CSC 102 Lecture Notes Week 5
Shallowness and Deepness
Exceptions
File I/O

I. **Relevant reading.**

A. Horstmann Chapter 10, Special Topic 10.6; Chapter 11

B. References cited in lab and program writeups

II. **Shallow and deep, equality and copying**

A. In lecture week 2, we discussed the idea of *shallow* versus *deep* equality.

B. This idea extends to copying objects as well.

C. The Week 5 lecture examples include the following files that illustrate the differences between shallow and deep implementations of important methods:

1. `ShallowCircle.java` -- the perils of shallowness
2. `DeepCircle.java` -- the benefits of deepness
3. `CircleTester.java` -- explaining what goes on

D. Figure 1 is picture of memory after line 23 of `CircleTester.java` has executed.

E. Here are some key points about this picture, and the subsequent execution of `CircleTester.java`:

1. The `ShallowCircle` constructor copies the reference to its `Point` parameter; in contrast, the `DeepCircle` constructor makes a new copy of the `Point` parameter.
2. The `ShallowCircle.getCenter` method returns a reference to its `Point` data field; in contrast, `DeepCircle.getCenter` returns a copy of its `Point` data field.
3. The `ShallowCircle.equals` method compares `Point` references; in contrast, `DeepCircle.equals` does a deep comparison by calling `Point.equals`.

III. **Labs 9 and 10** -- *see the writeups*

A. The Lab 9 topic is ***exception handling***

B. The Lab 10 topic is ***file I/O***

IV. **Exceptions (Sections 11.2, 11.3)**

A. You've used ***throw*** already.

B. For lab 9, and beyond, you'll use ***try-catch*** to ***handle*** exceptions.

C. You'll also write some exception classes of your own.

V. **The basic idea of exception handling.**

A. Normally, a method returns to its caller.

B. *Exceptionally*, a method can ***throw*** an exception.

C. The exception must be ***caught*** in order for the program to continue normally.

D. If a program doesn't catch the exception, Java's runtime will, and your program will then be terminated.

E. To handle an exception, and avoid program termination, you use **`try-catch`**.

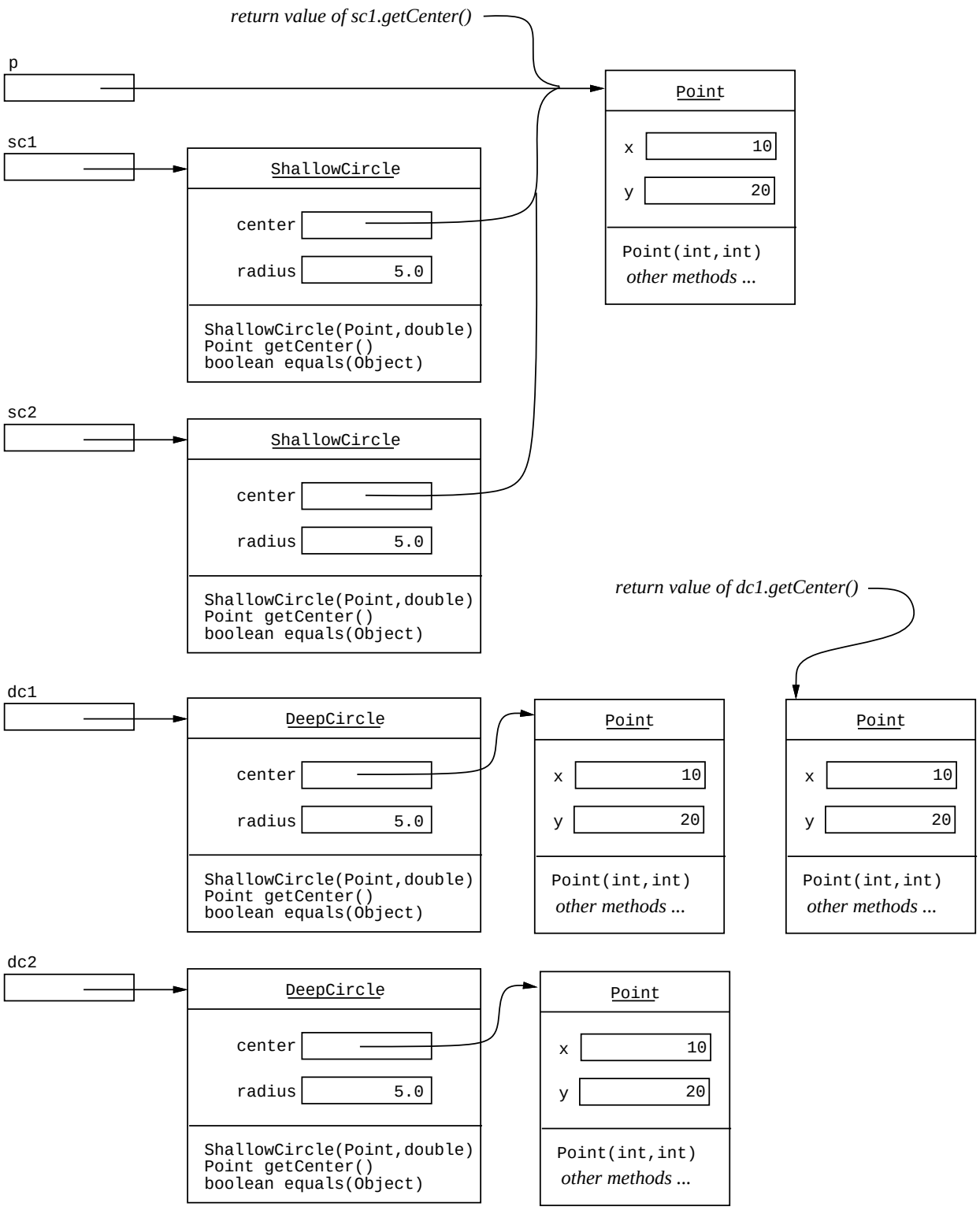

Figure 1: State of memory after line 23 of CircleTester.java.

VI. **The syntax of Java's try-catch**

```
try {
      ... code that may throw
} catch (Exception variable) {
      ... code to handle exception
}
```

VII. **The Throwable hierarchy (Section 11.3)**

A. Exceptions in Java must be defined within the `IThrowable` class hierarchy.

B. Here's an excerpt:

```
Throwable
    Error
        IOError
        VirtualMachineError
            ...
    Exception
        IOException
        RuntimeException
            IndexOutOfBoundsException
            NullPointerException
                ...
            ...
```

C. There's a very nice picture of this on page 482 of Chapter 11 of the book.

VIII. **Checked and Unchecked Exceptions (11.4)**

A. The Java compiler requires handling of checked exceptions.

B. Unchecked exceptions can be ignored.

C. The classes `Error` and `Runtime` exception are unchecked, along with their subclasses.

D. In the words of Java's authors:

1. Class `Error` represents

   "serious problems that a reasonable application should not try to catch"

2. Class `Exception` represents

   "a form of Throwable that indicates conditions that a reasonable application might want to catch"

3. Class `RuntimeException` represents

   "exceptions that can be thrown during the normal operation of the Java Virtual Machine"

IX. **Examples of exceptions**

A. A dumb one, a la lab 8 -- `DumbPrint.java`

B. An uncaught banking exception -- `BankAccountExceptionTester.java`

C. A caught banking exception -- `BankAccountCaughtExceptionTester.java`

D. A caught banking exception, with `finally` -- `BankAccountCaughtFinallyExceptionTester.java`

E. A programmer-defined banking exception -- `NegativeBalanceException.java`

X. **Throws clause in methods (11.5)**

A. A method that throws a checked exception must declare this with *`throws`* in its method header.

B. This allows the compiler to know that the exception is thrown.

XI. **Programmer-defined exceptions (11.7)**

A. Somewhere in the `BankAccount` program:

```
if (amount > balance) {
  throw new
    InsufficientFundsException(...);
}
```

B. Define the exception extension:

```
public class InsufficientFundsExcpetion
    extends IllegalArgumentException {

  public InsufficientFundsException(
        String message) {
    super(message);
  }
}
```

XII. **File I/O (Sections 11.1 and 11.2)**

A. We've seen already how to read from and write to the standard I/O stream that's the UNIX terminal.

B. You can also read from and write to files.

C. With a bit of care, doing this is pretty easy.

D. Here are the highlights:

1. Use a `Scanner` to read from a file, calling its constructor like this:

   `Scanner s = new Scanner(new File(`*filename*`))`

   where *filename* is a `String`.
2. Note well the use of `new File (...)` instead of just the name of the file as a string.
3. Once you have the `Scanner` open on the file, you can use it just like you've done in previous labs.
4. Use a `PrintStream` to output to a file, calling its constructor like this:

   `PrintStream ps = new PrintStream(`*filename*`)`

   where *filename* is a `String`
5. Note that you don't call `new File(...)` around the *filename*.
6. There are a variety of print methods available from `PrintStream`; see its library documentation for details.
7. When you're done with the output, be sure to call the `PrintStream.close()` method, or else your output will not be properly concluded.