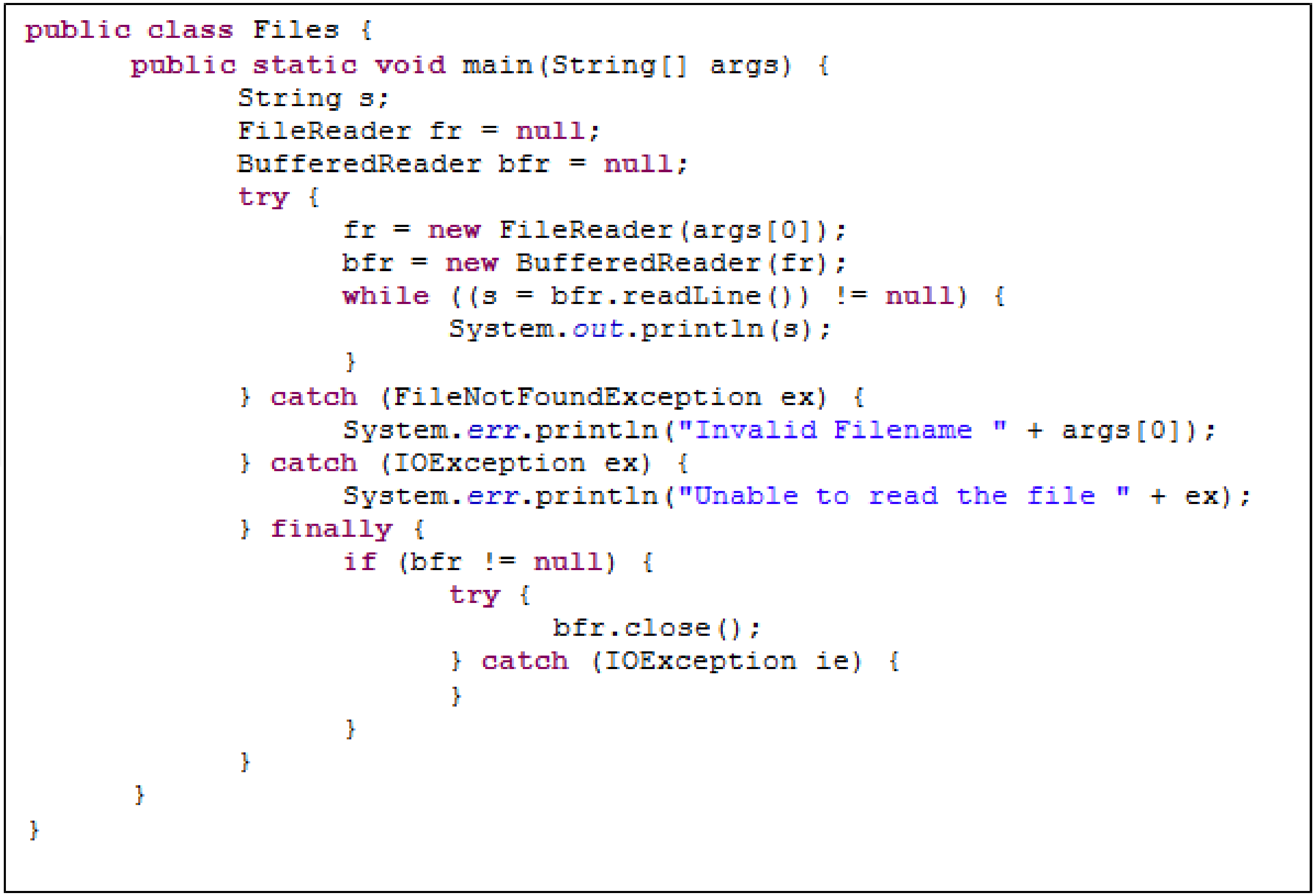
Exceptions

What are Exceptions?

- Exceptions are unexpected run-time situations
  - may indicate an error, or just something unusual
  - allows controlled management of errors
- Java exceptions are objects
  - classes form an extensible exception hierarchy
- Directly supported in the language and VM
- May be caught and ..
  - handled (immediately), or
  - propagated (delayed)

Exception Handling Syntax



```java
public class Files {
    public static void main(String[] args) {
        String s;
        FileReader fr = null;
        BufferedReader bfr = null;
        try {
            fr = new FileReader(args[0]);
            bfr = new BufferedReader(fr);
            while ((s = bfr.readLine()) != null) {
                System.out.println(s);
            }
        } catch (FileNotFoundException ex) {
            System.err.println("Invalid Filename " + args[0]);
        } catch (IOException ex) {
            System.err.println("Unable to read the file " + ex);
        } finally {
            if (bfr != null) {
                try {
                    bfr.close();
                } catch (IOException ie) {
                }
            }
        }
    }
}
```

Using the Exception Object

- Exception define the error condition
  - exception object's class defines general category
  - exception-specific message provides further information

- Useful exception methods
  - defined in exception class or inherited
  - check documentation for additional type specific methods

`toString()`	includes exception name and message
`getMessage()`	message specific to the exception thrown
`printStackTrace()`	dumps stack trace to `System.err` overloaded to dump to other places

Catching Multiple Exceptions

- Substitutability allows multiple related exceptions to be caught

```
try{
...
}catch(IOException ie){
//Handle all IOExceptions
}
```

- Java 7 allows multiple unrelated exceptions to be caught

```
try{
...
catch(FileNotFoundException | NotActiveException ex){
// Do something for these specific exceptions
}
}catch(IOException ie){
//Do something else for other IOExceptions
}
```

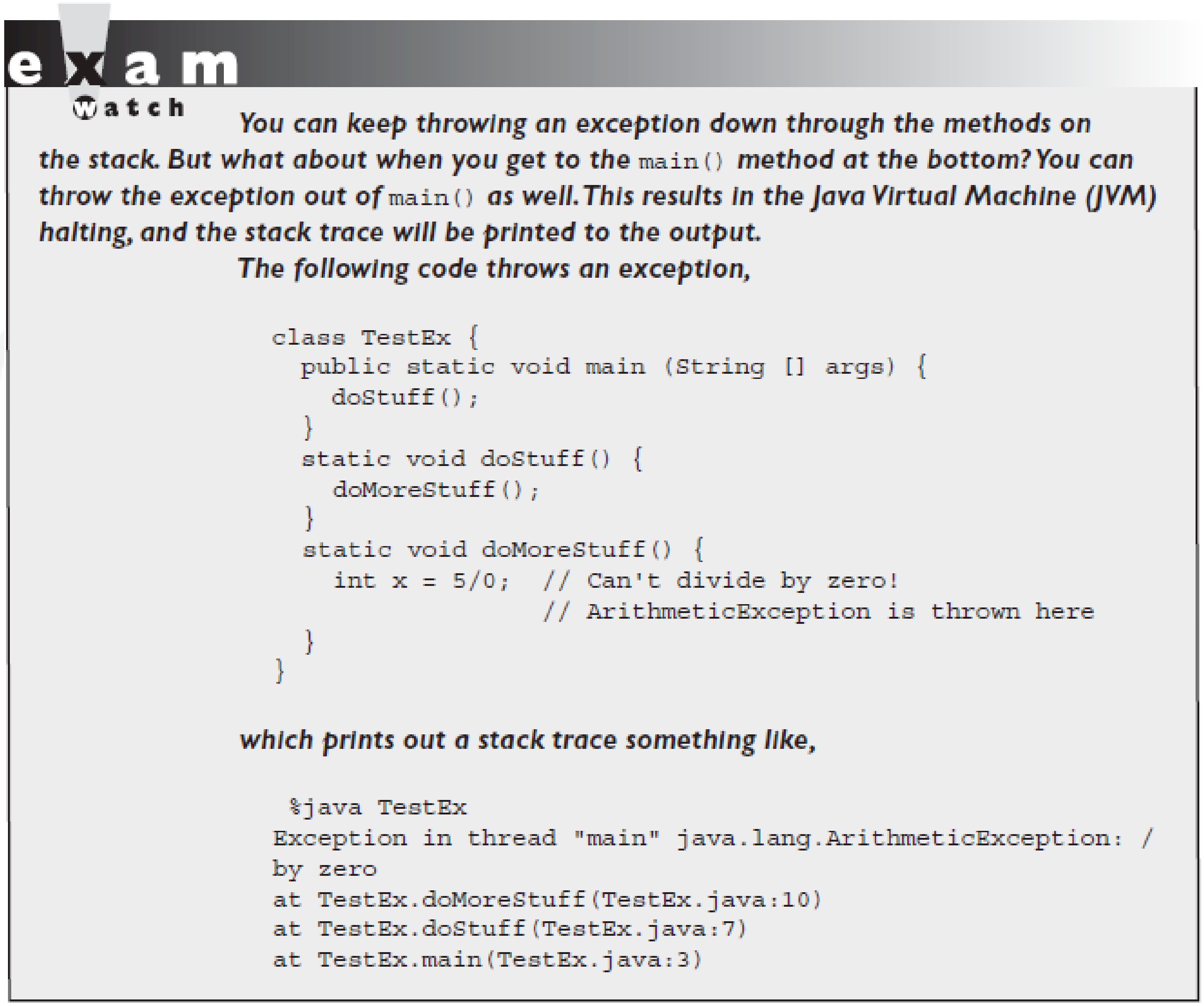
Propagating Exceptions

- Exception need not be fully handled where it occurs
  - declare method as throwing exception
  - now caller must deal with exception

```java
private static void cat(String fname) throws IOException {
      String s;
      FileReader fr = null;
      BufferedReader bfr = null;
      try {
            fr = new FileReader(fname);
            bfr = new BufferedReader(fr);
            while ((s = bfr.readLine()) != null) {
                  System.out.println(s);
            }
      } catch (FileNotFoundException ex) {
            System.err.println("Invalid Filename " + fname);
      } catch (IOException ex) {
            System.err.println("Unable to read the file " + ex);
      } finally {
            if (bfr != null) {
                  try {
                        bfr.close();
                  } catch (IOException ie) {
                  }
            }
      }
}
```

```java
try {
 cat("myfile.txt");
} catch (IOException ie) {
 System.err.println("Problem " + ie);
}
```

Exam Watch



You can keep throwing an exception down through the methods on the stack. But what about when you get to the `main()` *method at the bottom? You can throw the exception out of* `main()` *as well. This results in the Java Virtual Machine (JVM) halting, and the stack trace will be printed to the output.*

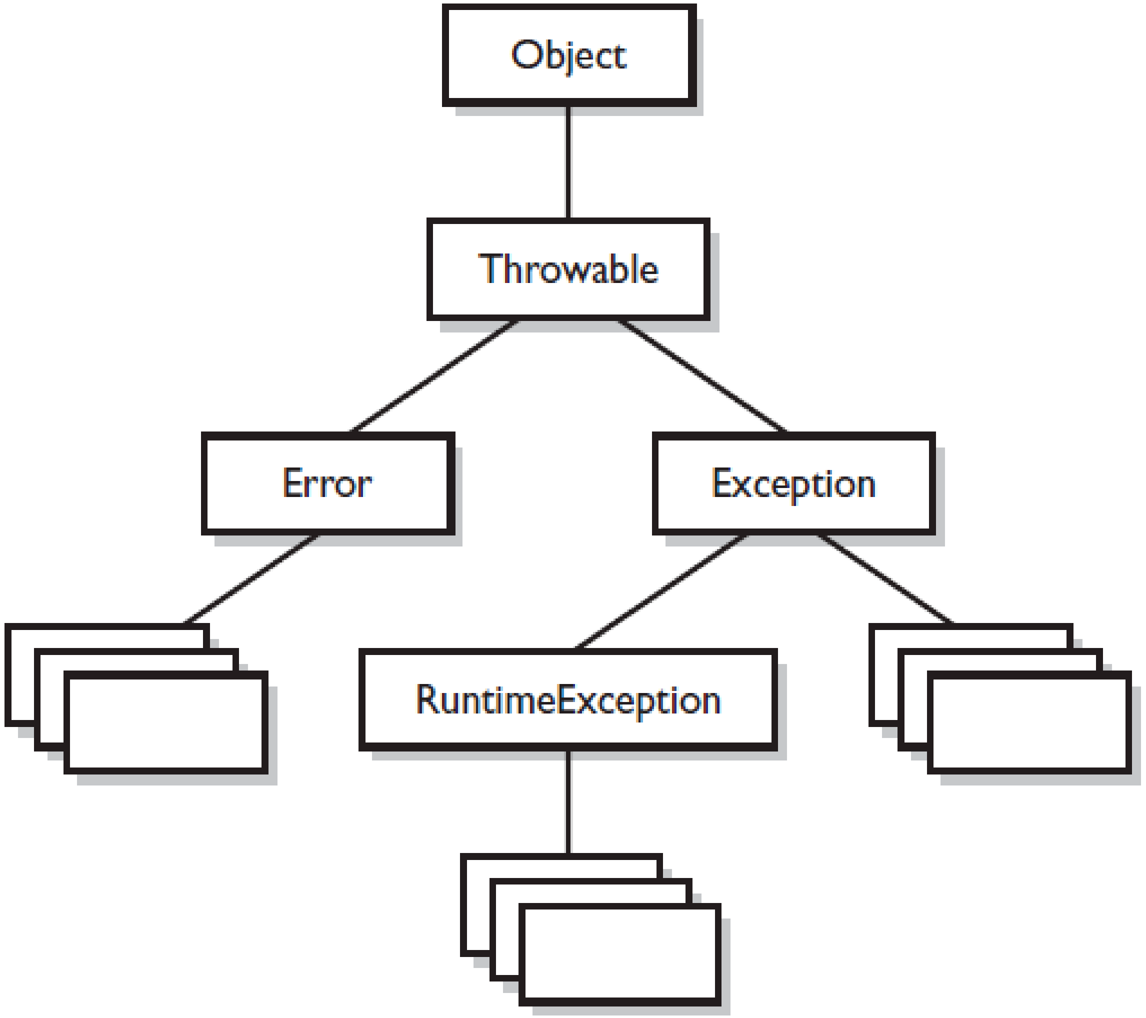
The following code throws an exception,

```
class TestEx {
  public static void main (String [] args) {
    doStuff();
  }
  static void doStuff() {
    doMoreStuff();
  }
  static void doMoreStuff() {
    int x = 5/0;  // Can't divide by zero!
                  // ArithmeticException is thrown here
  }
}
```

which prints out a stack trace something like,

```
 %java TestEx
Exception in thread "main" java.lang.ArithmeticException: /
by zero
at TestEx.doMoreStuff(TestEx.java:10)
at TestEx.doStuff(TestEx.java:7)
at TestEx.main(TestEx.java:3)
```

Exception Class Hierarchy



Exception Class Hierarchy

- Classes that derive from **Error** represent unusual situations that are not caused by program errors(i.e JVM out of memory)

- Not required to handle **Error**

- **Error** are not **Exceptions**

Exception Class Hierarchy

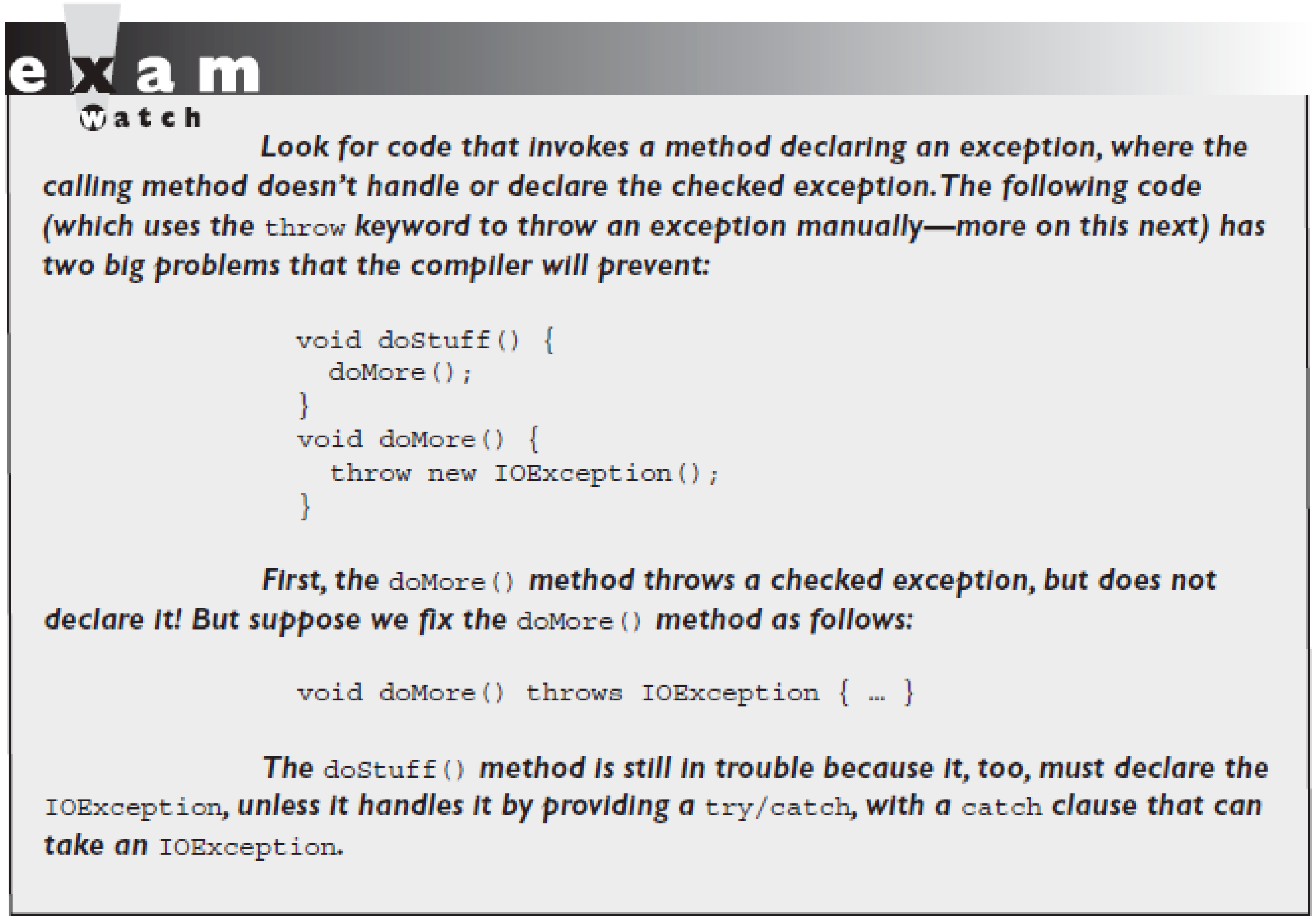
- An exception represents something that happens not as a result of a programming error. For example, network communication

- Exception Matching

- All non-RuntimeExceptions are considered Checked Exceptions. Example: IOException,EOFException,etc

- All RuntimeExceptions, Error are considered Unchecked Exceptions. Example: NullPointerException

Exam watch



Look for code that invokes a method declaring an exception, where the calling method doesn't handle or declare the checked exception. The following code (which uses the `throw` *keyword to throw an exception manually—more on this next) has two big problems that the compiler will prevent:*

```
void doStuff() {
  doMore();
}
void doMore() {
  throw new IOException();
}
```

First, the `doMore()` *method throws a checked exception, but does not declare it! But suppose we fix the* `doMore()` *method as follows:*

```
void doMore() throws IOException { … }
```

The `doStuff()` *method is still in trouble because it, too, must declare the* `IOException`*, unless it handles it by providing a* `try/catch`*, with a* `catch` *clause that can take an* `IOException`.

Question

Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

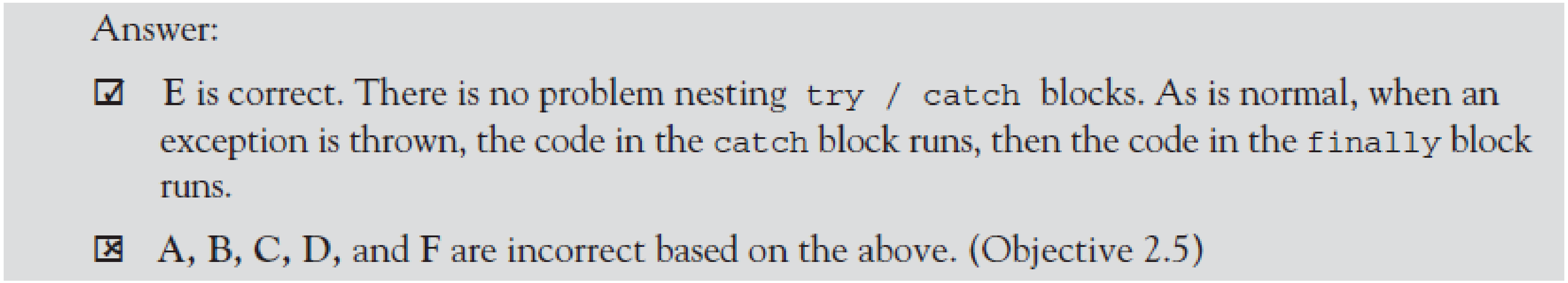
D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

Answer



Answer:

☑ E is correct. There is no problem nesting `try / catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, then the code in the `finally` block runs.

☒ A, B, C, D, and F are incorrect based on the above. (Objective 2.5)

Question

Given:

```
 3. class SubException extends Exception { }
 4. class SubSubException extends SubException { }
 5.
 6. public class CC { void doStuff() throws SubException { } }
 7.
 8. class CC2 extends CC { void doStuff() throws SubSubException { } }
 9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

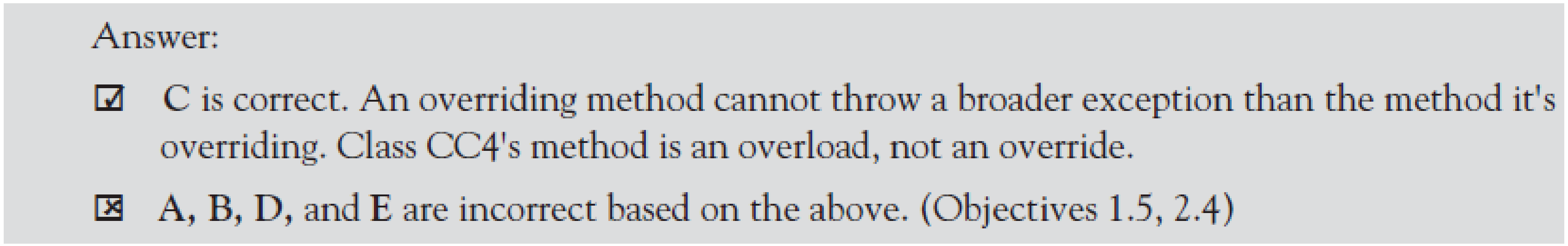
B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

Answer



Answer:

☑ C is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.

☒ A, B, D, and E are incorrect based on the above. (Objectives 1.5, 2.4)

Question

Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I - IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

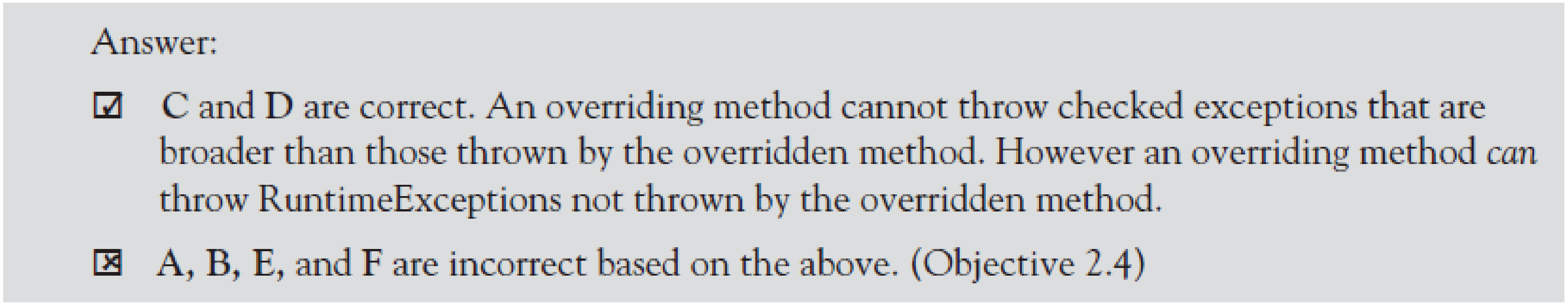
C. Some, but not all, will compile

D. All of those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

Answer



Answer:

☑ C and D are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However an overriding method *can* throw RuntimeExceptions not thrown by the overridden method.

☒ A, B, E, and F are incorrect based on the above. (Objective 2.4)

Contact Info

- trainers@finaldesk.com
- rishabh@finaldesk.com
- nilesh@finaldesk.com
- jignesh@finaldesk.com
- yash@finaldesk.com
- anand@finaldesk.com