Object-Oriented Programming CSE-703029

Faculty of Computer Science
Phenikaa University

Lecture 9: Exception Handling

Today's Topics

- Older, and more modern, strategies for error handling.
- Exception handling basics.
- Some exception details:
  - plain **Exceptions** vs. **RuntimeExceptions**
  - Exceptions containing information
  - Exception hierarchies
- What's really practical using exceptions.

Exception Handling

- The compiler is supposed to report syntax errors, it but can't discover many other types of errors:
  - casts to the wrong type
  - files that don't exist
  - dividing by zero
  - indexing out of bounds
  - incorrect data formats
  - badly formed SQL queries
  - etc. etc.
- Unresolved, many of these errors prevent a program from continuing correctly.

Exception Handling

- We would like to be aware of these “exceptional” situations, in hopes of
  - recovery
  - retrying
  - trying an alternate strategy
  - cleanup before exit
  - or, just finding out where the problem is!
- Nothing is more mystifying than a program that just “goes up in smoke”!

Strategies for Error Handling

- In the course of programming, we constantly test for situations that routinely arise.
- We include logic to deal with the possibilities (switch, if-else, etc.).
- “Exceptional” situations are different. They are things that “should never happen”.
- We expect our code will be free from bugs, but…
- We’re usually wrong.

Strategies for Error Handling

- Pre-testing the arguments to each function call.
- Checking return values indicating error.
- Setting and checking global error variables.
- These are not formal methods, not part of the programming language itself.
- They require programmer discipline (but so does the use of exceptions…).

Example

```
void workOnArray(double[] myArray, int otherInfo) {
    int i = 0;
    // complicated calculation of array index i, using otherInfo
    myArray[i] = 3.14159;
    // what if i is out of bounds?
}
```

Example (cont.)

```
int workOnArray(double[] myArray, int otherInfo) {
    int i = 0;
    // complicated calculation of array index i, using otherInfo
    if (i >= 0 && i < myArray.length) {
        myArray[i] = 3.14159;
        return 0;  // indicating everything OK
    }
    else
        return -1;  // indicating an error
}
```

Potential Problem

- What if **workOnArray()** needs to return a value (say, a **double**)?
- The "C" approach: values are returned through additional reference arguments in the method:

  int workOnArray(double[] myArray, int otherInfo, Double returnValue)

- This quickly gets cumbersome.

Another Technique: Globals

- There are no true global variables in Java, but we "fake it" all the time.
- Write a class with static variables!
- These are effectively available anywhere in a program, and could be used to signal error conditions.

Faking A Global Variable

```
public class MyGlobal {
    public static int indexError;
    MyGlobal() { }  // indexError automatically initialized to 0
}
void workOnArray(double[] myArray, int otherInfo) {
    int i = 0;
    // complicated calculation of array index i, using otherInfo
    if (i >= 0 && i < myArray.length) {
        myArray[i] = 3.14159;
    }
    else
        MyGlobal.indexError = -1;
}
```

Three Important Issues

- Where should the tests be done?
  - Before the array is “indexed into”?
  - By the array class itself?
- Where should the error be reported?
  - Locally, or “further down” in the call stack?
  - Stroustrup says that authors of libraries can’t know their user’s contexts, so can’t know what to do.
- Who is responsible for adjudicating the error?
- Exception handling in Java helps with these problems, but doesn’t completely solve them.

Exception vs RuntimeException

Exception : **Must Throws**, then try..catch..finally

```
import java.lang.Exception;

class MyException extends Exception {
    MyException(String message) {
        super(message);
    }
}
class MyExceptionThrower {
    void f() throws MyException {
        throw new MyException("Throwing MyException");
    }
}
```

Exception

```
public static void main(String[] args){
        MyExceptionThrower t = new MyExceptionThrower();
        try {
            t.f();
        }
        catch (MyException e) {
            e.printStackTrace();
        }
        finally {
            System.out.println("Done");
        }
    }
```

Points

- f() must have “**throws** MyException”. Otherwise, compiler will complain.
- The compiler insists any call to this method be “tested” by enclosing it in a **try** block, or else we get an “unreported exception” error.
- If we *do* include a **try** block , there has to be a corresponding **catch** block or **finally** clause.
- When an exception is thrown, control goes to the matching **catch** block.

Points

- All of this is true because our exception extended the **Exception** class.
- If we extend **RuntimeException** instead, we don’t need to say **throws**, nor include **try** and **catch** blocks.
- **RuntimeExceptions** are special; the Java runtime system takes care of them automatically.

RuntimeException
Exceptions Always Get Caught

```
public class NeverCaught {
    static void g() {
        throw new RuntimeException(“From g()”); //Line 5
    }
    static void f() {
        g(); //Line 8
    }
    public static void main(String[] args) {
        f(); //Line 11
    }
}
```

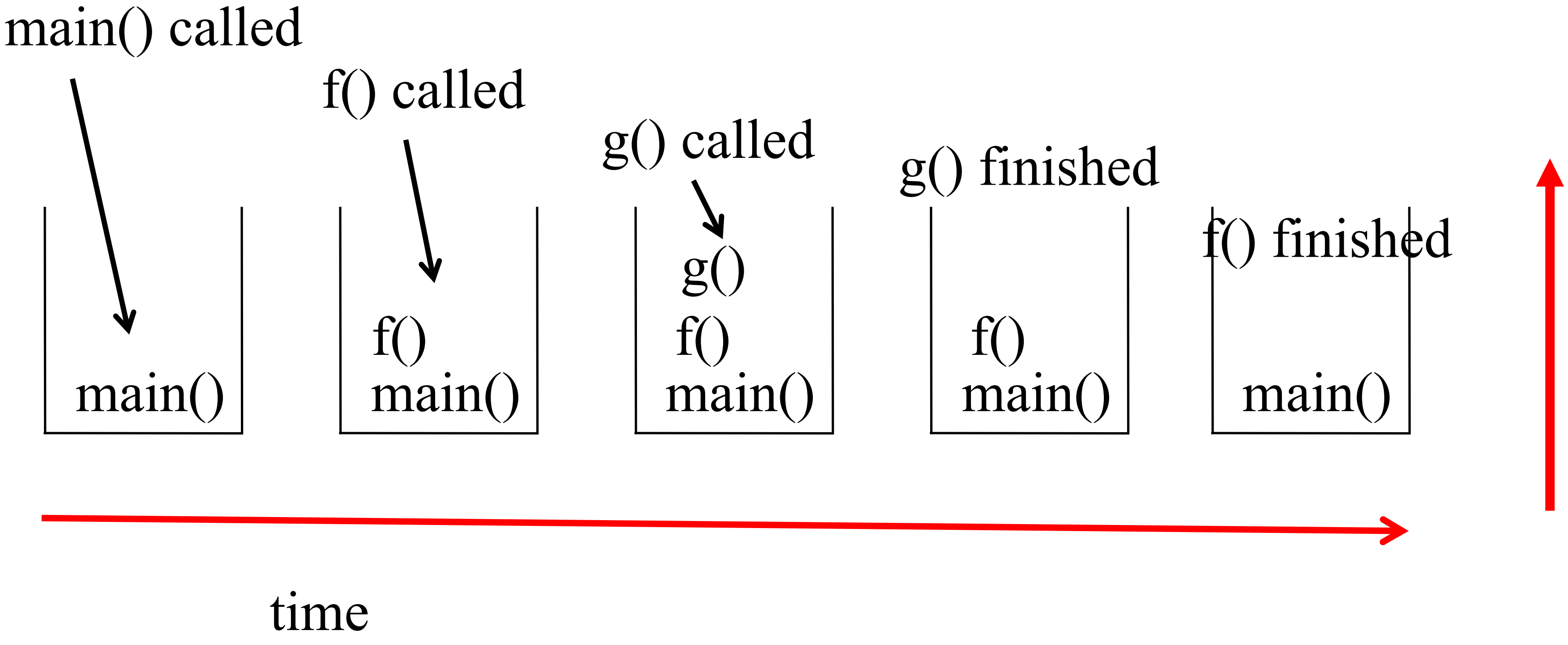
“Uncaught” Exceptions

- If an exception makes it all the way “back” to **main()** without being caught, the Java runtime system calls **printStackTrace()** and exits the program:

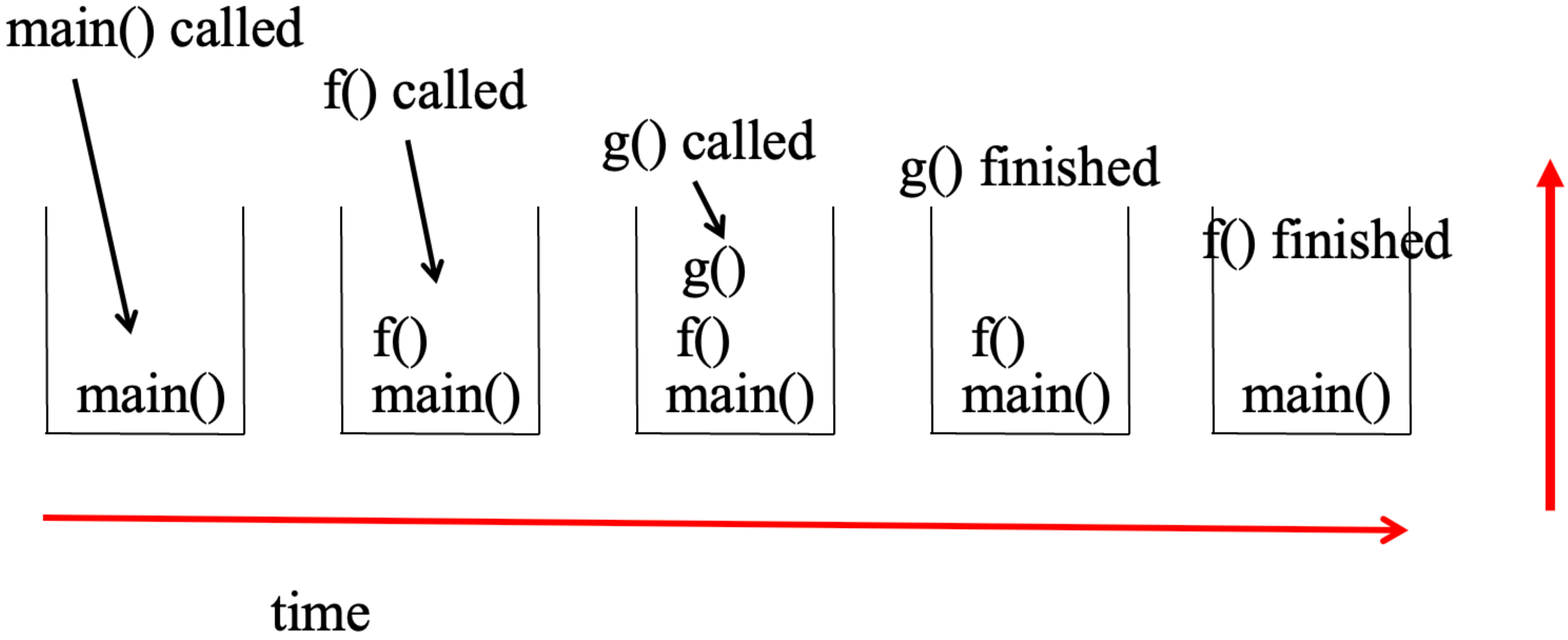
```
java.lang.RuntimeException: From g()
at NeverCaught.f(NeverCaught.java:5)
at NeverCaught.g(NeverCaught.java:8)
at NeverCaught.main(NeverCaught.java:11)
Exception in thread "main"
```

- You can call **printStackTrace()** yourself if you want (and it’s useful to do it).

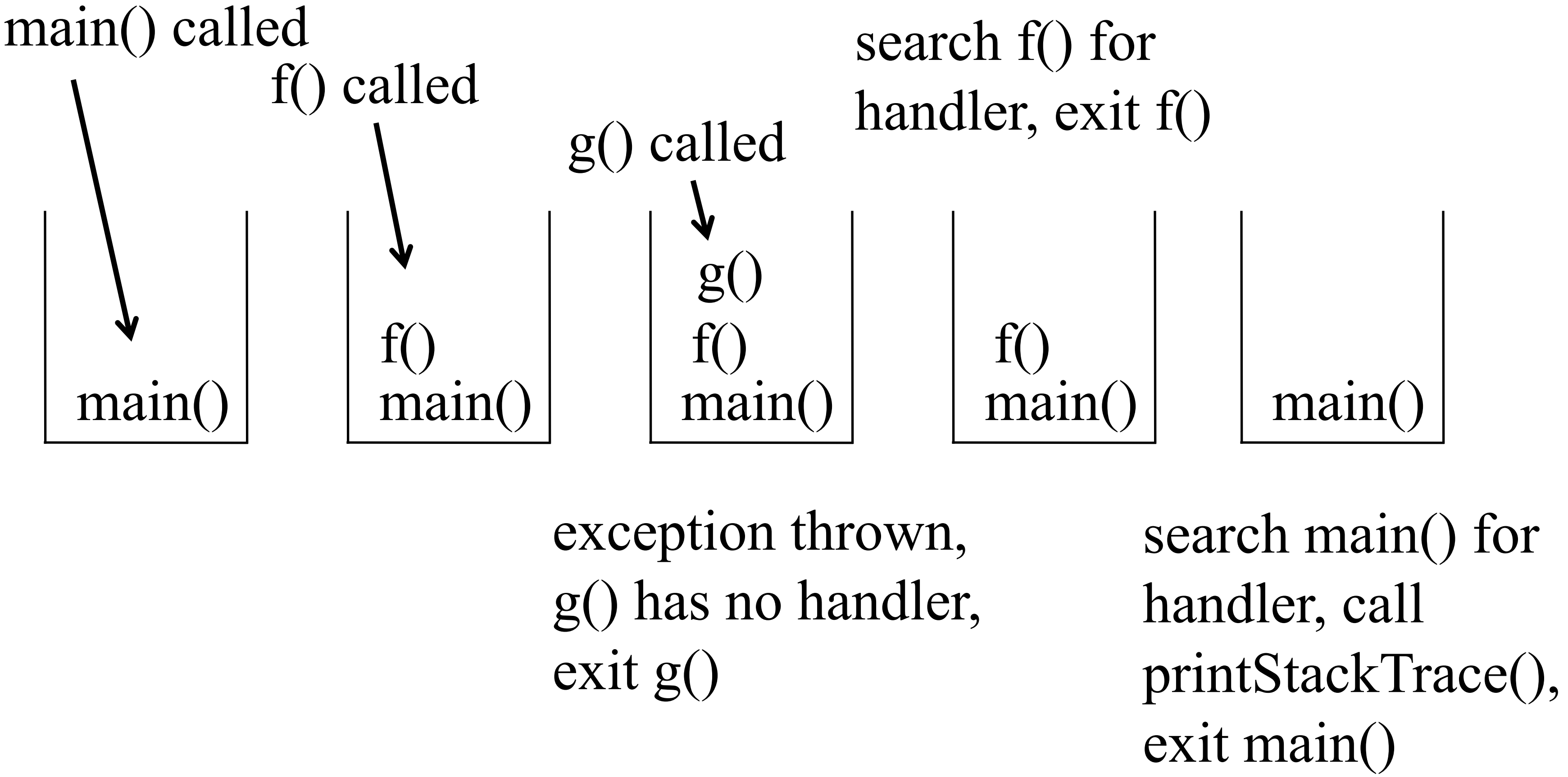
Call Stack, Normal Execution



Call Stack, Normal Execution



Call Stack, With **Exception**



main() called

f() called

g() called

search f() for
handler, exit f()

exception thrown,
g() has no handler,
exit g()

search main() for
handler, call
printStackTrace(),
exit main()

Catching Any Exception

- We are always interested in exceptions that implement the interface **Exception**.
- So, a catch block like

  ```
  catch(Exception e) {
     System.out.println("Caught an exception");
  }
  ```

  will catch any exception.
- If you have multiple catch blocks, this one should be last.

Multiple Catch Blocks

- There may be several possible errors generated by a block of code:

```
try {
   // try this
   // and this
}
catch(YourException e) {
   System.out.println(“Caught exception defined by you”);
}
catch(Exception e) {
   System.out.println(“Caught some other exception”);
}
```

Rethrowing an Exception

- Suppose you've caught an exception, and decided you can't recover from it, but perhaps a higher context can.
- You can rethrow it:

```
catch(Exception e) {
    System.out.println("An exception was caught");
    throw e;
}
```

- The stack trace remains unchanged if it is caught higher up.

Catching, Fixing and Retrying

```
public class Retry {
    static int i = 0;
    public void f() {
        try { g(); }
        catch(Exception e) {
            System.out.println("Caught exception, i = " + i);
            i++;
            f();
        }
    }
    void g() throws gException {
        if (i < 3) { throw new gException(); }
        else
            System.out.println("g() is working now");
    }
```

This Can Be Dangerous

```
public class Retry {
    int i = 0;
    boolean fileIsOpen = false;
    public void f() {
        try {
            if (fileIsOpen)
                System.out.println("Opening an already opened file!");
            else
                fileIsOpen = true;  // i.e., open the file
            g();
            fileIsOpen = false;     // i.e., close the file
        }

        // file will be left open: Dangerous!
```

What’s So Dangerous?

- Just **close** the file in the **catch** block? Good idea! But, what if some other exception were thrown, one that you didn’t catch?

```
catch(gException e) {
    System.out.println("Caught exception, i = " + i);
    i++;
    fileIsOpen = false;
    f();
}
finally {
    fileIsOpen = false;
}
```

Exception Hierarchies

- Exceptions are classes, so can be in inheritance hierarchies. Have constructors and data members.
- The usual polymorphism rules apply.
- A handler for a superclass exception will catch a subclass exception.
- This makes it easy to catch groups of exceptions.
- Instance of Exceptions are real objects (created with new)

```
Exception exp = new Exception();
```

Termination Vs. Resumption

- Java makes it hard to complete this cycle:
  - find a problem,
  - throw an exception,
  - fix the problem in the handler, and
  - go back to *where you left off*.
- This is called “resumption”.
- Java assumes you don’t *want* to go back.
- This is called “termination”.

What You *Can* Do

- Fix the problem and call the method that caused the exception once more.
- “Patch things up” and continue without retrying the method.
- Calculate some alternative result.
- Do what you can in the current context, and rethrow the *same* exception to a higher context.
- Do what you can, and throw a *different* exception to a higher context.
- Terminate the program: Closure and Finally()