Lecture 20

Java

Exceptional Event Handling

Dr. Martin O'Connor
CA166
www.computing.dcu.ie/~moconnor

Topics

- What is an Exception?
- Exception Handler
- Catch or Specify Requirement
- Three Kinds of Exceptions
- What are Checked Exceptions?
- Three Components to Handling Exceptions
- The try, catch, and finally Blocks
- Exceptions Thrown by a Method
- How to Throw Exceptions
- The throw Statement

What is an Exception?

- The Java programming language uses *exceptions* to handle errors and other exceptional events
- What is an exception?

  An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- The term *exception* is shorthand for the phrase "exceptional event."

Throwing an Exception

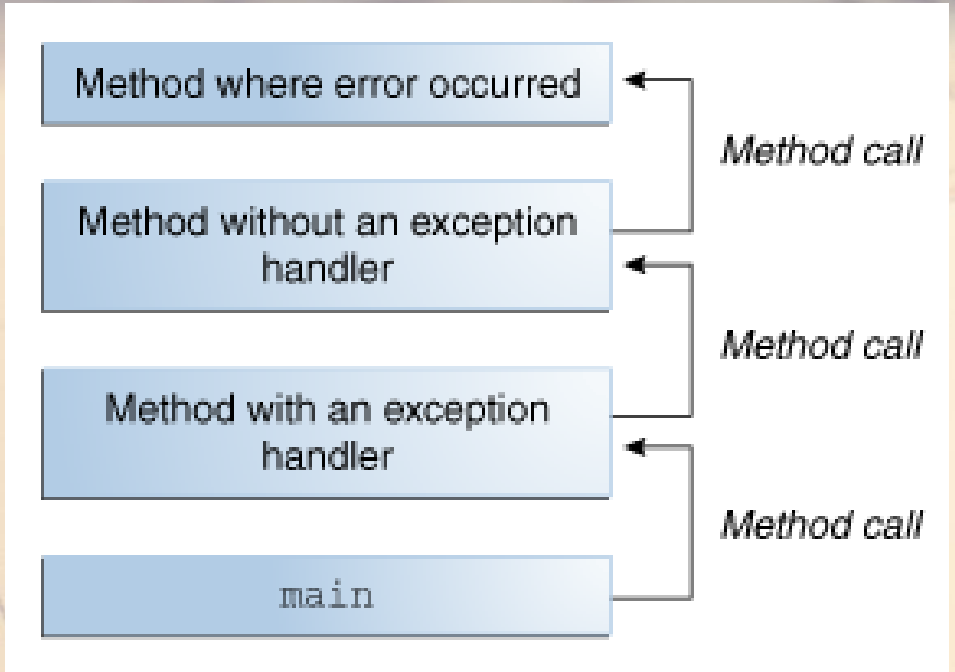
- When an error occurs within a method, the method creates an object and hands it off to the runtime system
- The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*

Throwing an Exception

- After a method throws an exception, the runtime system attempts to find something to handle it
- The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred
- The list of methods is known as the *call stack*

Exception Handler

- The Call Stack



- The runtime system searches the call stack for a method that contains a block of code that can handle the exception
- This block of code is called an *exception handler*

Exception Handler

- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called
- When an appropriate handler is found, the runtime system passes the exception to the handler
- What is considered an appropriate handler?
  An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler

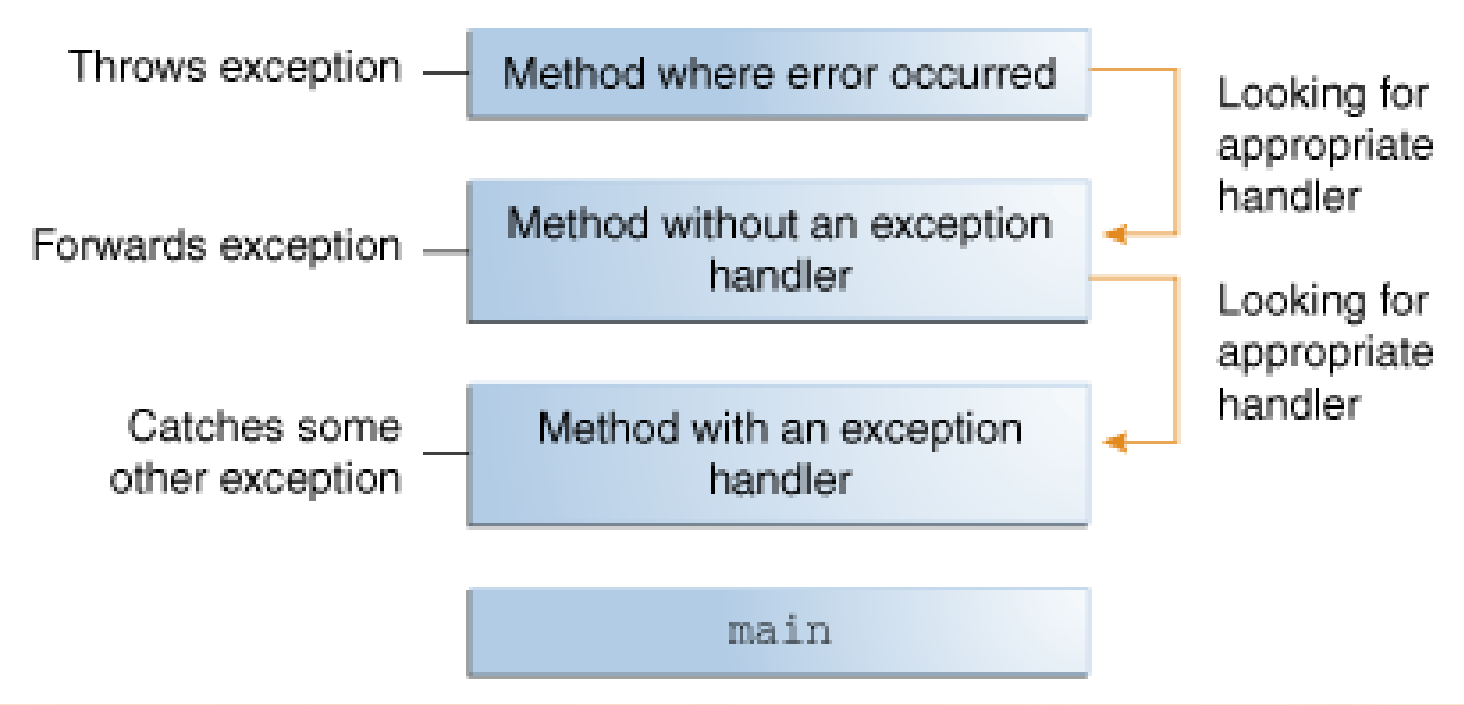
Exception Handler

- The exception handler chosen is said to *catch the exception*

- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates

Exception Handler

- Searching the call stack for the exception handler



Catch or Specify Requirement

- The Java runtime system requires that a method must either catch or specify all checked exceptions that can be thrown by that method
- This means that code that might throw certain exceptions must be enclosed by either of the following:
  - A try statement that catches the exception. The try must provide a handler for the exception (explained later)
  - A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception (explained later)

Catch or Specify Requirement

- Code that fails to honour the Catch or Specify Requirement will not compile
- Note: not all exceptions are subject to the Catch or Specify Requirement
- To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement

Three Kinds of Exceptions

- The first kind of exception is the *checked exception*
- These are exceptional conditions that a well-written application should anticipate and recover from
- For example, attempting to open a file using a user-supplied name of a nonexistent file. This will throw an *java.io.FileNotFoundException*
- A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name

Three Kinds of Exceptions

- The second kind of exception is the *error*
- These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from
- For example, unable to read a file because of a hardware or system malfunction. The unsuccessful read will throw *java.io.IOError*
- Errors are not subject to the Catch or Specify Requirement.

Three Kinds of Exceptions

- The third kind of exception is the *runtime exception*
- These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These include:
  - arithmetic exceptions (such as when dividing by zero),
  - pointer exceptions (such as trying to access an object through a null reference),
  - indexing exceptions (such as attempting to access an array element through an index that is too large or too small)

Three Kinds of Exceptions

- Runtime exceptions *are not subject* to the Catch or Specify Requirement

- Errors and runtime exceptions are collectively known as *unchecked exceptions*

What are Checked Exceptions?

- Checked exceptions *are subject* to the Catch or Specify Requirement
- Note: the compiler ensures that checked exceptions are caught or specified
- **The *unchecked exceptions* classes are the class RuntimeException and its subclasses, and the class Error and its subclasses.**
- **The *checked exception classes* are all exception classes other than the unchecked exception classes.**

Three Components to Handling Exceptions

- There are three exception handler components to writing an exception handler.
  - try block
  - catch block
  - finally block

The try Block

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block
- In general, a try block looks like the following

```
try {
    code
}
catch and finally blocks . . .
```

- The segment in the example labelled *code* contains one or more legal lines of code that could throw an exception

The try Block

- For example, a try block when creating a new file for writing.

```
private static final int SIZE = 10;

PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
}
catch and finally statements . . .
```

- If an exception occurs within the try block, that exception is handled by an exception handler associated with it.
- To associate an exception handler with a try block, you must put a catch block after it

The try Block
Two approaches

- If you have a block of code that might throw several exceptions, there are two approaches to handling them

- You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each

- Or, you can put all the code within a single try block and associate multiple handlers with it

The catch Block

- You associate exception handlers with a try block by providing one or more catch blocks directly after the try block.
- No code can be between the end of the try block and the beginning of the first catch block

```
try {
    statements
    ...
} catch (ExceptionType name) {
    statements
    ...
} catch (ExceptionType name) {
    statements
    ...
}
```

The catch Block

- Each catch block is an exception handler and handles the type of exception indicated by its argument

- The argument type, *ExceptionType*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class

- The handler can refer to the exception with *name*

The catch Block

- The catch block contains code that is executed if and when the exception handler is invoked
- The runtime system invokes the exception handler when the handler is the first one in the call stack whose *ExceptionType* matches the type of the exception thrown
- The system considers it a match if the thrown object can legally be assigned to the exception handler's argument

The catch Block

- An adaption of an example from
  http://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html

```
try {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
    out.close();
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

The catch Block

- An excerpt of an example from http://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html
- Both handlers print an error message. The second handler does nothing else. By catching any IOException that's not caught by the first handler, it allows the program to continue executing
- The first handler, in addition to printing a message, throws a user-defined exception (to be covered later)
- You might want to throw a user-defined exception if you want your program to handle an exception in this situation in a specific way.

The catch Block

- Exception handlers can do more than just print error messages or halt the program
- They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions

The finally Block

- The finally block *always* executes when the try block exits
- This ensures that the finally block is executed even if an unexpected exception occurs
- But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.
- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated

Common Error Trap

If the JVM exits while the try or catch code is being executed, then the finally block may not execute

Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues

The finally Block

- An example of a finally block that ensures the Printwriter stream is closed.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    }
}
```

- Note: All File input and output streams should be closed before exiting the program.
  - The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is *always* recovered

A Complete Example

- An adapted example from
  http://docs.oracle.com/javase/tutorial/essential/exceptions/putItTogether.html

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++)
            out.println("Value at: " + i + " = " + list.elementAt(i));

    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Caught ArrayIndexOutOfBoundsException: "
                           +  e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
    }
}
```

Exceptions thrown by a Method

- If a method doesn't catch the checked exceptions that can occur within it, the method must specify that it can throw these exceptions
- To specify a method can throw exceptions, add a throws clause to the method declaration
- The throws clause comprises the *throws* keyword followed by a comma-separated list of all the exceptions thrown by that method

Exceptions thrown by a Method

- The clause goes after the method name and argument list and before the brace that defines the scope of the method
- An example:

```
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
```

- Remember that *ArrayIndexOutOfBoundsException* is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following

```
public void writeList() throws IOException {
```

How to Throw Exceptions

- Before you can catch an exception, some code somewhere must throw one

- Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment

- Regardless of what throws the exception, it's always thrown with the *throw* statement

How to Throw Exceptions

- The Java platform provides numerous exception classes
- All the classes are descendants of the **Throwable** class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program
- You can also create your own exception classes to represent problems that can occur within the classes you write

The throw Statement

- All methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object

- Throwable objects are instances of any subclass of the Throwable class. Here's an example of a throw statement

```
throw someThrowableObject;
```

The throw Statement

- Let's look at the throw statement in context.
- The following pop method is taken from a class that implements a common stack object.
- The method removes the top element from the stack and returns the object

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The throw Statement

- The pop method checks to see whether any elements are on the stack.
- If the stack is empty (its size is equal to 0), pop instantiates a new EmptyStackException object (a member of java.util) and throws it
- Note that the declaration of the pop method does not contain a throws clause. *EmptyStackException* is not a checked exception, so pop is not required to state that it might occur

Required Reading

- The slides are based on the the Exceptions trail of the Online Oracle Java Tutorial

- The Exceptions trail of the Online Oracle Java Tutorial is required reading. It may be viewed at:

  http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html