Exceptions

Lecture 15
CGS 3416 Spring 2017

April 12, 2017

What is an Exception?

- An exception is an object that represents an error or *exceptional* event that has occurred.
- These events are usually errors that occur because the run-time environment has detected an operation that is impossible to carry out.
- Exception objects are all children of the `Throwable` class.
- Exceptions represent normal error events that can occur in your program.
- Examples:
  - Array index out of bounds - `IndexOutOfBoundsException`
  - Open a file that does not exist - `FileNotFoundException`
  - Call a method that does not exist - `NoSuchMethodException`

Types of Exceptions

Exceptions generally come in two flavors:

- **Normal Exceptions (checked exceptions)**
  - These exceptions are the ones that every good program should watch for (for example, the `FileNotFoundException`.
  - You have to handle these (either catch them or declare that your method can throw them).
- **Runtime Exceptions (unchecked exceptions)**
  - These exceptions have the potential to be in all code you write (example - `IndexOutOfBoundsException`).
  - You do not need to handle these.
- **Errors**
  - There is a class of exceptions called errors these are usually not recoverable (example - `VirtualMachineError`).
  - These exceptions do not need to be handled.

Some Common Built-In Exception Types

- `ClassNotFoundException` - raised if you attempt to use a nonexistent class.
- `CloneNotSupportedException` - raised on an attempt to call clone() for an object that doesn't implement the `Cloneable` interface.
- `RunTimeException` - numerous types of programming errors that usually cause the program to abort.
  - `ArithmeticException`
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - others
- `IOException` - raised on input/output errors. Several subtypes.
- `AWTException` - raised to deal with graphics errors.

You can also build your own exception types. These should be derived from class Exception, or from one of its subclasses.

Why have exceptions?

- Exceptions are used to build robust programs.
- Exceptions allow the programmer to recover from an error or exceptional event.
- Usually, if an exception is not handled, it can cause the program to terminate unnaturally and prematurely.
- Java was originally a language for embedded systems (TVs, phones, watches, etc.) These systems should never stop working, exceptions are needed for these systems.

How do you do exception handling?

The process involves:

- **Claiming exceptions** - each method needs to specify what exceptions it expects might occur
- **Throwing an exception** - When an error situation occurs that fits an exception situation, an exception object is created and *thrown.*
- **Catching an exception** - Exception handlers (blocks of code) are created to handle the different expected exception types. The appropriate handler *catches* the thrown exception and performs the code in the block.

Claiming Exceptions

In a method, to claim an exception, use the keyword `throws` and list the exceptions that may occur in the method. Examples:

```
public void myMethod() throws IOException
```

```
public void yourMethod() throws IOException, AWTException,
BobException
```

Throwing Exceptions

Use the keyword throw, along with the type of exception being thrown. An exception is an object, so it must be created with the new operator. Examples:

```
throw new BadHairDayException();

MyException m = new MyException();
throw m;

if (personOnPhone != bubba)
     throw new Exception("Stranger on the phone!!");
```

Notice that this is different than the keyword throws, which is used in claiming exceptions.

Catching Exceptions

- Any group of statements that can throw and exception, or a group of statements that you want to watch for Runtime or Error exceptions, must be within a **try** block. At the end of the `try` block there must be either a `catch` or a `finally` block.
- A **catch** block has a parameter that is the type of exception this catch block will handle. There can be several `catch` blocks for a `try` block. If an exception is thrown then the first catch block that that has a parameter matching the exception's type will be the one that catches the exception.
- A **finally** block is ALWAYS executed no matter how control leaves a `try` block. This will happen even if a return statement is executed in the `try` block, and even if control passes to a `catch` block.

Example

```
try
{
    IO code opening and reading from/to files
}
catch (FileNotFoundException)
{
    tell the user and probably repeat try block
}
catch (IOException)
{
    blanket catch for all other IO problems
}
finally
{
    make sure to close any files that might be open
}
```

What happens if an exception is not caught?

- If your method does not catch a checked exception and does not declare that your method can throw it then the compiler will complain.
- If your method throws an exception, then the method that called your method must handle the exception or declare that it can throw that exception.
- If no method handles the exception then the program crashes and a message is printed out describing the exception.
- The same happens if an unchecked exception should occur.
- The only difference between a checked an unchecked exception is that checked exceptions must be handled.

Rethrowing exceptions

- Writing code to handle exceptions is tedious and often you have no idea what to do for error recovery.
- It is sometimes easier just to re-throw the checked exception as an unchecked exception.
- Example:

```
catch (Exception e)
{
     throw new RuntimeException(e);
}
```

When to use exceptions?

- Exceptions are not appropriate for all error-checking tasks.
- Exceptions are good for situations in which the error doesn't need to be handled in the same block where it occurred.
- Conventional error-checking is better for simple tests. For example, validating user input falls into this category – it's best to test user input values with simple if-statements and loops.
- Exceptions are good for handling errors that would result in termination of the program, otherwise.

Instance methods in exception objects

- Exception objects are created from classes, which can have instance methods.
- There are some special instance methods that all exception objects have (inherited from `Throwable`):
    - `public String getMessage()` – returns a detailed message about the exception.
    - `public String toString()` – returns a short message describing the exception.
    - `public String getLocalizedMessage()`
    - `public void printStackTrace()`