Penn Engineering
Online Learning

Video 2.1

Arvind Bhusnurmath

Topics

- Why is testing important?
- Different types of testing
- Unit testing

Software testing

- Integral part of development.
- If you ship a software with bugs, you will lose money.
- The easier a bug is detected, the less time it takes to fix it.
- Huge job market for software testers.
- In most organizations, a software developer is responsible for writing their own tests as well.

Software testing

- **Black box** - does this method (or collection of methods) with this input lead to this specified output. You do not care what the method is doing internally.
- **White box** - You *do* care how the thing being tested actually works. As an example, instead of just checking what the method returns, you are checking that its local variables are all having correct values.
- **Unit testing** - testing software components. For our purposes in this course we will usually call one method one **unit**.

Old way v/s unit testing way

```
int max(int a, int b) {
     if (a > b) {
           return a;
     } else {
          return b;
     }
}
void testMax() {
   int x = max(3, 7);
   if (x != 7) {
     SOPL("max(3, 7) gives" + x);
   }
}
public static void main(String[] args)
{
   new MyClass().testMax();
}
```

```
@Test
void testMax() {
   assertEquals(7, max(3,7));
   assertEquals(3, max(3,-7));
}
```

Unit testing advantages

- First line of defense. If you have code that fails a unit test, that code is not deployed to the production environment.
- Modification of code becomes a less risky process.
- It represents a developers view of the software specifications.
- If a bug shows up in code despite the unit tests, a new unit test can be added to ensure that situation is covered.

Penn Engineering
Online Learning

Video 2.2

Arvind Bhusnurmath

Topics

- Writing a unit test
- Test driven development (TDD)

Writing the stub of a method

```
public class BankAccount {
   double balance;
   String accountOwner;
}
```

Assume we want to add a method called deposit that will deposit a certain amount of money into the account

First decide on the specs
parameter – amount to be deposited.
return void – the account balance will change

```
public void deposit(double amount){
   // do not write anything here initially
}
```

The goal is to use the unit test to guide the development.

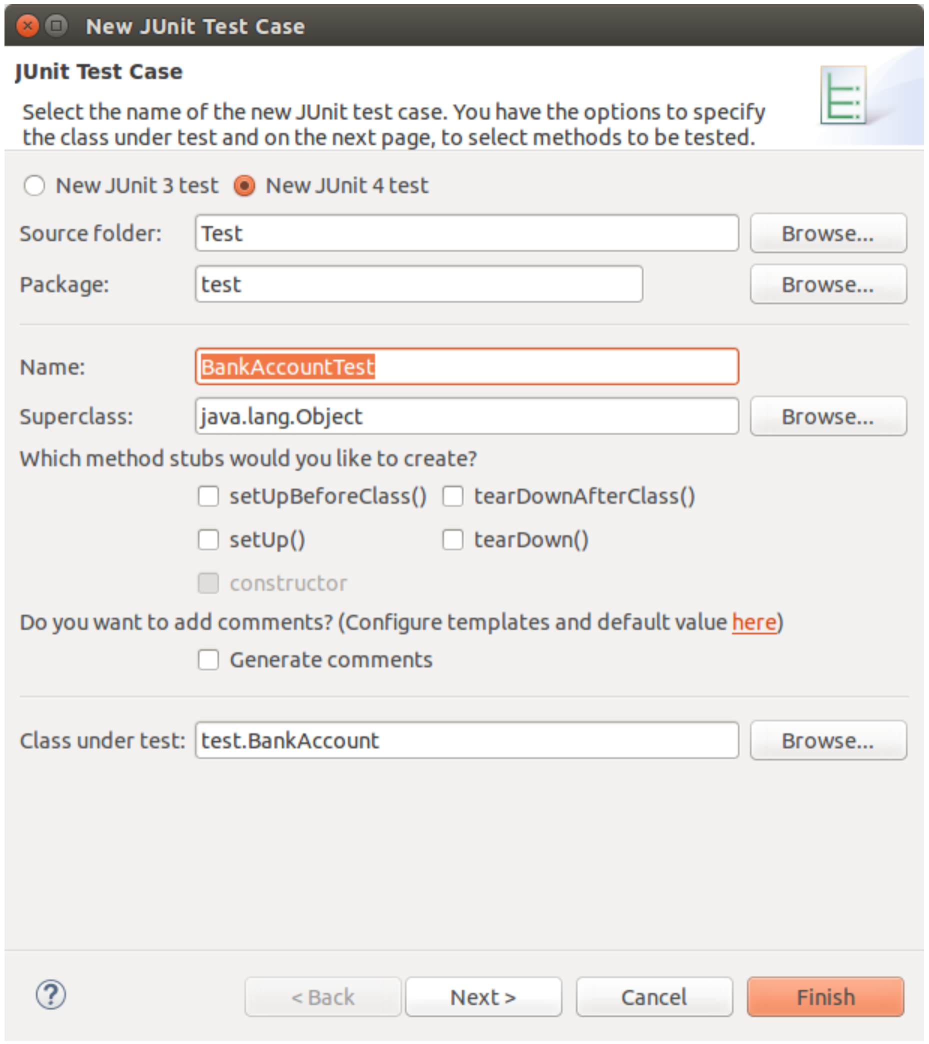
In cases where some value is being returned
- If it is an object return null
- If it is a primitive datatype just return some random value. 0 for integers for example.

Create a JUnit class

To create a JUnit test class:

Step1:

Right click on the class you want to test → New → JUnit Test Case



Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder:	Test	Browse...
Package:	test	Browse...
Name:	BankAccountTest	
Superclass:	java.lang.Object	Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

☐ constructor

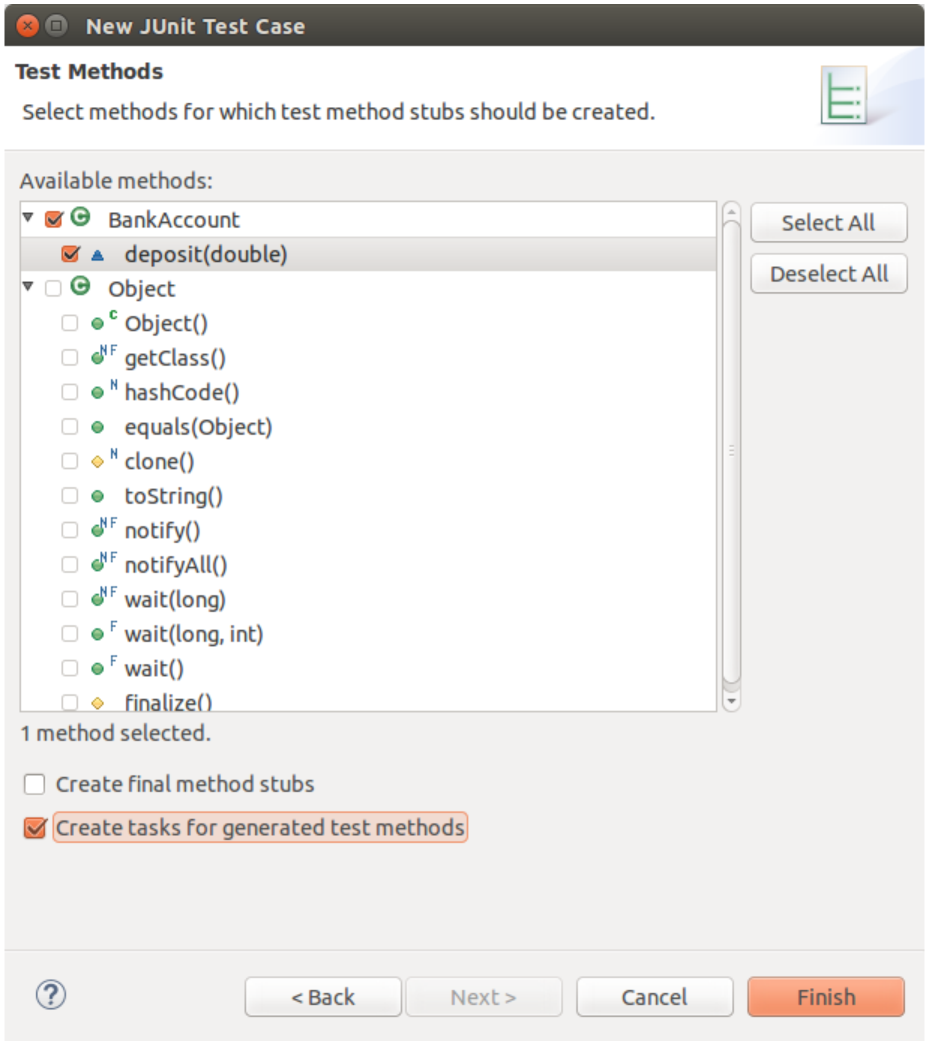
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: test.BankAccount Browse...

Create JUnit test cases

- To create a JUnit test class:
  - Do steps 1, if you haven't already
  - Click Next>
  - Use the checkboxes to decide which methods you want test cases for;
  - don't select Object or anything under it
  - Click Finish
- To run the tests:
  - Choose Run → Run As → JUnit Test


New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

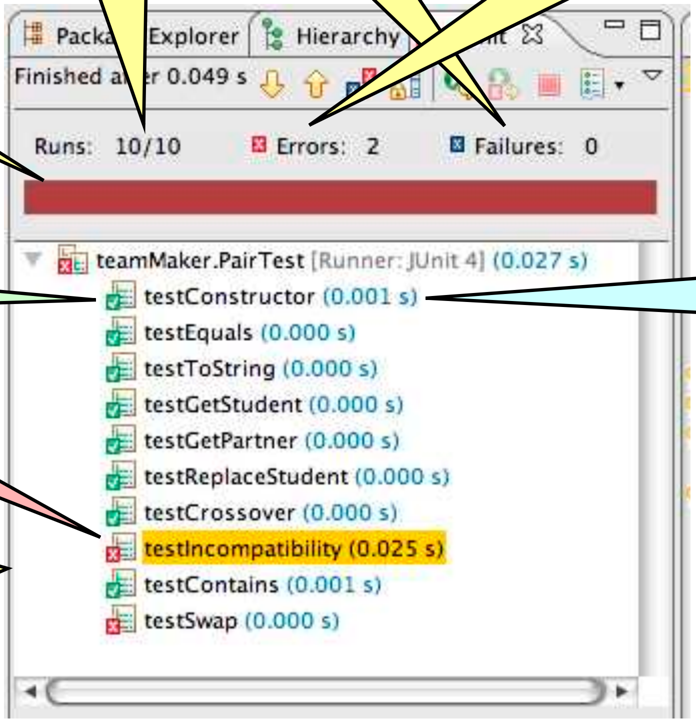
- [✓] BankAccount
  - [✓] deposit(double)
- [ ] Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)
  - [ ] clone()
  - [ ] toString()
  - [ ] notify()
  - [ ] notifyAll()
  - [ ] wait(long)
  - [ ] wait(long, int)
  - [ ] wait()
  - [ ] finalize()

1 method selected.

- [ ] Create final method stubs
- [✓] Create tasks for generated test methods

```
package banking;

import static org.junit.Assert.*;
import org.junit.Test;

public class BankAccountTest {
	@Test
	public void testDeposit() {
		fail("Not yet implemented");
	}
}
```

Viewing results in Eclipse



```
Runs:  10/10      Errors:  2      Failures:  0

teamMaker.PairTest [Runner: JUnit 4] (0.027 s)
    testConstructor (0.001 s)
    testEquals (0.000 s)
    testToString (0.000 s)
    testGetStudent (0.000 s)
    testGetPartner (0.000 s)
    testReplaceStudent (0.000 s)
    testCrossover (0.000 s)
    testIncompatibility (0.025 s)
    testContains (0.001 s)
    testSwap (0.000 s)
```

Bar is green if *all* tests pass, red otherwise

Ran 10 of the 10 tests

No tests failed, but…

Something unexpected happened in two tests

This test passed

This is how long the test took

Something is wrong

Depending on your preferences, this window might show *only* failed tests

Penn Engineering Online Learning

Video 2.3

Arvind Bhusnurmath

Topics

Using JUnit assertions
Complete example of Test Driven Development

XP approach to testing

- Extreme programming
- If code has no automated test case, it is assumed not to work
- A test framework is used so that automated testing can be done after every small change to the code.
- This may be as often as every 5 or 10 minutes
- If a bug is found after development, a test is created to keep the bug from coming back

A simple example

- Suppose you have a class Arithmetic with methods int multiply(int x, int y), and boolean isPositive(int x)
- import org.junit.*;
- import static org.junit.Assert.*;

```
public class ArithmeticTest {
@Test
public void testMultiply() {
    assertEquals(4, Arithmetic.multiply(2, 2));
    assertEquals(-15, Arithmetic.multiply(3, -5));
@Test
public void testIsPositive() {
   assertTrue(Arithmetic.isPositive(5));
   assertFalse(Arithmetic.isPositive(-5));
   assertFalse(Arithmetic.isPositive(0));
}
```

Assert methods I

- Within a test, Call the method being tested and get the actual result
- Assert what the correct result should be with one of the assert methods
- These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an `AssertionError` if the test fails
- JUnit catches these Errors and shows you the result

```
static void assertTrue(boolean test)
static void assertTrue(String message, boolean test)
```

  - Throws an AssertionError if the test fails
  - The optional message is included in the Error

```
static void assertFalse(boolean test)
static void assertFalse(String message, boolean test)
```

Example: Counter class

- We'll create a simple counter class
  - The constructor will create a counter and set it to zero
  - The increment method will add 1 to the counter and return the new value
  - The decrement method will subtract 1 from the counter and return the new value
- We write the test methods before we write the code
  - Write the method stubs
  - Let the IDE take care of generating the test method stubs

Junit tests for Counter

```
public class CounterTest {
    Counter counter1;

    @Before
    void setUp() {
         counter1 = new Counter();
    }

    @Test
    public void testIncrement() {
         assertTrue(counter1.increment() == 1);
         assertTrue(counter1.increment() == 2);
    }

    @Test
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

The actual Counter class

```
public class Counter {
    int count = 0;
    public int increment() {
         count += 1; return count;
    }
    public int decrement() {
        count -= 1; return count;
    }
    public int getCount() {
       return count;
    }
}
```

Quick version of the equals method

- You can compare primitives with ==
- Java has a method x.equals(y), for comparing objects
  - This method works great for Strings and a few other Java classes
  - For objects of classes that you create, you have to define equals
- To define equals for your own objects, define exactly this method:

```
public boolean equals(Object obj) {...}
```

- The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (say, Person):

```
public boolean equals(Object something) {
   Person p = (Person)something;
   return this.name == p.name;//test for equality
}
```

Assert Methods

```
assertEquals(expected, actual)
assertEquals(String message, expected, actual)
```

expected and actual must be both objects or the same primitive type

For objects, uses your equals method, if you have defined it properly, as described previously

```
assertNull(Object object)
assertNull(String message, Object object)
Asserts that the object is null (undefined)
```

Penn Engineering
Online Learning

Video 2.4

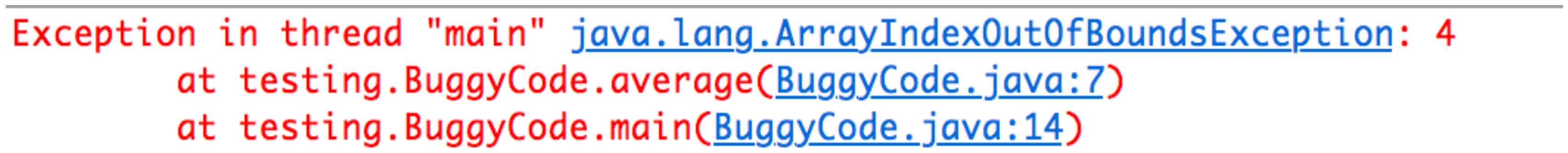
Arvind Bhusnurmath

Topics

- Reading an error stacktrace
- Common Exceptions you will encounter
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundException`
  - `NullPointerException`

Debugging with a stacktrace

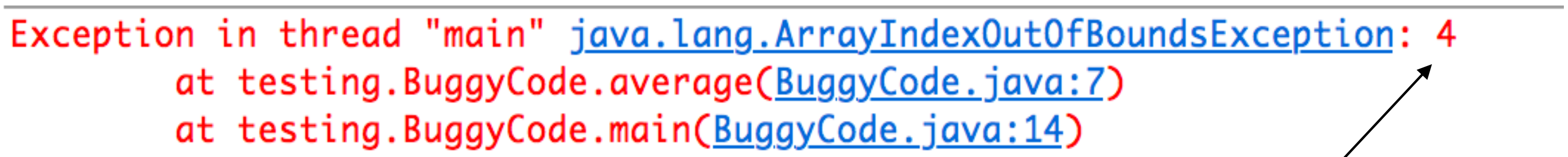
- A sequence of method calls



```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
	at testing.BuggyCode.average(BuggyCode.java:7)
	at testing.BuggyCode.main(BuggyCode.java:14)
```

Where to begin the debugging process

- Read the stacktrace top to bottom
- Find the first line of code that *you* wrote.
- Exceptions "bubble up". More on this later.
- Some exceptions have self explanatory names
  - `ArrayIndexOutOfBoundsException`
  - `ArithmeticException`
- Pay attention to any extra information being provided by the stacktrace

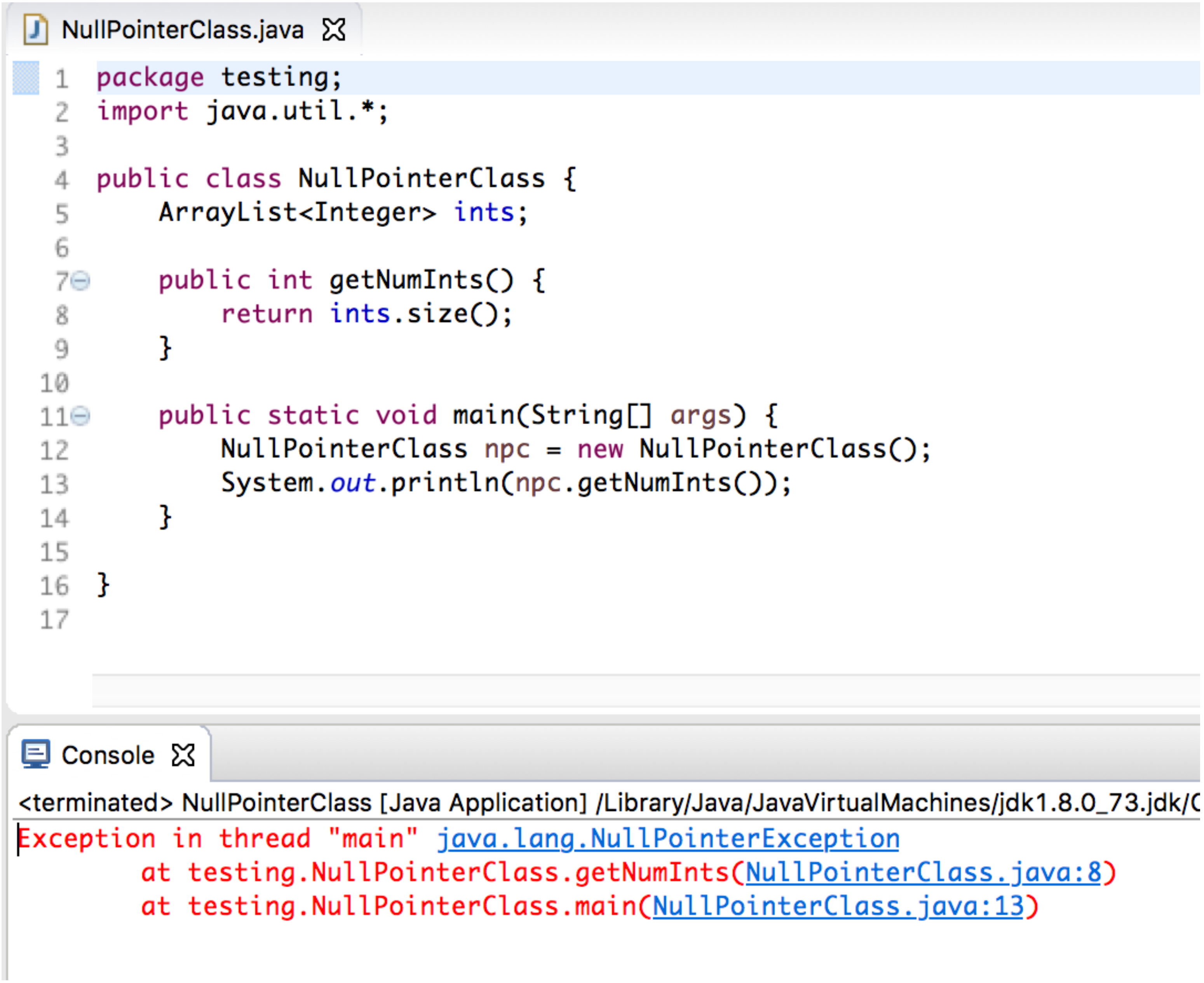


```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
	at testing.BuggyCode.average(BuggyCode.java:7)
	at testing.BuggyCode.main(BuggyCode.java:14)
```

Trying to index array index 4 caused this exception.

NullPointerException

- One of the most common errors you see
- Occurs when you are trying to access a method or instance variable from an object that is null.
- Usually caused by
  - Forgetting to call a constructor to first create the object
  - If you have a collection of data, then accessing beyond the first or last element of data.

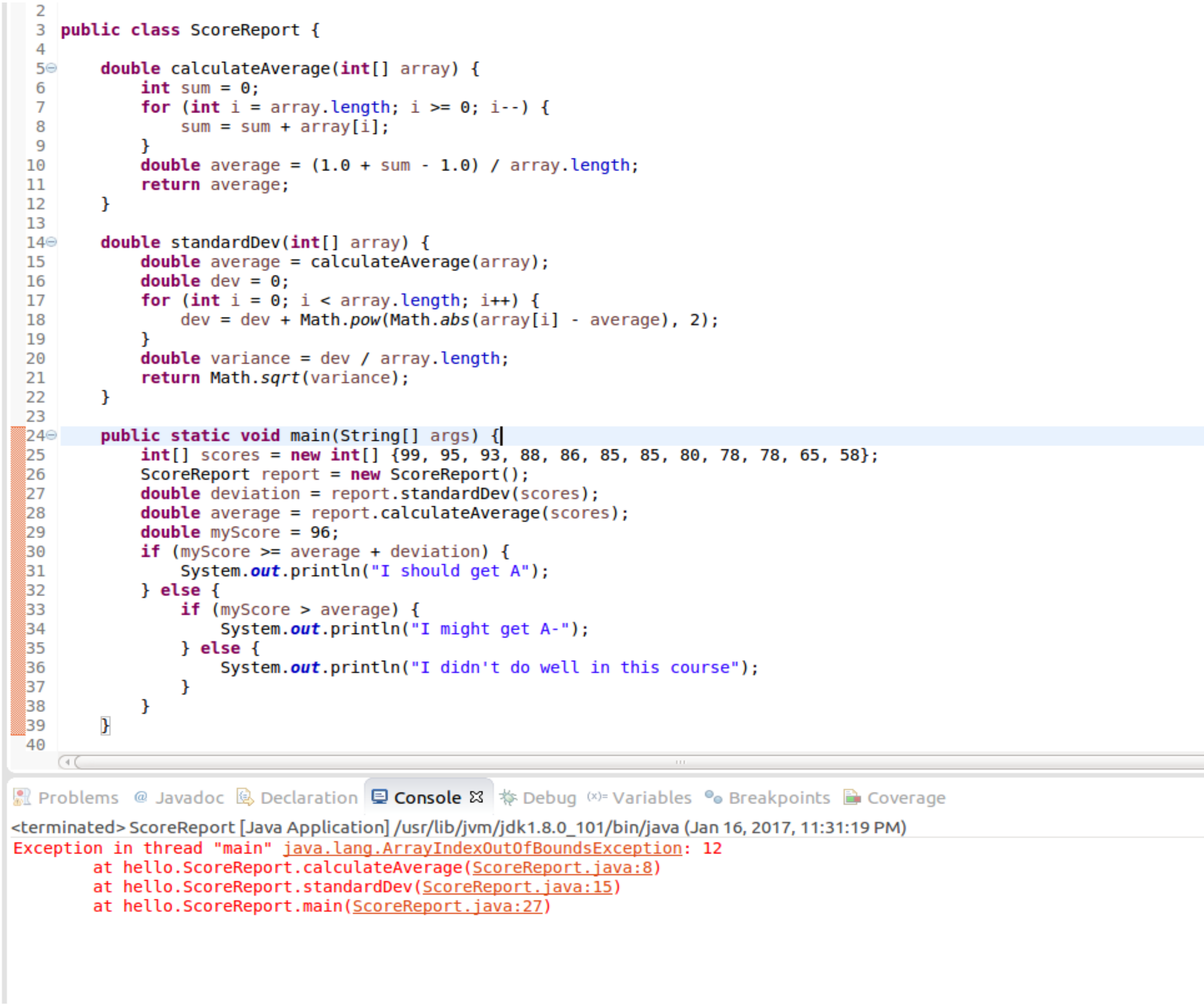

NullPointerClass.java

```java
package testing;
import java.util.*;

public class NullPointerClass {
    ArrayList<Integer> ints;

    public int getNumInts() {
        return ints.size();
    }

    public static void main(String[] args) {
        NullPointerClass npc = new NullPointerClass();
        System.out.println(npc.getNumInts());
    }

}
```

Console

```
<terminated> NullPointerClass [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_73.jdk/C
Exception in thread "main" java.lang.NullPointerException
	at testing.NullPointerClass.getNumInts(NullPointerClass.java:8)
	at testing.NullPointerClass.main(NullPointerClass.java:13)
```

Longer stacktraces



```java

public class ScoreReport {

    double calculateAverage(int[] array) {
        int sum = 0;
        for (int i = array.length; i >= 0; i--) {
            sum = sum + array[i];
        }
        double average = (1.0 + sum - 1.0) / array.length;
        return average;
    }

    double standardDev(int[] array) {
        double average = calculateAverage(array);
        double dev = 0;
        for (int i = 0; i < array.length; i++) {
            dev = dev + Math.pow(Math.abs(array[i] - average), 2);
        }
        double variance = dev / array.length;
        return Math.sqrt(variance);
    }

    public static void main(String[] args) {
        int[] scores = new int[] {99, 95, 93, 88, 86, 85, 85, 80, 78, 78, 65, 58};
        ScoreReport report = new ScoreReport();
        double deviation = report.standardDev(scores);
        double average = report.calculateAverage(scores);
        double myScore = 96;
        if (myScore >= average + deviation) {
            System.out.println("I should get A");
        } else {
            if (myScore > average) {
                System.out.println("I might get A-");
            } else {
                System.out.println("I didn't do well in this course");
            }
        }
    }

```

```
<terminated> ScoreReport [Java Application] /usr/lib/jvm/jdk1.8.0_101/bin/java (Jan 16, 2017, 11:31:19 PM)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 12
	at hello.ScoreReport.calculateAverage(ScoreReport.java:8)
	at hello.ScoreReport.standardDev(ScoreReport.java:15)
	at hello.ScoreReport.main(ScoreReport.java:27)
```

Penn Engineering

Online Learning

Video 2.6

Arvind Bhusnurmath

Topics

- Exception handling
- try. catch. finally

Errors and Exceptions

- An error is a bug in your program. For example a divide by zero.
- An exception is a problem whose cause is outside your program. For example running out of memory.

What to do with errors and exceptions

- An error is a bug and therefore should be fixed.
- An exception is a problem that your program may encounter.
- The situation in which you encounter the problem might not be the norm but you want to ensure that your program does not completely crash.

Dealing with exceptions

- A lot of exceptions arise when you are handling files
  - A needed file may be missing
  - You may not have permission to write a file
  - A file may be the wrong type
- Exceptions may also arise when you use someone else's classes (or they use yours)
  - You might use a class incorrectly
  - Incorrect use should result in an exception

Three approaches to error checking

- **Ignore all but the most important errors**
  - The code is cleaner, but the program will misbehave when it encounters an unusual error
- **Do something appropriate for every error**
  - The code is tough to read, but the program works better
  - You might still forget some error conditions
- **The Java method - Do the normal processing in one place, handle the errors in another** The code is at least reasonably uncluttered
  - Java tries to ensure that you handle every error

The try statement

- The try statement (also called the try-catch statement) separates normal code from the error handling

```
try {
    do the "normal" code, ignoring
exceptions
}
 catch (some exception) {
    handle the exception
}
 catch (some other exception) {
    handle the exception
}
```

Exception handling is *not* optional in Java

- For certain situations, most commonly in file handling, Java insists that you do something about the exceptional situations.

- If you do not catch an exception, the code does not even compile.

How Java handles errors behind the scenes

- In Java, an error doesn't *necessarily* cause your program to crash
- When an *error* occurs, Java throws an Error object for you to use
  - You can catch this object to try to recover
  - You can *ignore* the error (the program will crash)
- When an *exception* occurs, Java throws an Exception object for you to use
  - You **cannot ignore** an Exception; you must catch it
  - You get a *syntax error* if you forget to take care of any possible Exception

A few kinds of Exceptions

- `IOException`: a problem doing input/output
  - `FileNotFoundException`: no such file
  - `EOFException`: tried to read past the End Of File
- `NullPointerException`: tried to use a object that was actually null
- `NumberFormatException:` tried to convert a non-numeric String to a number
- `OutOfMemoryError`: the program has used all available memory
- There are about 200 predefined Exception types

What to do about Exceptions

- You have two choices:
  - You can "catch" the exception and deal with it
    - For Java's exceptions, this is usually the better choice
  - You can "pass the buck" and let some other part of the program deal with it
    - This is often better for exceptions that you create and throw
- Exceptions should be handled by the part of the program that is best equipped to do the right thing about them

What to do about Exceptions II

- You can catch exceptions with a try statement
  - When you catch an exception, you can try to repair the problem, or you can just print out information about what happened
- You can "pass the buck" by stating that the method in which the exception occurs "throws" the exception
  - Example:
    ```
     void openFile(String fileName) throws
    IOException { ... }
    ```
- Which of these you do depends on *whose responsibility it is* to do something about the exception
  - If the method "knows" what to do, it should do it
  - If it should really be up to the user (the method caller) to decide what to do, then "pass the buck"

How to use the try statement

- Put try {...} around any code that *might* throw an exception
  - This is a *syntax* requirement you cannot ignore
- For each Exception object that might be thrown, you must provide a catch phrase:
  - catch (***exception_type name***) {...}
  - You can have as many catch phrases as you need
  - *name* is a formal parameter that holds the exception object
  - You can send messages to this object and access its fields

finally

- After all the catch phrases, you can have an *optional* finally phrase

-
  ```
  try { ... }
  catch (AnExceptionType e) { ... }
  catch (AnotherExceptionType e) { ... }
  finally { ... }
  ```
- Whatever happens in try and catch, *even if it does a return statement,* the finally code will be executed
  - If no exception occurs, the finally will be executed after the try code
  - In an exception does occur, the finally will be executed after the appropriate catch code

How the try statement works

- The code in the try {...} part is executed

- If there are no problems, the catch phrases are skipped
- If an exception occurs, the program jumps *immediately* to the first catch clause that can handle that exception

- Whether or not an exception occurred, the finally code is executed

Using the exception

- When you say `catch(IOException e)`, e is a *formal parameter* of type `IOException`
  - A catch phrase is almost like a miniature method
  - e is an instance (object) of class IOException
  - Exception objects have methods you can use
- Here's an especially useful method that is defined for every exception type:
  - `e.printStackTrace();`
  - This prints out what the exception was, and how you got to the statement that caused it

printStackTrace

- `PrintStackTrace()` does *not* print on `System.out`, but on another stream, `System.err`
  - Eclipse writes this to the same Console window, but writes it in red
  - From the command line: both `System.out` and `System.err` are sent to the terminal window
- printStackTrace(***stream***) prints on the given stream
  - `printStackTrace(System.out)` prints on `System.out`, and this output is printed along with the "normal" output

Penn Engineering

Online Learning

Video 2.7

Arvind Bhusnurmath

Topics

- File I/O
- Reading and writing text files

Reading Files

- Reading a file is not too different from using the Scanner class.
- So far we have used the Scanner to read from standard input (input taken from the console).

Reading the contents of a file line by line

- Open the file

```
File textFile = new File(“test.txt”);
```

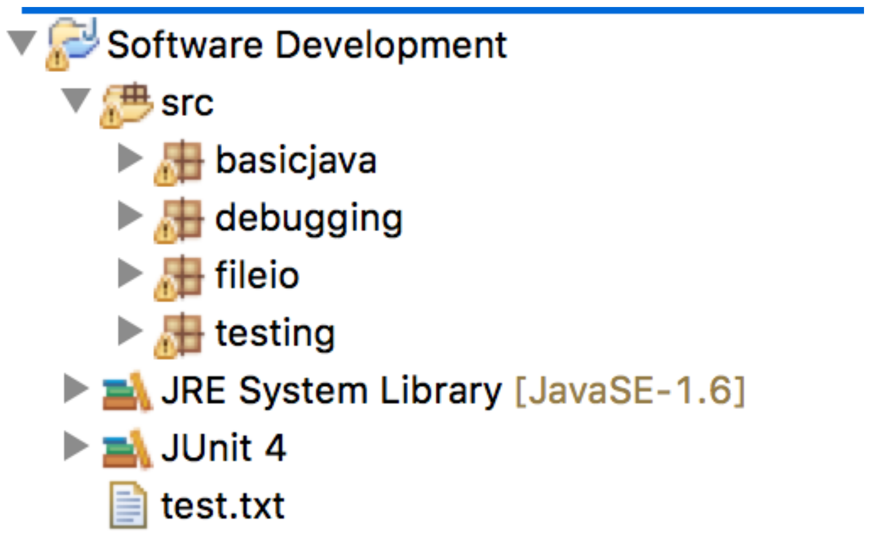
- Define a scanner on the file

```
Scanner scnr = new Scanner(textFile);
```

- Note that Java will insist that the `FileNotFoundException` gets handled. Use a try catch block. Decide what you want to do in the catch block if the file is not found.

Where should the file be located?

- Either specify the complete path
- If you want to include the file in the Eclipse project



Reading the contents of a file line by line

```
while(scnr.hasNextLine()){
   String line = scnr.nextLine();
   System.out.println(line);
}
```

Writing to a text file

- FileWriter writer = new FileWriter(filename, append?);
  - If the append argument is set to true the text will be added to the end of the file
  - If the append argument is set to false the file will be overwritten.
- PrintWriter printer = new PrintWriter(writer);
- printer.println(whatever you wanted to write to the file)
- printer.flush() in order to ensure that text is written out to the file. Without the flush method being called, the printing out to the file could be buffered.

Complete example

```
try {
      FileWriter fileWriter = new

       FileWriter(“test.txt”,true)
        PrintWriter printer = new

       PrintWriter(fileWriter);
        printer.println(”some words");
        printer.flush();
}
catch(IOException e) {
    e.printStackTrace();
}
```