Introduction to Object-Oriented Programming

Exceptions

Christopher Simpkins
chris.simpkins@gatech.edu

Erorr Handling Code

Consider this code from Company.java:

```
employees = initFromFile2(new File(empDataFile));
if (null == employees) {
    System.out.println("There was an error initializing employees.");
    System.out.println("Perhaps " + empDataFile + " doesn't exist?");
    System.exit(1);
}
```

- The main logic and error-handling logic are intertwined (not complex in this case, but could be much worse in other cases)
- We have to remember to check for the sentinel value that indicates an error
- We have to remember what the sentinel value is (`null` in this case)
- If we wanted to distinguish between different kinds of errors, we'd have to have multiple sentinel values
- The compiler doesn't force us to handle errors

Exceptions

An exception is

- an event that occurs during the execution of a program that disrupts the normal flow of instructions (Java Tutorial - Exceptions);
- a violation of the semantic constraints of a program;
- an object that you create when an exception occurs.

An exception is said to be

- *thrown* from the point where it occurred and
- *caught* at the point to which control is transferred (JLS §11).

The basic syntax is:

```
try {
    // Code that may throw an exception
} catch (Exception e) {
    // Code that is executed if an exception is
    // thrown in the try-block above
}
```

Using Exceptions

Here's our previous example rewritten to use exceptions:

```
try {
    employees = initFromFile(new File(employeeDataFile));
} catch (FileNotFoundException e) {
    System.out.println("Need an employee data file.");
    System.out.println(e.getMessage());
} ...
```

`initFromFile()` declares that it `throws FileNotFoundException` and some other exceptions:

```
private List<Employee> initFromFile(File empData)
        throws FileNotFoundException, IOException, ParseException {
```

The fact that `initFromFile` declares that it throws checked exceptions (more later) means `javac` will require us to handle the exceptions.

`try` Statements

```
try {
    initFromFile(new File(employeeDataFile));
} catch (FileNotFoundException e) {
    System.out.println("Need an employee data file.");
    System.out.println(e.getMessage());
} ...
```

- If `initFromFile()` does throw a `FileNotFoundException`, control is transferred to the catch block.
- The `FileNotFoundException` object that is thrown from `initFromFile()` is bound to the variable `e` in the catch block.
- The absence of the specified file is a violation of a semantic constraint of the `initFromFile` method (which it propagates from `FileReader` - more later).

Now you know the basics. Let's explore the details ...

Run-Time Exception Handling

Throwing an exception causes a nonlocal transfer of control from the point where the exception is thrown to the nearest dynamically enclosing catch clause.

> *A statement or expression is dynamically enclosed by a catch clause if it appears within the try block of the try statement of which the catch clause is a part, or if the caller of the statement or expression is dynamically enclosed by the catch clause. – JLS §11.3*

More simply, a statement is dynamically enclosed by a catch clause if it is

- contained within the corresponding try block of the catch clasuse, or
- within a method (or constructor) that is called within the corresponding try block of the catch clause.

Identifying a Dynamically Enclosing Catch Clause

```java
public Company(String employeeDataFile) {
    try {
        employees = initFromFile(new File(employeeDataFile));
    } catch (FileNotFoundException e) {
        System.out.println("Missing employee file:" + e.getMessage());
    }
}
private void initFromFile(File empData) throws FileNotFoundException {
    BufferedReader reader =
        new BufferedReader( new FileReader(empData) );
}
```

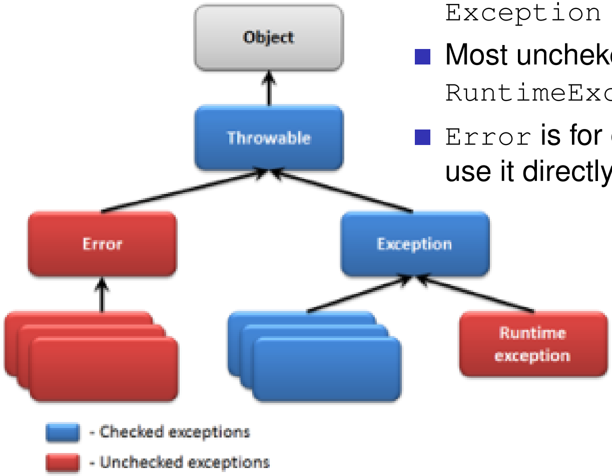
The highlighted statement in `initFromFile` is dynamically enclosed by the highlighted catch clause in the `Company` constructor because it is within a method that is called within the coresposnding try block of the catch clause (and the statement within the `FileReader` constructor that acutally throws the exception is also dynamically enclosed by the highlighted catch clause).

Catch Block Parameters

```
public Company(String employeeDataFile) {
    try {
        employees = initFromFile(new File(employeeDataFile));
    } catch (FileNotFoundException e) {
        System.out.println("Missing employee file:" + e.getMessage());
    }
}
private void initFromFile(File empData) throws FileNotFoundException {
    BufferedReader reader =
        new BufferedReader( new FileReader(empData) );
}
```

- If `new FileReader(empData)` throws a `FileNotFoundException`, it will be caught in the catch block and bound to the catch block's parameter `e`.
- The object `e` has type `FileNotFoundException` and can be used just like any other object.
- `FileNotFoundException` is a standard library exception, so you can look up its API documentation just like any other standard library class

Java’s Exception Hierarchy



- Most (checked) exceptions will subclass `Exception`
- Most uncheked exceptions will subclass `RuntimeException`
- `Error` is for compiler hackers. Don’t use it directly.

Checked and Unchecked Exceptions

Checked exceptions are subclasses of `Throwable` that are not subclasses of `RuntimeException` or `Error`. The compiler requires that checked exceptions declared in the throws clauses of methods are handled by:

- a dynamically enclosing catch clause, or
- a `throws` declaration on the enclosing method or constructor.

This rule is sometimes called "catch or specify" or "catch or declare."

Unchecked exceptions (subclasses of `RuntimeException` or `Error`) are not subject to the catch or declare rule.

Catch or Declare

For example, here are the two ways to deal with the `FileNotFoundException` thrown by `initFromFile`.

Catch:

```
public Company(String employeeDataFile) {
  // ...
  try {
    employees = initFromFile(new File(employeeDataFile));
  } catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
  }
}
```

Declare:

```
public Company(String employeeDataFile) throws FileNotFoundException {
  // ...
  initFromFile(new File(employeeDataFile));
}
```

Throwing Exceptions is a Control Flow Mechanism

What does this code print?

```
public class Wee {

    static void bar() throws Throwable {
        throw new Throwable("Wee!");
    }

    static void foo() throws Throwable {
        bar();
        System.out.println("Foo!");
    }

    public static void main(String[] args) {
        try {
            foo();
        } catch (Throwable t) {
            System.out.println(t.getMessage());
        }
        System.out.println("I'm still running.");
    }
}
```

Taking Advantage of Unchecked Exceptions

We've been using the Scanner class without having to handle exceptions because its methods throw unchecked exceptions. But we can make use of these exceptions to make our code more robust.

```java
Scanner kbd = new Scanner(System.in);
int number = 0;
boolean isValidInput = false;
while (!isValidInput) {
    try {
        System.out.print("Enter an integer:  ");
        number = kbd.nextInt();
        // If nextInt() throws an exception, we won't get here
        isValidInput = true;
    } catch (InputMismatchException e) {
        // This nextLine() consumes the token that
        // nextInt() couldn't translate to an int.
        String input = kbd.nextLine();
        System.out.println(input + " is not an integer.");
        System.out.println("Try again.");
    }
}
```

Multiple Catch Clauses

Two important points in writing multiple catch clauses for a try statement:

- The exception type in a catch clause matches subclasses.
- The first catch clause that matches an exception is the (only) one that executes.

This means that you should order your catch clauses from most specific (most derived, lowest in exception class hierarchy) to least specific (highest in exception class hierarchy).

Writing and Using Your Own Exceptions

Define your own exception classes by subclassing `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).

```
public class MyException extends Exception {

    public MyException(String msg) {
        super(msg);
    }
}
```

And use them just like any other exception:

```
if (checkProblem()) {
    throw new MyException("Oops!");
}
```

But remember: in most cases there is an Exception class in the standard library that you can use. Don't write your own exception classes unless you really need to.

Use The Most Specific Applicable Exception

Recall our `Company` constructor:

```
try {
    employees = initFromFile(new File(employeeDataFile));
} catch (FileNotFoundException e) {
    //...
} catch (ParseException e) {
    //...
} catch (Exception e) {
    //...
}
```

With separate exceptions we can take more specific actions, e.g.:

- We can tell the user to check for the right file (FileNotFoundException).
- We can tell the user that the data file is malformed (ParseException).

Final Thoughts

- Use exceptions for their intended purpose: separating your core logic from the code that handles exceptional conditions.
- Use exceptions judiciously (not too many).
- Think about how you handle exceptions:
  - have sound reasons for propagating exceptions you propagate
  - have sound reasons for catching exceptions where you catch them
  - recover if you can
  - store information in your exceptions to aid in debugging or error recovery by the user