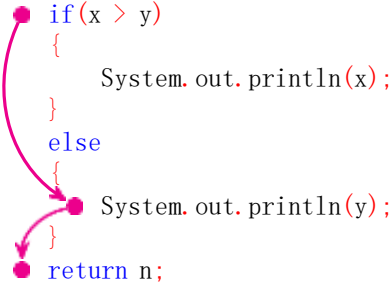
The diagram shows a control flow graph for the Python code. It starts with a pink dot at the beginning of the function. A curved arrow leads to a pink dot at the start of the 'if' statement. From there, a straight arrow leads to a pink dot before 'print(x)'. Another straight arrow leads to a pink dot before the 'else' block. A curved arrow then leads to a pink dot before 'print(y)'. Finally, a straight arrow leads to the end of the function.

if-else Paths

- ▶ if-else statements also have two paths.
- ▶ 2nd path:

Java

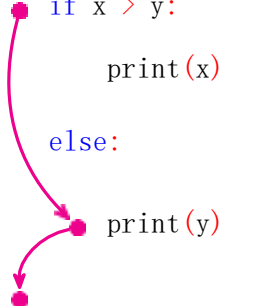
```
void printMax(int x, int y)
{
    if(x > y)
    {
        System.out.println(x);
    }
    else
    {
        System.out.println(y);
    }
    return n;
}
```



The control flow graph for the Java code shows a start node (red dot) at the beginning of the `if` statement. A curved arrow leads to a second node (red dot) at the start of the `else` block. From there, a curved arrow leads to a third node (red dot) at the end of the `return` statement. The `if` and `else` blocks are represented by curly braces.

Python

```
def printMax(x, y):
    if x > y:
        print(x)
    else:
        print(y)
```



The control flow graph for the Python code shows a start node (red dot) at the beginning of the `if` statement. A curved arrow leads to a second node (red dot) at the start of the `else` block. From there, a curved arrow leads to a third node (red dot) at the end of the function. The `if` and `else` blocks are represented by colons.

if-else Test Design

Path	Test Data	Expected Result
1. Enter the if part	$x = 10, y = 5$	output: "10"
2. Enter the else part	$x = 10, y = 20$	output: "20"

Loop Paths


- ▶ How does the concept of a “path” apply to loops?
 - ▶ while, do-while, for.
- ▶ It may seem like a loop should have *many* paths.
 - ▶ You can go around a `while` loop *any number* of times, for instance.
- ▶ In fact, loops have exactly two paths, just like `if` statements.
 1. The path that never enters the loop.
 - ▶ (Or, for Java’s `do-while` loop, never repeats the loop.)
 2. A path that *does* enter the loop.
- ▶ Whether the loop repeats twice, or a million times, it’s considered the same path.

while Paths

1st path:

 **Java**


```
int readPositive()  
{  
    int val = ...; // Input  
    while(val <= 0) {  
        System.out.println(  
            "Not positive");  
        val = ...; // Input  
    }  
    return val;  
}
```



The flowchart for the Java code shows a vertical path starting from the input line, going down to the while loop condition, then down to the loop body, and finally down to the return statement. A curved arrow on the left side of the loop body indicates the loop iteration back to the condition.

 **Python**

```
def readPositive():  
    val = int(input())  
    while val <= 0:  
        print(  
            "Not positive")  
        val = int(input())  
    return val
```



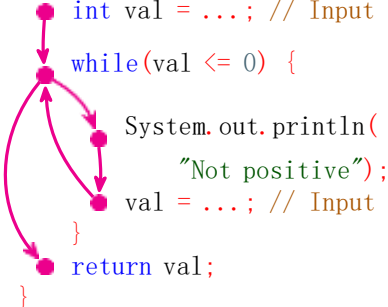
The flowchart for the Python code shows a vertical path starting from the input line, going down to the while loop condition, then down to the loop body, and finally down to the return statement. A curved arrow on the left side of the loop body indicates the loop iteration back to the condition.

while Paths

2nd path:

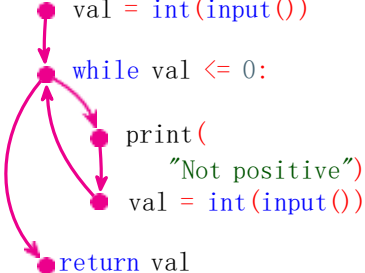
 **Java**

```
int readPositive()  
{  
    int val = ...; // Input  
    while(val <= 0) {  
        System.out.println(  
            "Not positive");  
        val = ...; // Input  
    }  
    return val;  
}
```



 **Python**

```
def readPositive():  
    val = int(input())  
    while val <= 0:  
        print(  
            "Not positive")  
        val = int(input())  
    return val
```



while Test Design

Path	Test Data	Expected Result
1. Enter loop	input: "-5\n10"	val = 10, output: "Not positive"
2. Skip loop	input: "5"	val = 5, output: ""

do-while Paths

- ▶ do-while loops don't exist in Python, so this is just FYI.
- ▶ 1st path (no iteration):

```
void userAdd() {  
    int val1, val2;  
    System.out.println("Enter two positive numbers");  
    do {  
        val1 = ...; // Input  
        val2 = ...; // Input  
    }  
    while(val1 <= 0 || val2 <= 0);  
    System.out.println(val1 + val2);  
}
```

Java

do-while Paths

- ▶ do-while loops don't exist in Python, so this is just FYI.
- ▶ 2nd path (some iteration):

```
void userAdd() {  
    int val1, val2;  
    System.out.println("Enter two positive numbers");  
    do {  
        val1 = ...; // Input  
        val2 = ...; // Input  
    }  
    while(val1 <= 0 || val2 <= 0);  
    System.out.println(val1 + val2);  
}
```



try-catch/except Paths

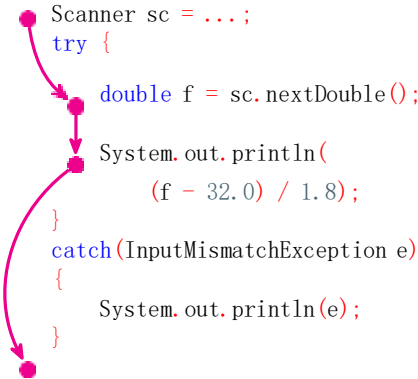
- ▶ This construct is intended for exception handling.
 - ▶ Java has a `try-catch` statement.
 - ▶ Python has a `try-except` statement.
 - ▶ Same thing (but the names of exceptions are different).
- ▶ You'd have one path for "success", where no exception occurs.
- ▶ You'd have one additional path for each `catch/except` clause.
 - ▶ i.e., for each different kind of exception that you're handling.

try-catch/except Paths

1st path (success):

 **Java**

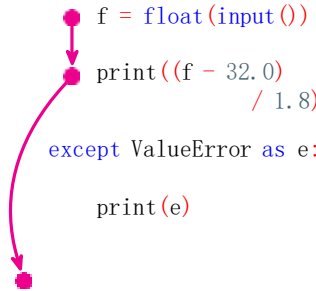
```
void convertF2C() {  
    Scanner sc = ...;  
    try {  
        double f = sc.nextDouble();  
        System.out.println(  
            (f - 32.0) / 1.8);  
    }  
    catch (InputMismatchException e)  
    {  
        System.out.println(e);  
    }  
}
```



The flow diagram for the Java code shows three nodes (pink circles) connected by arrows. The first node is at the start of the try block. An arrow points down to the second node, which is at the end of the try block (before the catch block). From the second node, an arrow points down to the third node, which is at the end of the catch block. A curved arrow also points from the second node back to the third node, representing the flow from the try block to the catch block.

 **Python**

```
def convertF2C():  
    try:  
        f = float(input())  
        print((f - 32.0)  
              / 1.8)  
    except ValueError as e:  
        print(e)
```



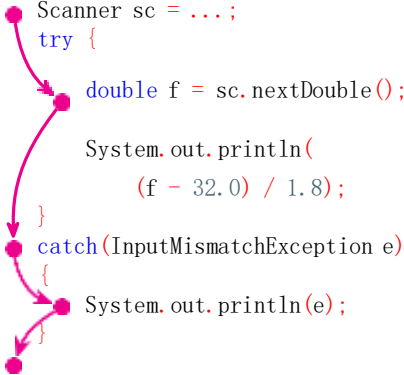
The flow diagram for the Python code shows three nodes (pink circles) connected by arrows. The first node is at the start of the try block. An arrow points down to the second node, which is at the end of the try block (before the except block). From the second node, an arrow points down to the third node, which is at the end of the except block. A curved arrow also points from the second node back to the third node, representing the flow from the try block to the except block.

try-catch/except Paths

2nd path (invalid, non-numerical input):

 **Java**

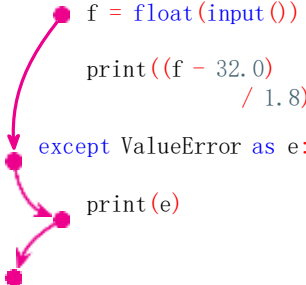
```
void convertF2C() {  
    Scanner sc = ...;  
    try {  
        double f = sc.nextDouble();  
        System.out.println(  
            (f - 32.0) / 1.8);  
    }  
    catch (InputMismatchException e)  
    {  
        System.out.println(e);  
    }  
}
```



The flowchart illustrates the execution path for the Java code. It starts at a pink dot at the beginning of the try block, moves down to a pink dot at the end of the try block, then down to a pink dot at the beginning of the catch block, and finally down to a pink dot at the end of the catch block. Arrows indicate the flow between these points.

 **Python**

```
def convertF2C():  
    try:  
        f = float(input())  
        print((f - 32.0)  
              / 1.8)  
    except ValueError as e:  
        print(e)
```



The flowchart illustrates the execution path for the Python code. It starts at a pink dot at the beginning of the try block, moves down to a pink dot at the end of the try block, then down to a pink dot at the beginning of the except block, and finally down to a pink dot at the end of the except block. Arrows indicate the flow between these points.

Other Control Statements and Combinations

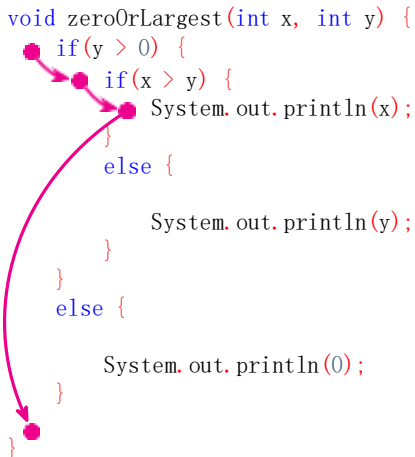
- ▶ `for` loops are like specialised `while` loops.
- ▶ `if-else-if-...-else` sequences create several paths.
 - ▶ Two paths for the first `if`.
 - ▶ One additional path for each additional `else-if/elif`.
- ▶ `switch` statements (in Java) create several paths.
 - ▶ One for each case, plus one for the default (even if the default is not specified).
 - ▶ These do not exist in Python.
- ▶ Everything can occur in combinations.
 - ▶ Think about control constructs as *splitting* one path into two (or more).
 - ▶ A single `if` gives you two paths.
 - ▶ *Another* nested `if` will split one of paths into *two more* paths, giving you three.

Nested ifs

1st path:

 **Java**

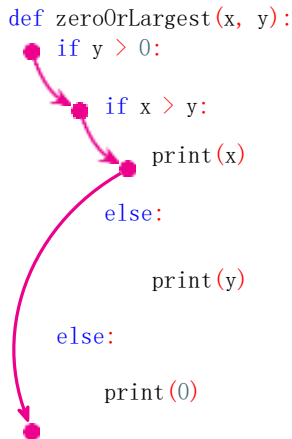
```
void zeroOrLargest(int x, int y) {  
    if(y > 0) {  
        if(x > y) {  
            System.out.println(x);  
        }  
        else {  
            System.out.println(y);  
        }  
    }  
    else {  
        System.out.println(0);  
    }  
}
```



The control flow graph for the Java code starts at a node before the first 'if' statement. An arrow leads to a node before the second 'if' statement. From there, an arrow leads to a node before 'System.out.println(x);'. Another arrow leads to a node before the 'else' block. From the 'else' block, an arrow leads to a node before 'System.out.println(y);'. Another arrow leads to a node before the 'else' block of the outer 'if'. From there, an arrow leads to a node before 'System.out.println(0);'. Finally, an arrow leads to the exit node after the closing brace of the method.

 **Python**

```
def zeroOrLargest(x, y):  
    if y > 0:  
        if x > y:  
            print(x)  
        else:  
            print(y)  
    else:  
        print(0)
```



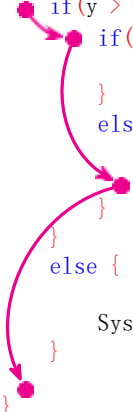
The control flow graph for the Python code starts at a node before the first 'if' statement. An arrow leads to a node before the second 'if' statement. From there, an arrow leads to a node before 'print(x)'. Another arrow leads to a node before the 'else:' block. From the 'else:' block, an arrow leads to a node before 'print(y)'. Another arrow leads to a node before the 'else:' block of the outer 'if'. From there, an arrow leads to a node before 'print(0)'. Finally, an arrow leads to the exit node after the function definition.

Nested ifs

2nd path:


 **Java**

```
void zeroOrLargest(int x, int y) {  
    if(y > 0) {  
        if(x > y) {  
            System.out.println(x);  
        }  
        else {  
            System.out.println(y);  
        }  
    }  
    else {  
        System.out.println(0);  
    }  
}
```



 **Python**

```
def zeroOrLargest(x, y):  
    if y > 0:  
        if x > y:  
            print(x)  
        else:  
            print(y)  
    else:  
        print(0)
```




Nested ifs

3rd path:

 **Java**


```
void zeroOrLargest(int x, int y) {  
    if(y > 0) {  
        if(x > y) {  
            System.out.println(x);  
        }  
        else {  
            System.out.println(y);  
        }  
    }  
    else {  
        System.out.println(0);  
    }  
}
```



The control flow graph for the Java code starts at a red dot at the beginning of the function. A curved magenta arrow leads to another red dot at the start of the first 'if' statement. From there, a straight magenta arrow leads to a red dot at the start of the nested 'if' statement. A straight magenta arrow then leads to a red dot at the end of the nested 'if' block. From this dot, a straight magenta arrow leads to a red dot at the start of the 'else' block. A straight magenta arrow then leads to a red dot at the end of the 'else' block. Finally, a straight magenta arrow leads to a red dot at the end of the function.

 **Python**

```
def zeroOrLargest(x, y):  
    if y > 0:  
        if x > y:  
            print(x)  
        else:  
            print(y)  
    else:  
        print(0)
```



The control flow graph for the Python code starts at a red dot at the beginning of the function. A curved magenta arrow leads to another red dot at the start of the first 'if' statement. From there, a straight magenta arrow leads to a red dot at the start of the nested 'if' statement. A straight magenta arrow then leads to a red dot at the end of the nested 'if' block. From this dot, a straight magenta arrow leads to a red dot at the start of the 'else' block. A straight magenta arrow then leads to a red dot at the end of the 'else' block. Finally, a straight magenta arrow leads to a red dot at the end of the function.

Nested ifs Test Design

Path	Test Data	Expected Result
1. Enter both if parts	$y = 5, x = 10$	output: "10"
2. Enter inner else part	$y = 5, x = 2$	output: "5"
3. Enter outer else part	$y = -5$	output: "0"

- ▶ Notice that we've omitted x in the 3rd test case.
 - ▶ We must provide *some* x value when calling `zeroOrLargest()`, but the value is irrelevant to the test design.
- ▶ In a *black box* design, we would have *four* test cases instead:
 1. $y > 0, x > y$ (equivalent to 1 above);
 2. $y > 0, x \leq y$ (equivalent to 2 above);
 3. $y \leq 0, x > y$;
 4. $y \leq 0, x \leq y$.

Black Box or White Box?

- ▶ Why choose one over the other?
- ▶ Number of paths (in white-box testing) may differ from the number of equivalence categories (in black-box testing).
- ▶ Black Box testing:
 - ▶ Test cases can be designed *before* the production code exists.
 - ▶ You can change algorithms without changing the test code.
- ▶ White Box testing:
 - ▶ You can better understand the different behaviours that the production code should have.
 - ▶ What decisions must the production code make?
 - ▶ In black-box testing, you take an educated guess.
 - ▶ In white-box testing, you can *see* the decisions.
 - ▶ Changes to production code usually mean changes to test code.
 - ▶ However, your test code may be more up-to-date as a result.

That's all for now!