CS122 Lecture: Exceptions

Last revised April 3, 2017

Objectives:

1. Introduce the concepts of program robustness and reliability
2. Introduce exceptions
3. Differentiate between exception throwers, catchers, and propagators

Materials:

1. Online Java documentation to project
2. ExceptionsDemo1 and ExceptionsDemo2. to project, demonstrate
3. Modified ExceptionDemo1.java - showing effect of not catching an unchecked exception
4. Throwable type hierarchy to project.

I. Robustness and Reliability of Programs

A. It is generally desirable that software exhibit two distinct, but related characteristics:

1. We say that a program is reliable if its output is consistently correct for any legitimate input.

2. We say that a program is robust if it can handle illegitimate input and/or other external failures in a reasonable way

   Example: consider a program whose task it is to calculate the square root of a number that the user inputs.

   a) We say that it is reliable if, whenever it is given a number >= 0 (and within the range of the double data type) it produces the correct square root, accurate to the number of decimal places used for the double type.

b) We say that it is robust if it does something reasonable (e.g. pops up a dialog informing the user of the problem) whenever its input is a malformed number (typing “O” instead of “0”) or is negative or is greater than the maximum value for a double.

3. Note the relationship between these concepts: a reliable program does not produce incorrect results without complaint (e.g. outputting 1000 if the user inputs -1); but if it is not robust it may crash in such a case, rather than failing gracefully.

4. Notice that reliability is primarily concerned with avoiding <u>internal</u> errors; robustness is primarily concerned with coping gracefully with <u>external</u> errors.

B. Producing robust and reliable software requires a great deal of care and clear thinking. Today, we will talk about a tool found in Java (and many modern programming languages) that can help in doing so: the exception mechanism.

II. **Exceptions**

A. Consider the various kinds of things that can go wrong outside a program’s control.

ASK

1. The program’s user may input data that is malformed or inconsistent with the program’s requirements.

2. An attempt to access data on an external storage medium (e.g. a disk) may fail due to:

a) Attempting to access a non-existent file.

b) Some violation of the file protection mechanism

c) Attempting to perform an operation that is inherently impossible (e.g. writing to an ordinary CD)

d) Attempting to read past the end of a file.

e) Attempting to write when the storage medium is full.

f) Various hardware problems.

3. Another computer system with which the program communicates over a network may be inaccessible, or may fail during the course of an interaction.

4. etc.

B. To provide for such contingencies, Java utilizes the notion of "throwing an exception".

1. An exception is a special kind of object (belonging to the class `java.lang.Exception` or a subclass of it) that encapsulates information about something that has gone wrong - generally a message and sometimes other information as well.

2. A method that detects an anomaly may "throw" an exception. For example, a statement like the following may appear in a method that attempts to establish a network connection to another system, but is unable to contact it

```
throw new ConnectException("Unable to contact ...");
```

(the class `ConnectException` is defined in the package `java.net`. Like most exception classes, its constructor takes as a parameter a message that describes the cause of the exception)

3. Actually, most exceptions are thrown by classes in the Java API, though it is possible for a program to create and throw its own exceptions as well.

SHOW online documentation for the `parseDouble()` method of class `java.lang.Double`. Note that it can throw a `NumberFormatException` if the number it is asked to parse is malformed.

4. Note that an exception is thrown by the method that detects an exception. Usually, this method does not know how to handle the exception.

   Example: if the `parseDouble()` method of class `java.lang.Double` is given a String parameter that does not represent a legitimate number, it has no knowledge of what to do to ask the user to enter a new one. Perhaps the value is coming from a text field in a GUI, or from console input, or perhaps its read from a file. All that the method can do is detect that something has gone wrong and expect that object that called it to do something appropriate.

5. The other end of the process of throwing an exception is "catching an exception.". Generally speaking, the method that knows how to handle an exception is the one that wants to catch it and do something about it.

   Example: A program that illustrates some simple uses of exceptions: ExceptionsDemo1:

   RUN

   Demonstrate with a legitimate value, "two", and -1

6. PROJECT CODE

   a) If the user enters a malformed number, `parseDouble()` will throw a `NumberFormatException`. That causes computation within the `try` block to immediately terminate (i.e. no attempt is made to validate the value or calculate the square root), and control goes immediately to the `catch` block, where the a message is printed.

b) If the user enters a negative number, then the `throw new ArithmeticException()` statement is executed. Once again, control goes to the `catch` block, where a message is printed, again skipping the square root calculation.

(1) Note that it is possible for Java code you write to explicitly throw an exception.

(2) Actually, a `throw` statement just like this appears in the code for the parseDouble method of java.lang.Double.

c) Once the user enters a legitimate value (hopefully the first time!), the code to actually calculate and display the square root is executed.

C. Sometimes, there are other methods between the method thrower (which detected the problem) and the exception catcher (which knows how to take corrective action.) The intervening methods are called exception propagators.

Example: ExceptionsDemo2.

DEMO

PROJECT Code for evaluate() method

1. evaluate() throws an arithmetic exception if division by zero is attempted. (Note: Java allows division by zero for doubles, because it has an internal representation for infinity! So we have to throw an exception explicitly in this case.)

2. evaluate() propagates a NumberFormat exception if either of the numbers the user typed is malformed.

D. Thus, with regard to a particular exception, a given method may be

1. The <u>thrower</u> of the exception - i.e. it detects and reports the problem, but is not able to handle it itself (e.g. division by 0 in the above example)

2. A <u>propagator</u> of the exception - i.e. it passes on exception information from methods it calls to the method that calls it (e.g. NumberFormat in the above example)

3. A <u>catcher</u> of the exception - i.e. it knows how to handle the exception appropriately (e.g. the ActionListener in ExceptionsDemo2)

4. It is possible for a method to be both the thrower and the catcher of some exception. (e.g. the ActionListener in ExceptionsDemo1).

E. In general, the Java compiler enforces the following constraints:

1. A method that may be a <u>thrower</u> of a given exception - but which does not itself catch it - must declare this fact via a throws clause in its prototype:

   ```
   somemethod(---) throws <type of exception>
   ```

2. A method that is a <u>propagator</u> of a given exception must declare that fact in the same way.

3. A method that calls a method that declares that it throws a certain exception (a thrower or a propagator) must either:

   a) Be itself a <u>propagator</u> of that exception

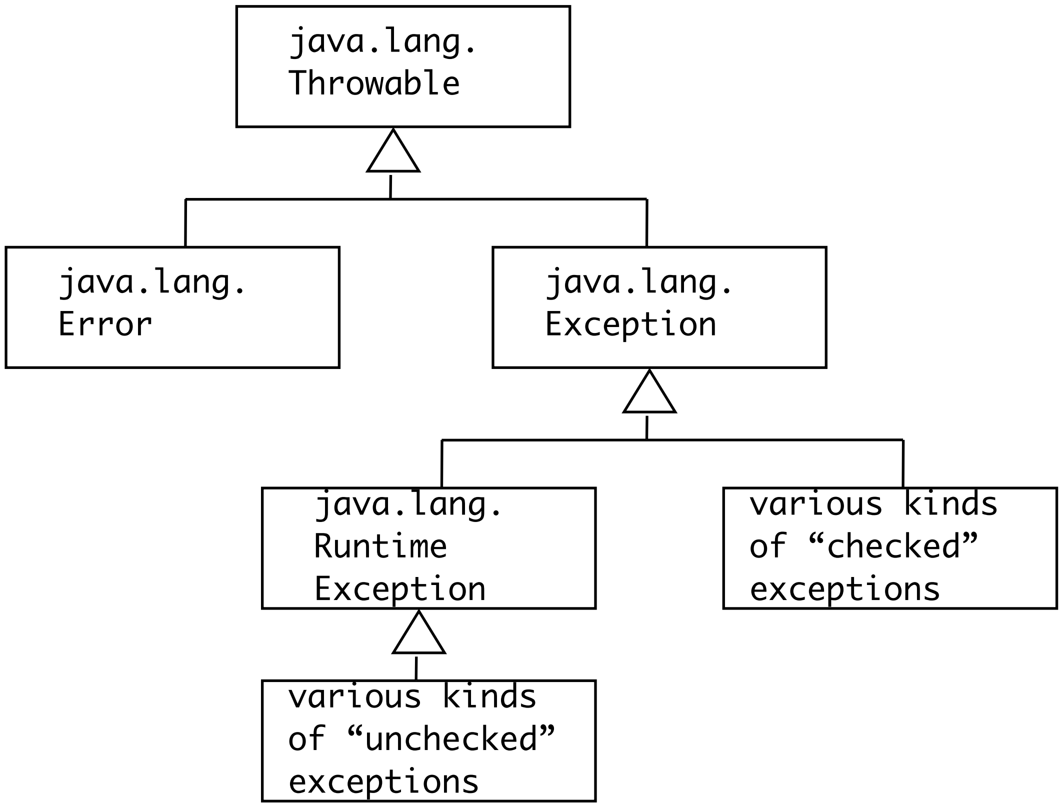
   b) Be a <u>catcher</u> of that exception

4. The combined effect of the above is that every exception that can be thrown is caught by someone - which, in turn, helps to ensure robustness of programs.

III. **Errors and Unchecked Exceptions**

A. There are, two giant "loopholes" in the Java compiler's enforcement of these rules.

The following is part of the hierarchical structure of "throwable" objects



java.lang.Throwable
java.lang.Error
java.lang.Exception
java.lang.Runtime Exception
various kinds of "checked" exceptions
various kinds of "unchecked" exceptions

:

Note that any object that is thrown by a `throw` statement must belong to a subclass of `Throwable`.

1. The class `Error` and its subclasses represent severe problems that should not be caught. The following statement appears in the Java API documentation: "An `Error` is a subclass of `Throwable` that

indicates serious problems that a reasonable application should not try to catch." For this reason, the Java compiler does <u>not</u> require a method that throws an `Error` to declare that fact via a `throws` clause. This, in turn, means that any method that calls another method might itself end up being a propagator of an `Error`, without having to declare that fact.

2. The class `RuntimeException` and its subclasses represent "unchecked exceptions" for which the compiler does not require a `throws` declaration either. The rationale for this is that these represent problems that are so pervasive that practically every method would have to declare them, which would end up littering the code and obscuring the checked exceptions.

   It should be noted that this category includes some types of exceptions that one might wish were checked exceptions - for example `NumberFormatException` is unchecked, though some methods that throw it do declare it even though they are not required to - e.g. `Double.parseInt`.

   Example: Show ModifiedExceptionDemo1.java, which deletes the code to catch a NumberFormatException. Note that it still compiles - because the exception is an unchecked exception. Show what happens if a bad number is entered in this case. (Show output on console as well.)

B. Nonetheless, most important kinds of exceptions do fall into the category of checked exceptions. (Indeed, this category includes some exceptions one might wish were not). And the Java compiler's enforcement of the rules regarding throwers, propagators, and catchers of such exceptions does help to ensure that a programmer will think about how to make a program robust in the face of the errors they represent.

IV. Writing Robust Programs.

A. Accordingly, in order to write robust programs, you should proceed as follows:

1. Anticipate the kinds of things that can go wrong due to external forces (user error, hardware problems, etc.)

2. Be sure that an appropriate exception is thrown when such a problem is detected. (In general, the methods of the standard java libraries do this for you - but there are times when you might want to code a `throw` statement yourself to handle a non-standard problem such as a value that is too small or large.)

3. Be sure your code handles each exception properly.

   a) A method that detects a problem, but does not itself know how to fix it, and so throws an exception, should be declared as a thrower of the exception.

   b) For each problem that can occur, identify the code that is capable of handling the exception appropriately, and incorporate a `try..catch` block in that code, making it a catcher of the exception.

   c) If there are methods between the thrower and the catcher, declare these methods as propagators.

4. Java enforces explicit declaration of throwers and propagators in the case of checked exceptions. If the compiler complains about failure to declare or catch an exception, think through why this is happening and cope with the exception appropriately - don't just put in throws declarations to satisfy the compiler without understanding what's happening.

5. There will be times when you will want to explicitly provide for handling of unchecked exceptions - even though the Java compiler does not force you to - e.g. a `NumberFormatException.`

B. Examples in the Library

Where are exceptions used to produce more robust code in the Library project?

ASK

1. LibraryDatabase throws exceptions in getPatron() and getCopy() if the requested patron or copy does not exist.

2. CheckoutUseCase throws exceptions if an attempt is made to check out a copy that is already out or on the list.

3. Several GUI components catch exceptions like these and report them to the user via a dialog box.