CS112 Lecture: Exceptions and Assertions

Last revised 3/22/04

Objectives:
1. Introduce the concepts of program robustness and reliability
2. Introduce exceptions
3. Introduce assertions

Materials:
1. Online Java documentation to project
2. ExceptionDemo.java to project, demonstrate
3. ExceptionDemoModified.java - showing effect of not catching an unchecked exception
4. Code illustrating exception propagation 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 internal errors; robustness is primarily concerned with coping gracefully with external errors.

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

1. The exception mechanism helps in producing robust software.

2. The assertion mechanism - just added to Java in JDK 1.4 - provides an elegant way for addressing one important issue relevant to producing reliable software. (Prior to JDK 1.4, exceptions were often used for this purpose as well - but the assertion mechanism helps provide a clean distinction between robustness measures and reliability measures.)

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 connect method of class java.net.URLConnection. Note that ConnectException is a subclass of IOException, which is documented as a kind of exception that the method can throw. (There are other things that can go wrong when attempting to make a connection - e.g. a malformed URL or an unknown host, for which there are other specific subclasses of the generic IOException.)

   SHOW online documentation for the parseInt() method of class java.lang.Integer. 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 parseInt() method of class java.lang.Integer is given a String parameter that does not represent a legitimate integer, 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.
C. 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: the following code fragment might have been used in your Project 2 applet to enforce the requirement that the value of hours entered for a clock is a valid integer in the range 1..12 and a minute is a valid integer in the range 0..59. It illustrates throwing an exception and catching two different kinds of exceptions - one thrown explicitly by the code fragment and one by the parseInt() method.

```java
int hours, minutes;
try {
    hours = Integer.parseInt(hoursField.getText());
    if (hours < 1 || hours > 12)
        throw new IndexOutOfBoundsException(
            "Hours must be between 1 and 12");
    minutes = Integer.parseInt(minutesField.getText());
    if (minutes < 0 || minutes > 59)
        throw new IndexOutOfBoundsException(
            "Minutes must be between 0 and 59");
    // Set the clocks to the new time
}
catch(NumberFormatException exception) {
    JOptionPane.showMessageDialog(this,
        "Invalid number format " + exception.getMessage());
}
catch(IndexOutOfBoundsException exception) {
    JOptionPane.showMessageDialog(this,
        exception.getMessage());
}
```

PROJECT / DEMO valid case and each error discussed below

1. If the user enters a malformed number, a parseInt() method call will throw a NumberFormatException. That causes computation within the try block to immediately terminate (i.e. the check on the value of hours or minutes is not done), and control goes immediately to the catch block, where the message encapsulated in the exception is printed. (All exception objects have a getMessage() method.).
2. If the user enters an hour less than 1 or greater than 12, or a minutes less than 0 or greater than 59, then one of the throw statements is executed. Once again, control goes to the catch block, where the message we specified when we threw the exception is printed. (Note, by the way, that if both hours and minutes are invalid, the exception thrown for hours causes the check of minutes to be skipped.)

3. Once the user enters a legitimate value (hopefully the first time!), the code to actually set the clocks is executed.

D. 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: Suppose we had a problem in which there were several different values that needed to be checked to be sure they lay in a valid range. We might create a method like the following:

```java
/** Validity check an integer given a range
 * @param stringForm the string form of the number
 * @param low the lowest valid value
 * @param hi the highest valid value
 * @return the number
 * @exception a NumberFormatException is thrown if the string form of the number is valid;
 * an IllegalArgumentException is thrown if it is too small or too large
 */
public int checkNumber(String stringForm, int low, int hi)
throws NumberFormatException, IllegalArgumentException
{
    int number = Integer.parseInt(stringForm);
    if (number < low || number > hi)
        throw new IllegalArgumentException("Number must lie between " + low + " and " + hi);
    return number;
}
```
This method is a **propagator** of a `NumberFormatException` thrown by `Integer.parseInt`.

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

1. The **thrower** of the exception - i.e. it detects and reports the problem, but is not able to handle it itself (e.g. the above example)

2. A **propagator** of the exception - i.e. it passes on exception information from methods it calls to the method that calls it

3. A **catcher** of the exception - i.e. it knows how to handle the exception appropriately.

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

1. A method that may be a **thrower** of a given exception must declare this fact via a `throws` clause in its prototype:

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

2. A method that is a **propagator** 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 **propagator** of that exception

   b) Be a **catcher** of that exception

4. Note that 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.

G. There are, however, two giant “loopholes” in the Java compiler’s enforcement of these rules.

The following is part of the hierarchical structure of “throwable” objects
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 not 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. Integer.parseInt.

Example: Show ModifiedExceptionDemo.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.)

H. 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.

III. Assertions

A. We said earlier that software designers seek to ensure two related properties of software: robustness and reliability.

B. The exception mechanism provides a good vehicle for dealing with robustness, but providing for orderly handling of anomalous situations that occur as a result of external events (e.g. bad input, failures of other systems). Historically, it has also been used to deal with certain kinds of reliability issues.

C. Many methods have some expectations about the parameters that are passed to them, and can only operate correctly if the parameters meet those expectations.

1. Example: The String class has methods substring(beginIndex) and substring(beginIndex, endIndex). Both require that the beginIndex parameter be >= 0 and <= the length of the string. The second also requires that the endIndex parameter be <= the length of the string and >= the beginIndex parameter. Clearly, it is impossible for these methods to compute meaningful results to one of these conditions is violated.
2. Historically, such problems have often been dealt with by code in the method that tests the parameter values and throws an exception if one is invalid.

Example: Show documentation for the `substring` methods of `String`.

3. Note, though, that the issue being dealt with here is more likely reliability than robustness - i.e. if one of these methods is called with an illegal parameter it is likely to be as a result of incorrect code, not some user error.

D. For this reason, a preferred way to think about issues like these is in terms of method preconditions. A precondition is a requirement that a method imposes on the state of the system (including its parameters) when it is called.

For example, the preconditions of the second version of `substring` might be stated as follows:

```
precondition: 0 ≤ beginIndex ≤ endIndex ≤ length of the string
```

E. Java 1.4 added an `assert` keyword that allows such preconditions to be specified and checked.

Example: `substring` could be coded as follows (it’s not, because it was written long before JDK 1.4!)

```
String substring(int beginIndex, int endIndex)
{
    assert 0 <= beginIndex && beginIndex <= endIndex &&
            endIndex <= length();
    ...
}
```

1. The compiler translates the assertion into code that checks to see if the required conditions hold, and this code is executed each time the method is called.

2. If the stated requirements do not hold, an `AssertionError` (which is a subclass of `Error`) is thrown, terminating the program with a message describing the assertion that failed. Since a failed assertion indicates an error in the logic of the program, the program’s author needs to fix the error that resulted in calling a method with its precondition not satisfied.
3. It is possible to compile a program with assertion-checking turned off (in fact, this is the default), which means that the code to test assertions can be excluded from “production” versions of the software - which avoids the time taken to compute the checks as well as the possibility that the user may be surprised by a weird error message!

4. It is also possible to run a program with assertion checking turned off (the code is there but it is ignored). Again, this is currently the default.

F. Note well: exceptions should be used for detecting anomalies in the runtime environment, including incorrect input. Assertions should be used for checking for flaws in program logic.

1. Note that it is likely that a program flaw that would cause an assertion to fail will either result in some sort of exception or incorrect results if assertion checking is turned off. The point of using assertions is to find and correct such problems before the software is released!

2. Assertions should not be used to validate input, because assertion checking can be turned off when a program is compiled - something one does not want to do with input validation.

3. However, it may be appropriate to use an assertion to detect an invalid situation that is the result of failure to properly validate input