

Object Oriented Programming

Week 9 Part 1
File I/O

Lecture

- Overview of Files
- Using Tests to learn Java
- Writing Text Files
- Reading Text Files

Overview of Files

Overview of Files

- How they are accessed:
 - sequential: data items must be accessed in the order in which they are stored (ie. start at the beginning and pass through all the items)
 - direct (or random): items are accessed by specifying their location
- How information is represented:
 - text files: data is stored in character form.
 - binary files: data is stored in internal binary form (faster and more compact than text files).

Types of Files

Summer.txt

Rough winds do shake the darling buds of May\nAnd Summer's lease hath all too short a date\nSometime too hot the eye of heaven shines,\nAnd oft is his gold complexion dimmed,\nAnd every fair from fair sometime declines...

Numbers.dat

Epôw10Žé;l
%®ú€9câÛ(3xLenf^x^{1 a}(İ½»¼øß:°µæEÝçMÛ¾à: ^qfõ
Ñ>|èæ=L¶...

Text Files

- Text file
 - human-readable with simple tools (Notepad, Ready, ...)
 - Each line terminated by end-of-line marker ('`\n`')
 - Example: `.java` files
- Easy to read and write text files
 - Advantage of "streams" approach to I/O
 - Use same classes and methods as `System.in` and `System.out`

Text Files: The File Class

- Need `File` object for each file program uses
 - `File inFile = new File("Summer.txt");`
- Purpose
 - Contains information about the file
 - A “go-between” for the file
 - Not the same as the file itself

Text Files: The File Class

- **Methods in the `File` class**
 - `exists()`: Tells if the file is there
 - `canRead()`: Tells if program can read the file
 - `canWrite()`: Tells if program can write to the file
 - `delete()`: Deletes the file
 - `isDirectory()`: Tells if file is really a directory
name
 - `isFile()`: Tells if the file is a file (not a directory)
 - `length()`: Tells the length of the file, in bytes

Using Tests to Explore Java

Learning from Tests

- To learn to use a new element of java, you need to use it
- But, you risk forgetting how it works if you do not use it frequently
- By putting your exploratory code in your test directory, you can develop examples of how a new feature works.

Tests as Specifications

- Running unit tests show that your code works
- But the code in the tests indicate what your code does
 - The assert statements tell the reader that value to expect when making a calls
 - They provide examples of working code
- Unit tests are specifications of the class, as well as tests of the class
 - What's more, they fail if the class does not meet the specification

Tests as specifications of Java

- You can add specifications of the crucial features of Java to your tests.
- Create unit tests that provide examples of Java classes you use
- Writing such a specification
 - Gives you examples if as you build your code
 - Gives you warning if Java specifications change
 - Indicate to others what elements you are using

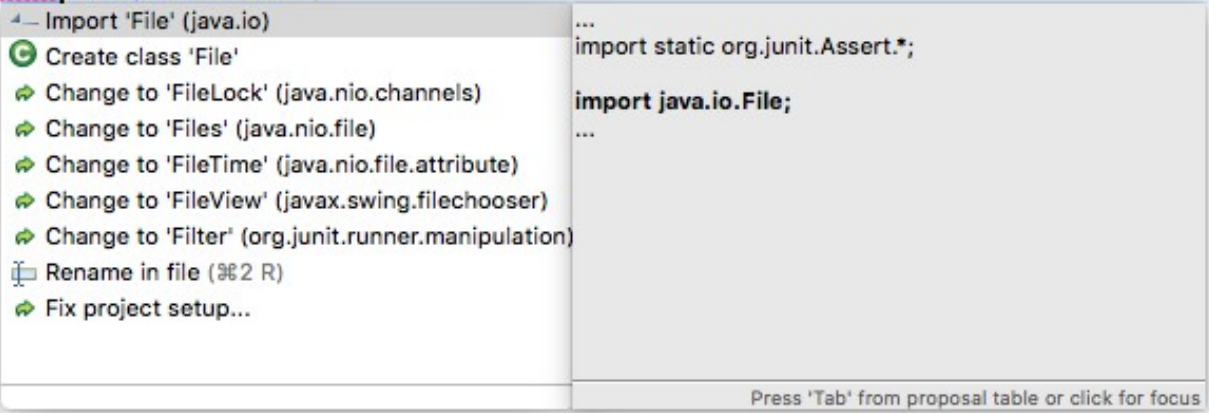
Text Files: The File Class

- Lets build tests of the following methods

- `exists()`: Tells if the file is there
- `canRead()`: Tells if program can read the file
- `canWrite()`: Tells if program can write to the file
- `delete()`: Deletes the file
- `isDirectory()`: Tells if file is really a directory
name
- `isFile()`: Tells if the file is a file (not a directory)
- `length()`: Tells the length of the file, in bytes

First we need to create a file

```
1 package oop.example;
2
3 import static org.junit.Assert.*;
4
5
6
7
8 public class ExampleFileIOTest {
9
10     @Before
11     public void setUp() throws Exception {
12     }
13
14     @Test
15     public void testFile() {
16         File f = new File("/tmp/test.txt");
17     }
18 }
19
20
```



The screenshot shows an IDE with a code completion menu open for the `File` class. The menu lists several options: 'Import 'File' (java.io)', 'Create class 'File'', 'Change to 'FileLock' (java.nio.channels)', 'Change to 'Files' (java.nio.file)', 'Change to 'FileTime' (java.nio.file.attribute)', 'Change to 'FileView' (javax.swing.filechooser)', 'Change to 'Filter' (org.junit.runner.manipulation)', 'Rename in file (⌘ R)', and 'Fix project setup...'. The 'Import 'File' (java.io)' option is selected. The background code is the same as shown in the previous block.

Import java.io.File

java.io.File



```
package oop.example;

import static org.junit.Assert.*;

import java.io.File;

import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void testFile() {
        File f = new File("/tmp/test.txt");
    }

}
```

Test that a file was created

```
package oop.example;

import static org.junit.Assert.*;

import java.io.File;

import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void testFile() {
        File f = new File("/tmp/test.txt");
        assertTrue("A new file was created", f instanceof File);
    }
}
```

Create a file and check that it was created →

Test exists()

```
package oop.example;

import static org.junit.Assert.*;

import java.io.File;

import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void testFile() {
        File f = new File("/tmp/test.txt");
        assertTrue("A new file was created", f instanceof File);
    }

    @Test
    public void testExists() {
        File f = new File("/tmp/test.txt");
        assertFalse("The new file does not yet exists", f.exists());
    }
}
```

The file does not yet exist

Refactor file creation to Before method

File f changed to field in class

Created in @Before method

Removed file creation

Removed file creation

Still works

```
package oop.example;

import static org.junit.Assert.*;

import java.io.File;

import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    private File f = null;

    @Before
    public void setUp() throws Exception {
        f = new File("/tmp/test.txt");
    }

    @Test
    public void testFile() {
        assertTrue("A new file was created", f instanceof File);
    }

    @Test
    public void testExists() {
        assertFalse("The new file does not yet exists", f.exists());
    }

}
```

```
▼ oop.example.ExampleFileIOTest [Runner: JUnit 4] (0.000 s)
  ✓ testFile (0.000 s)
  ✓ testExists (0.000 s)
```

Use `createNewFile()` to tests `exists()`

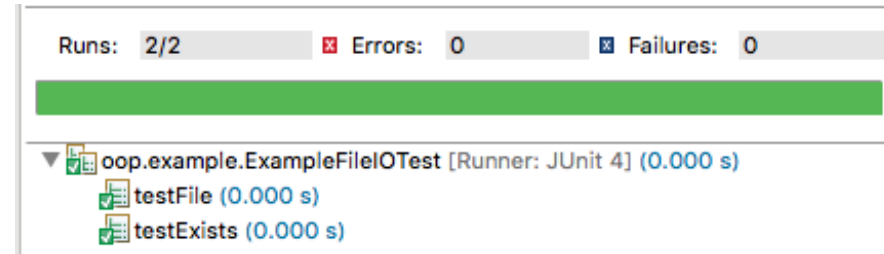
```
@Test
public void testExists() {
    assertFalse("The new file does not yet exists", f.exists());
    try {
        f.createNewFile();
    } catch (IOException e) {
        e.printStackTrace();
        fail("In testExists, f.createNewFile threw IO Exception");
    }
    assertTrue("The new file exists now", f.exists());
}
```

f.createNewFile creates empty file

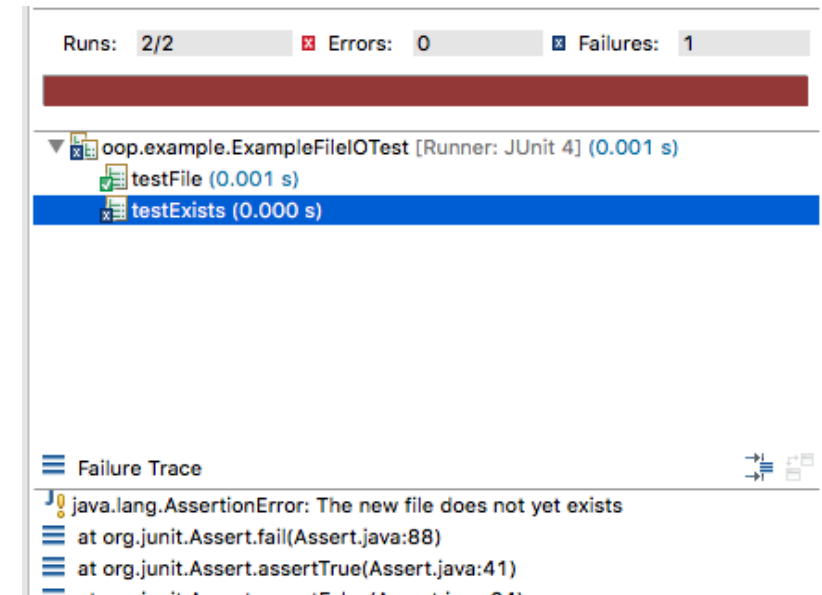
Throws IOException

Problem: Fails Second Time

First Time



Second Time



The testExists test fails

Assertion that the file does not exists fails

Solution: Clean up file after test

Import After tag

Create @After method

Delete f after every test

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    private File f = null;

    @Before
    public void setUp() throws Exception {
        f = new File("/tmp/test.txt");
    }

    @After
    public void tearDown() throws Exception {
        f.delete();
    }

    @Test
    public void testFile() {
        assertTrue("A new file was created", f

```

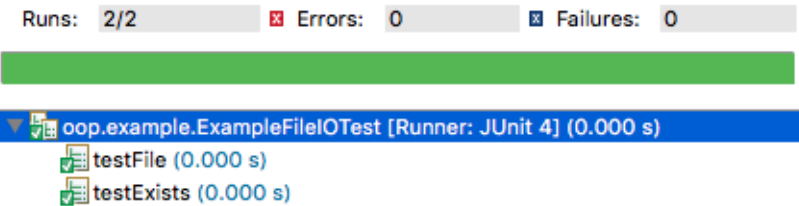
It is safe to delete f after every test because we create it before every test.

Now it works every time

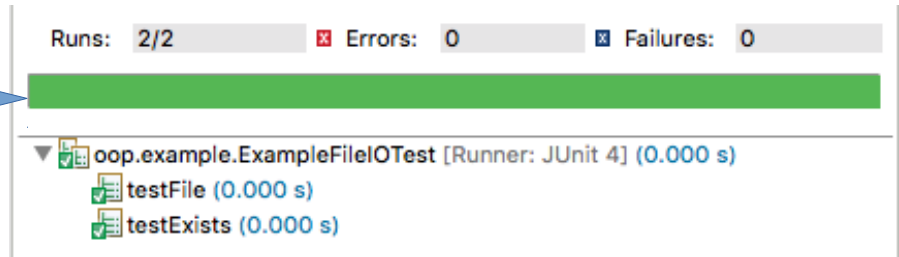
First Time



Second Time



Third Time



Idempotence

- Any time you take an action that changes the state of the computer in a test, make sure to undo that action before completing the test
- *Idempotence* means that multiple applications of a function or method should yield the same results
- Tests must be idempotent
- Methods are guaranteed to be idempotent if they change nothing while running

Test File: tests exists() and delete()

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

import java.io.File;
import java.io.IOException;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class ExampleFileIOTest {

    private File f = null;

    @Before
    public void setUp() throws Exception {
        f = new File("/tmp/test.txt");
    }

    @After
    public void tearDown() throws Exception {
        f.delete();
    }

    @Test
    public void testFile() {
        assertTrue("A new file was created", f instanceof File);
    }

    @Test
    public void testExists() {
        assertFalse("The new file does not yet exists", f.exists());
        try {
            f.createNewFile();
        } catch (IOException e) {
            e.printStackTrace();
            fail("In testExists, f.createNewFile threw IO Exception");
        }
        assertTrue("The new file exists now", f.exists());
    }
}
```


Writing to Text File

Text Files: Writing to a File

- Before writing, make sure either...

- File doesn't exist

- ```
!outFile.exists()
```

- Or file exists, and is writeable

- ```
outFile.exists() && outFile.canWrite()
```

- Combine conditions

- ```
!outFile.exists() || outFile.canWrite()
```

# Text Files: Writing to a File

- Attach file to a stream
  - `PrintStream` object: knows how to write stream to a file

# Creating a PrintStream from a file



# Text Files: Writing to a File

- Once finished writing to the file, close it

```
pWriter.close();
```

# Test Printing

Create a PrintStream from a File

Constructor throws error if directory

Or cannot be opened

Close file after printing

```
@Test
public void testWrite() {
 PrintStream ps = null;
 if (!f.exists() || !f.canWrite()) {
 try {
 ps = new PrintStream(f);
 } catch (IOException e) {
 e.printStackTrace();
 fail("In testWrite, FileWriter(f) threw an exception "
 + "because either f is a directory "
 + "or f cannot be opened");
 }
 ps.print("test");
 ps.close();
 }
}
```

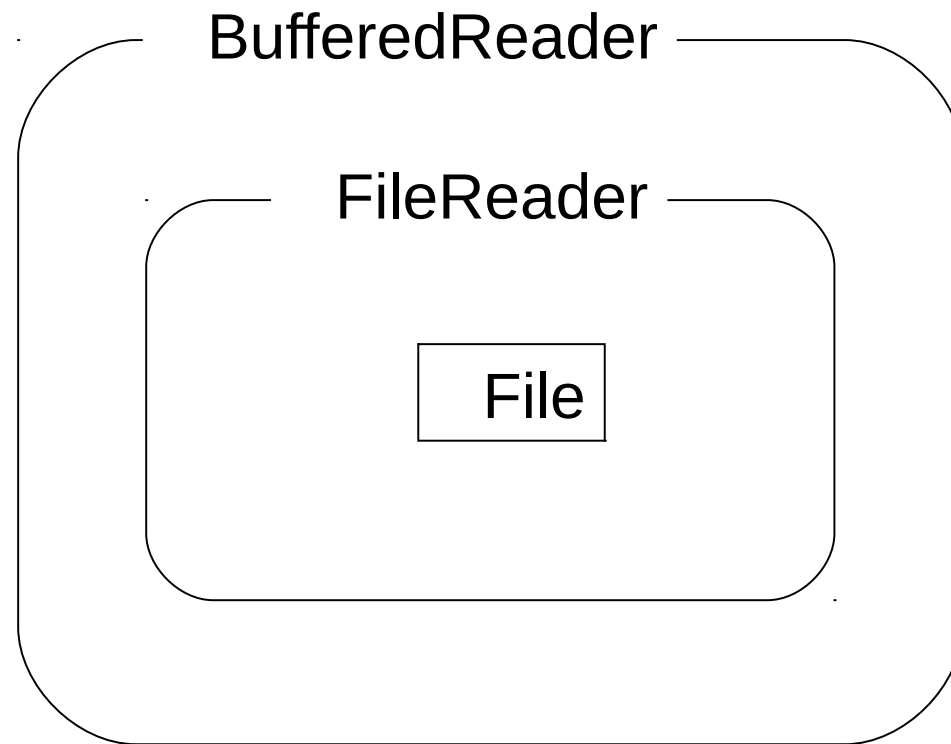
Here we only test that we can open a file for printing and print to the file. We do not test that the character are actually in the file. We will test that when we test read.

# Reading From Test Files

# Text Files: Reading from a File

- Before reading, make sure
  - File exists and is readable  
`inFile.exists() && inFile.canRead()`
- Attach file to a stream
  - `FileReader` object: knows how to read stream from a file
  - Wrap `FileReader` object in `BufferedReader` object
- `BufferedReader` works on files just like `System.in`
  - `read()`: read a single character, or -1 if EOF
  - `readLine()`: read a line, or null if EOF





# Plan to Test File Reading

- Copy the previous test to write “test” in file
- Create a `BufferedReader` from the same file
- Call `readLine()` on the `BufferedReader`
- Check that we get “test” back

# Testing Reading

Create a PrintStream from a File →

Print test →

Create BufferedReader from same File →

Read a line from the file →

Check that string read is string written →

```
@Test
public void TestRead() {
 BufferedReader br = null;
 PrintStream ps = null;
 String s = null;

 if (!f.exists() || !f.canWrite()) {
 try {
 ps = new PrintStream(f);
 } catch (IOException e) {
 e.printStackTrace();
 fail("In testWrite, FileWriter(f) threw an exception "
 + "because either f is a directory "
 + "or f cannot be opened");
 }
 ps.print("test");
 }
 if (f.exists() && f.canRead()) {
 try {
 br = new BufferedReader(new FileReader(f));
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 fail("In TestRead, FileReader threw an exception"
 + " because fi cannot be opened");
 }
 try {
 s = br.readLine();
 } catch (IOException e) {
 e.printStackTrace();
 fail("In TestRead, readLine() threw an exception"
 + " because an I/O error occurred");
 }
 }
 assertEquals("test", s);
}
```

# Note: error message from catch

- The error messages in the catch block are adapted from the Java documentation
- The provide information on why the error might have been thrown
- We can use that information as part of our documentation of the feature we are using
  - It indicates in the code, why the error might have been thrown.

# Refactoring: Extracting Print from Tests

- We now have identical code in two tests, and indication that we should refactor
- Eclipse has a function Refactor > Extract Method that will turn a block of code into a method.
- In addition we will add a parameter of the string we want to write.

# Pull out PrintStream ps = null

- First we need to make the PrintStream we are using a field because we use it in two places

New field



```
public class ExampleFileIOTest {

 private File f = null;
 private PrintStream ps = null;
}
```

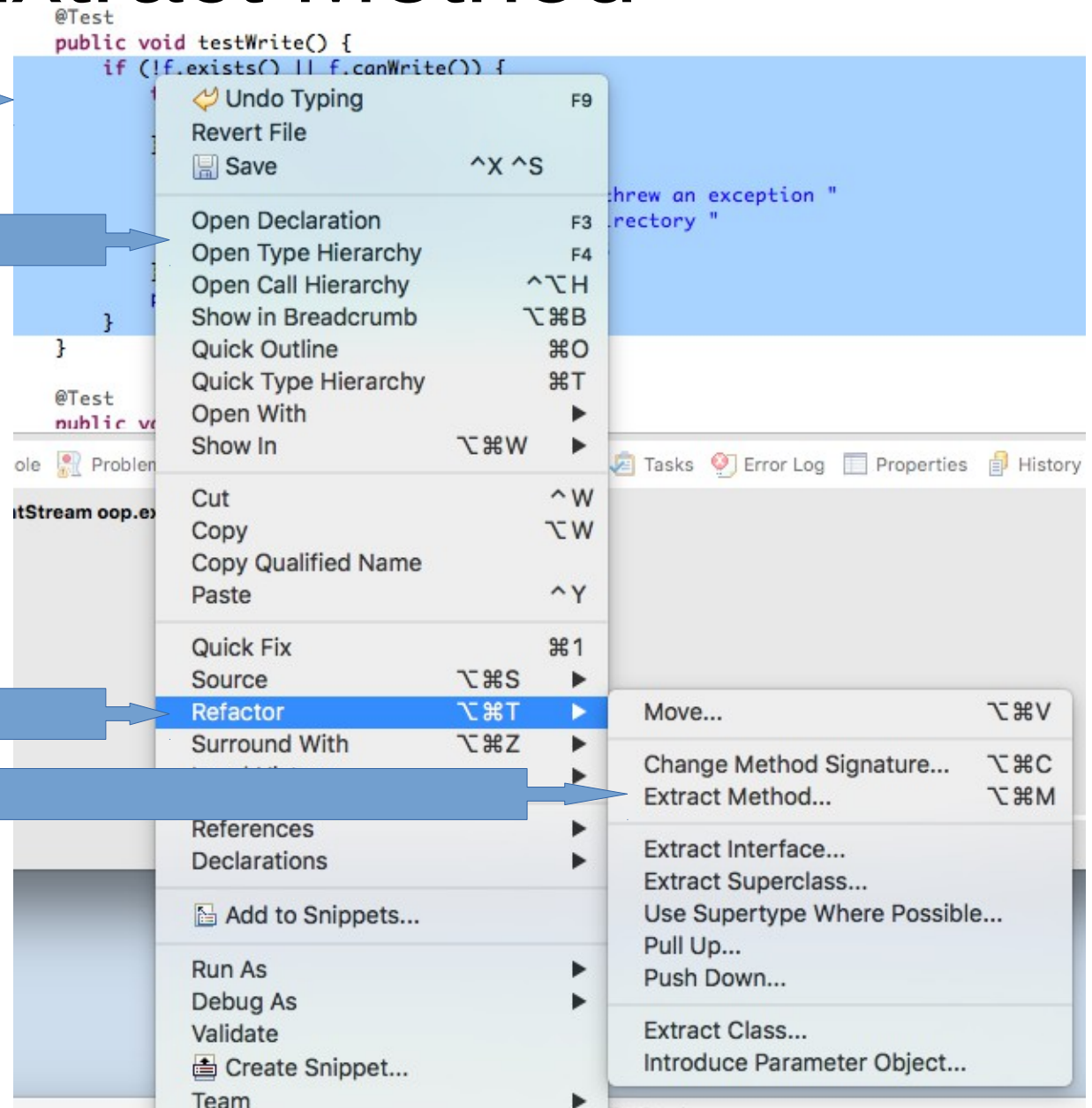
# Extract Method

Highlight code to extract

Right click

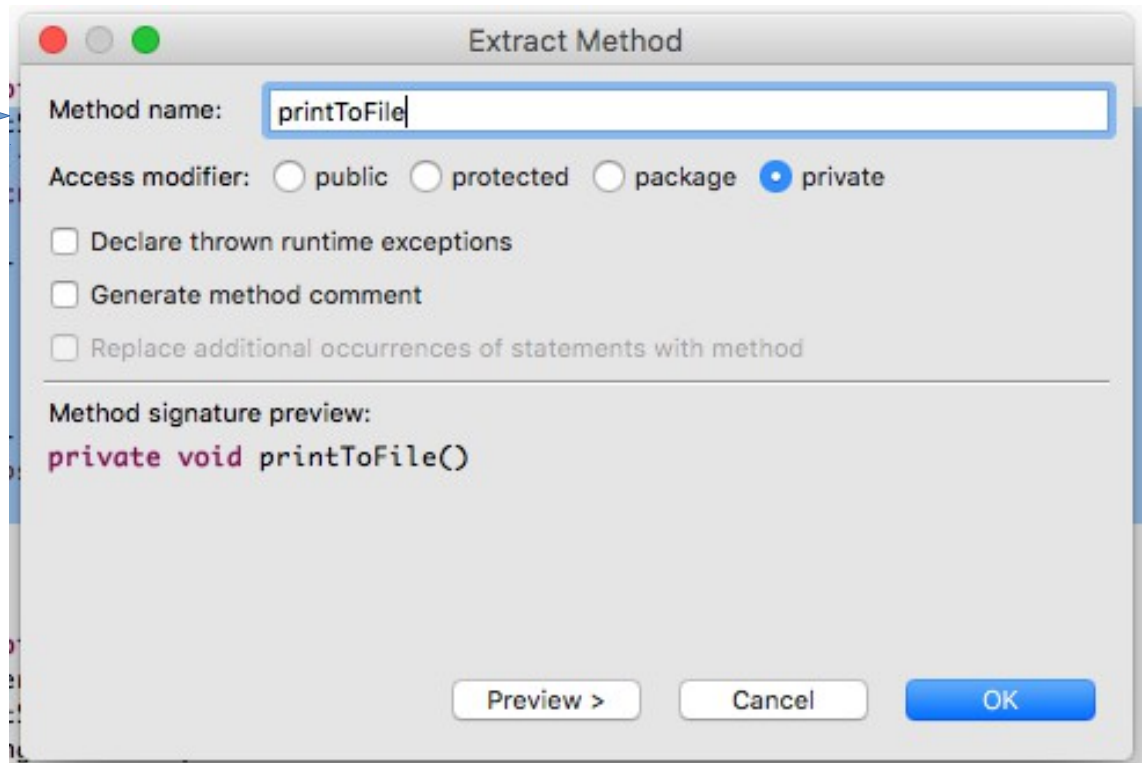
Refactor

Extract Method



# Give the method a name

Name method





# Result

```
@Test
public void testWrite() {
 printToFile();
}

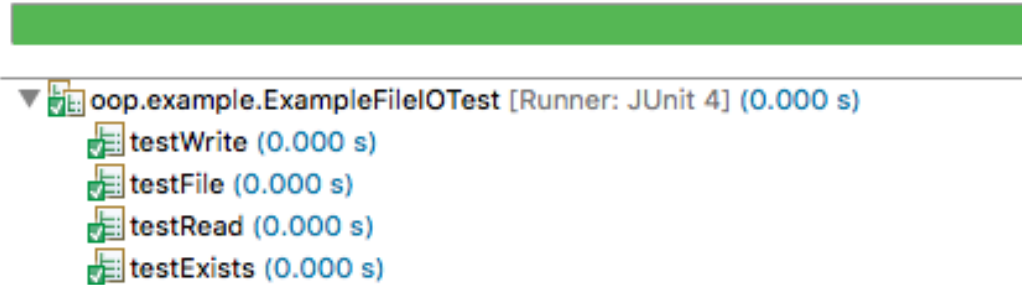
private void printToFile() {
 if (!f.exists() || !f.canWrite()) {
 try {
 ps = new PrintStream(f);
 } catch (IOException e) {
 e.printStackTrace();
 fail("In testWrite, FileWriter(f) threw an exception "
 + "because either f is a directory "
 + "or f cannot be opened");
 }
 ps.print("test");
 ps.close();
 }
}
```

# Fix testRead() to use printToFile()

```
@Test
public void testRead() {
 BufferedReader br = null;
 String s = null;

 printToFile();
 if (f.exists() && f.canRead()) {
 try {
 br = new BufferedReader(new FileReader(f));
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 fail("In TestRead, FileReader threw an exception"
 + " because fi cannot be opened");
 }
 try {
 s = br.readLine();
 } catch (IOException e) {
 e.printStackTrace();
 fail("In TestRead, readLine() threw an exception"
 + " because an I/O error occurred");
 }
 }
 assertEquals("test", s);
}
```

# Tests still run



# Refactoring Improvements

- We have made the code clearer
  - The name of the method indicates what it does
- We have reduce the amount of code
  - The only line of code you can be certain has no bug is the one that isn't there.
- We can change the behavior of printing in all of the tests at the same time

# Giving printToFile a parameter

- Let's change the method so we can pass in a string to test.
- The advantage is we can pass a string in to be written, then check that we get the same string back
  - The payoff is in testRead()

# Changing testWrite()

Called with String parameter

```
@Test
public void testWrite() {
 printToFile("test");
}
```

String parameter added to signature

```
private void printToFile(String s) {
 if (!f.exists() || !f.canWrite()) {
 try {
 ps = new PrintStream(f);
 } catch (IOException e) {
 e.printStackTrace();
 fail("In testWrite, FileWriter(f) threw an exception "
 + "because either f is a directory "
 + "or f cannot be opened");
 }
 ps.print(s);
 ps.close();
 }
}
```

String parameter printed

# Changing testRead()

```
@Test
public void testRead() {
 BufferedReader br = null;
 String sWritten = "This is a test";
 String sRead = null;

 printToFile(sWritten);
 if (f.exists() && f.canRead()) {
 try {
 br = new BufferedReader(new FileReader(f));
 } catch (FileNotFoundException e) {
 e.printStackTrace();
 fail("In TestRead, FileReader threw an exception"
 + " because fi cannot be opened");
 }
 try {
 sRead = br.readLine();
 } catch (IOException e) {
 e.printStackTrace();
 fail("In TestRead, readLine() threw an exception"
 + " because an I/O error occurred");
 }
 }
 assertEquals(sWritten, sRead);
}
```

Variables to read and write

Print write string

Read read string

Assert written and read are same

# Advantages of Parameter refactoring

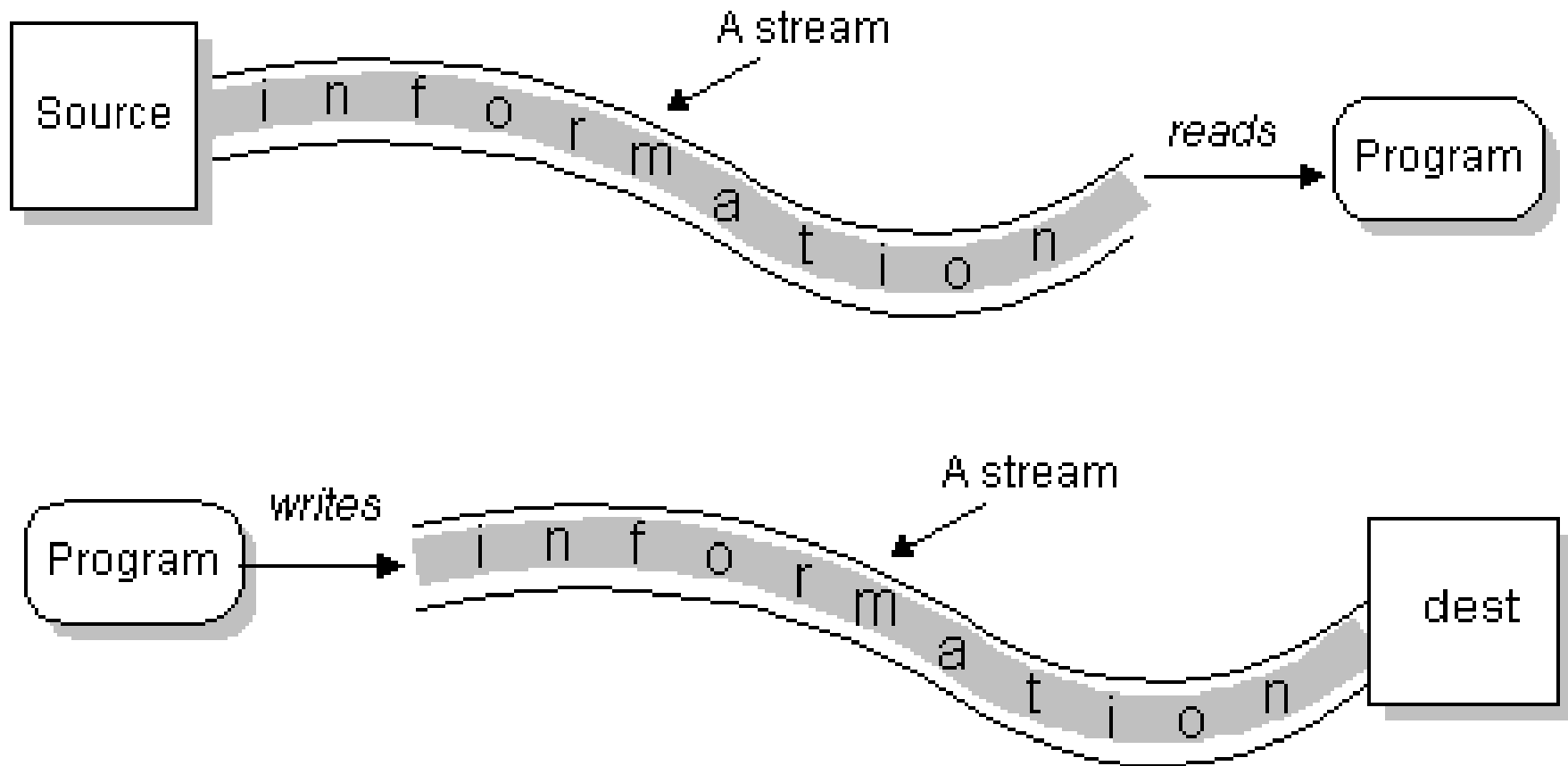
- testRead() is clearer
  - String written and String read are same
- We may be able to use the method in other tests



# Text Files: Reading from a File

- Can do same things we did with `System.in`
  - Read numbers (`NumberFormat`)
  - Read multiple “tokens” (`StringTokenizer`)

# Java I/O Summary



# Java I/O Summary

---

- Reading from Keyboard

- `BufferedReader( InputStreamReader( System.in ) )`

- Writing to Screen

- `System.out`

---

- Reading from File

- `BufferedReader( FileReader( File ) )`

- Writing to File

- `PrintStream( File )`

---