CSC220
Chapter 1: Object-Oriented Programming and Class Hierarchies
Part 2

Mitsunori Ogihara

University of Miami

September 2, 2010

Outline

Program Defects and “Bugs”

- An efficient, but erroneous program is worthless!
- Errors are found often after software is delivered.
- Rigorous testing alone cannot demonstrate correctness.
- Sometimes testing itself is very difficult.
- Debugging is the process of removing defects

Types of Errors

1. **Syntax errors**: mistakes in the grammar of the language
2. **Runtime errors or exception**: problematic situations that occur during execution that make it impossible for the program to continue
3. **Logic errors**: due to incorrect observation and conception of the programmer, sometimes do not result in runtime errors

Syntax Errors

- Unfortunately, it is easy to write syntactically incorrect programs. Some common syntax errors are:
  - Omitting or misplacing braces
  - Performing an incorrect type of operation on a primitive type value
  - Invoking an instance method not defined
  - Not declaring a variable before using it
  - Providing multiple declarations of a variable
- Fortunately, the Java compiler is designed to detect syntax errors during compilation.
- A programmer must fix them to be able to execute the program.

Run-time Errors or Exceptions

- Examples of run-time errors are:
  - Division by zero
  - Array index out of bounds
  - Number format and Input mismatch error
  - Null pointer exceptions
  - A named file doesn't exist
- These cause the Java Virtual Machine to throw an exception

Logic Errors

- Programs that are free of syntax and run-time errors may have logic errors
- Logic Errors
  - The “best” case: errors occur in a part of the program that is always executed
  - The “worst” case: errors occur in rare executions
- Sources of Logic Errors
  - Often occur during the design phase due to incorrect algorithm
  - Sometimes typographical errors

Examples of Run-time Errors or Exception

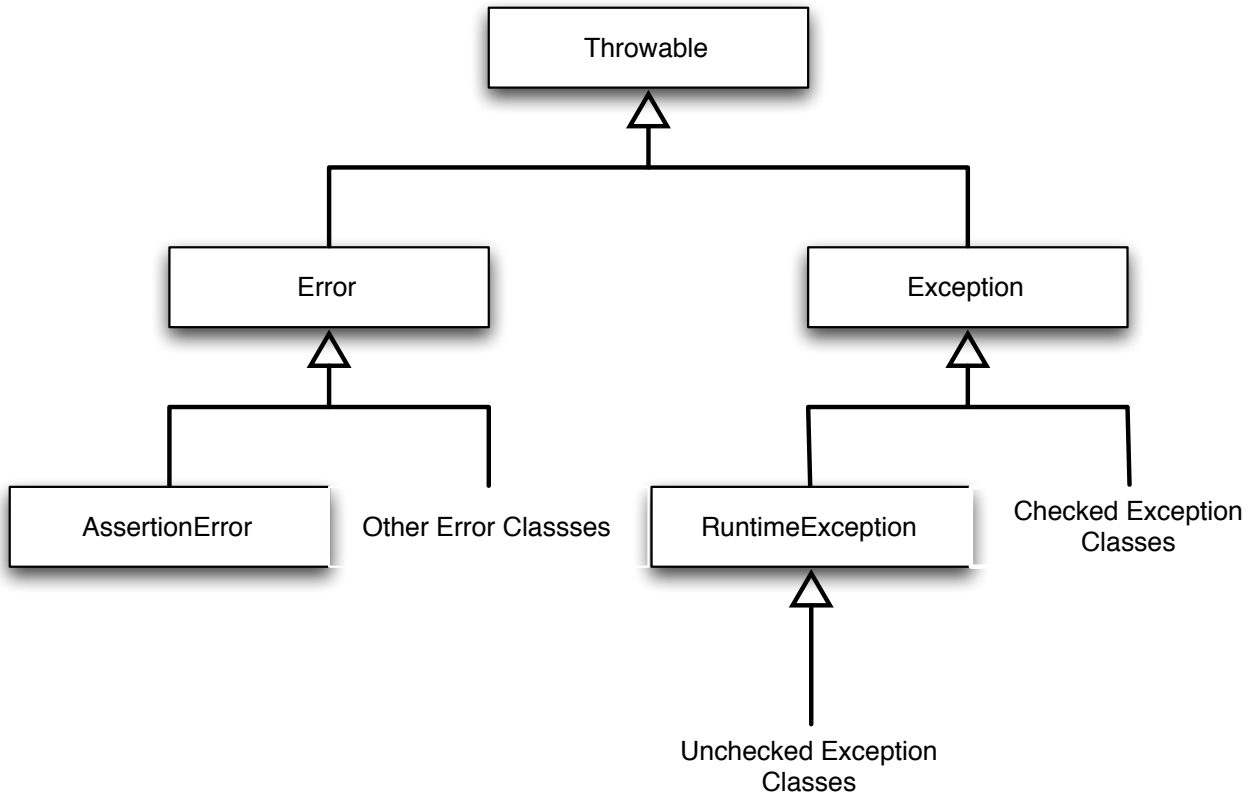
Exceptions	Causes/Consequences
ArithmeticException	Division by zero
ArrayIndexOutOf BoundsException	An attempt to access an array element outside the scope of the index of an array
IllegalArgument Exception	A call of a method with an incorrect type of argument
NumberFormat Exception	An attempt convert a string that is not representing numeric to a numeric
NullPointer Exception	An attempt to access an object pointed to by a null pointer
NoSuchElement Exception	An attempt to obtain the next token when there is no token remaining
InputMismatch Exception	The token returned by a Scanner data object's next method doesn't match the required format

Exception Hierarchy

- When an exception is thrown, one of the Java exception classes is instantiated
- Exceptions are defined within a class hierarchy that has the class Throwable as its superclass
- Classes Error and Exception are subclasses of Throwable
- Error is a class of serious problems that should not be caught by try-catch-finally (to be discussed later)
- RuntimeException is a subclass of Exception

The Class Throwable

- Throwable is the superclass of all exceptions
- All exception classes inherit the methods of Throwable



The Class Throwable

Summary of commonly used methods from java.lang.Throwable class

Method	Action
`String getMessage()`	Returns the detailed message
`void printStackTrace()`	Prints the stack trace to System.err
`String toString()`	Returns the name of the exception followed by the detailed message

Checked and Unchecked Exceptions

- The Java compiler imposes two categories of exceptions: checked and unchecked
  - Checked exceptions: the errors that the Java compiler enforces the program to be aware of the possibility of occurrences
    - All of these belong to Exception
  - Unchecked exceptions: the res
    - These included Error class objects

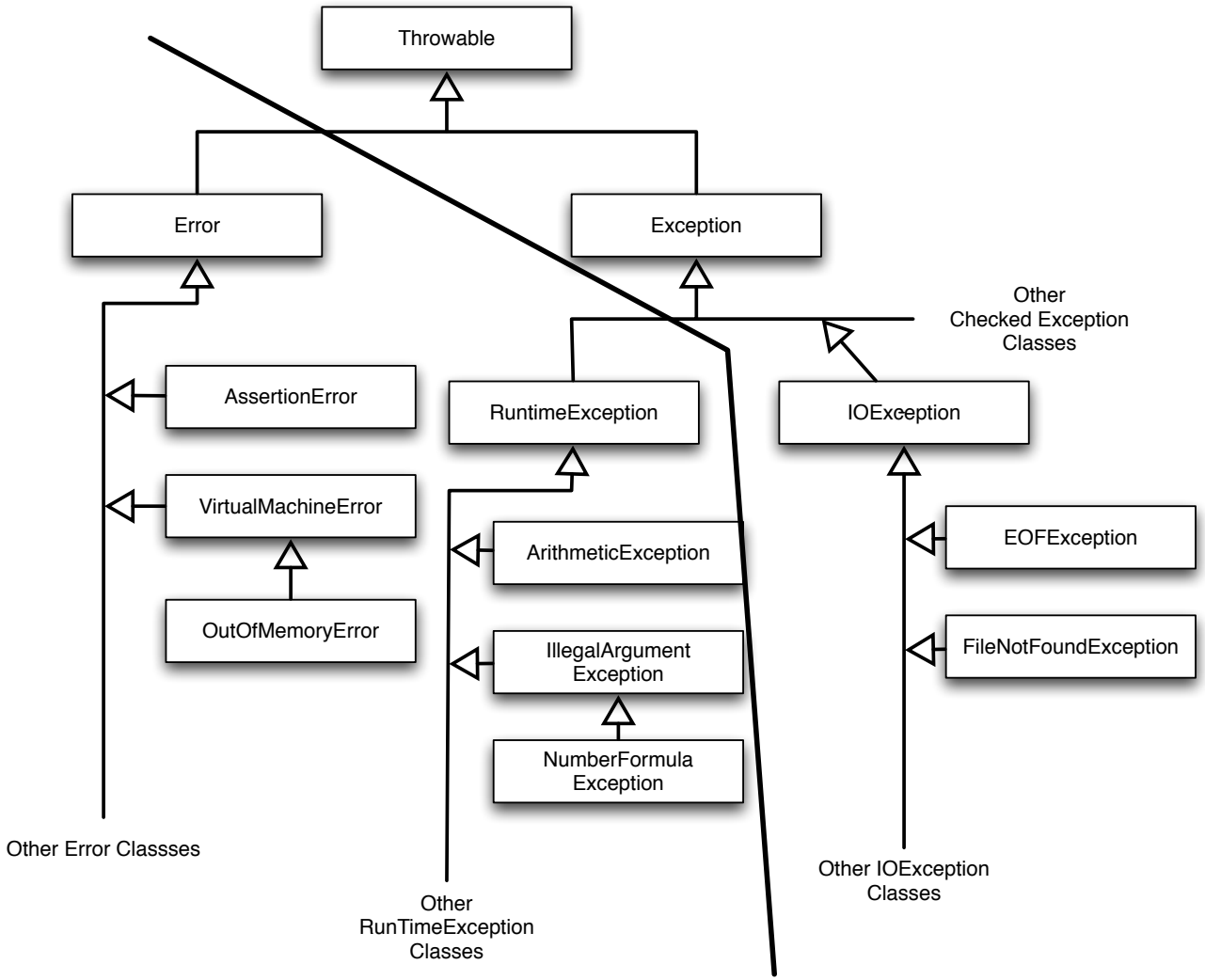
Checked and Unchecked Exceptions

- Checked exception is normally not due to programmer error and is beyond the control of the programmer
- Unchecked exception may result from
  - Programmer error
  - Serious external conditions that are unrecoverable

Some Checked Exceptions

Exception Class	Cause
`IOException`	Some sort of input/output error
`EOFEcception`	An attempt to read beyond the end of the data in a file
`FileNotFound`	A file not found

The Exception Hierarchy



Catching and Handling Exception

- An exception causes program execution interruption.
  - Console shows the series of calls starting from the main method that resulted in the exception
- Interruption can be avoided by:
  - Enclosing the execution in a try block
  - Processing the exception using a catch block

Try-Catch-Finally

- When an exception occurs, the series of catch statements is checked in order and the operations within the first matching block are executed; the rest are ignored.
- If all statements within the try block execute without error, the catch block is skipped.
- The code in the finally block is executed either after the try block is exited or after a catch clause is exited
- The finally block is optional

Example

```
Scanner scIn = new Scanner(System.in);
while (true) {
   try {
      System.out.print("enter a number, 0 for exist:  ");
      int n = scIn.nextInt();
      scIn.nextLine();
      System.out.println("Your number is " + n);
      if (n==0) {
         System.out.println("Quitting.");
         break;
      }
   }
   catch (InputMismatchException e) {
      System.out.println("You must enter a number.");
      scIn.nextLine();
   }  finally {
      System.out.prtinln("Try again.");
   }
}
```

How to Throw Exception

- Reporting an exception to the method that invoked the method in which an exception has occurred
- Can be chained and returned all the way back to the main method
  - Use `throws EXCETPTION_CLASS_NAME` at the method declaration to declare that the method may throw an exception
  - To throw, use
    `throw new EXCEPTION_CLASS_NAME(message)`
  - Handle recoverable exceptions on the spot

Big-O (from Chapter 2)

We want to be able to say something about efficiency of algorithms (programs) in terms of how their usage of resources (time and memory) increases as input size increases.
For example, there are many ways to reorder numbers appearing in an array. We can imagine that for each such method, the time it takes for it to reorder increases as the size of array increases.
Is there a simple way to say the way the time increases in this situation?

Big-O

Let f and g be functions from the set of nonnegative integers to the set of real numbers. We say that f is a "big-O" of g, write $f = O(g)$ or $f \in O(g)$, if there exists a positive constant c such that for all but finitely many n,

$$f(n) \leq c \cdot g(n)$$

This means that the growth rate of f can be bounded by some multiple of the growth rate of g.

Big-O Examples

In the following, f is a big-O of g

1. $f(n) = 100n$, $g(n) = 3n$.
2. $f(n) = 19n \log n$, $g(n) = 4n^2$.
3. $f(n) = n^2$, $g(n) = n!$.
4. $f(n) = n^{17}$, $g(n) = 2^n$.

Small-O

Let f and g be functions from the set of nonnegative integers to the set of real numbers. We say that f is a "small-O" of g, write $f = o(g)$ or $f \in o(g)$, if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In some sense, this means that the growth rate of f is strictly less than the growth rate of g.

Small-O Examples

In the following, f is a small-O of g

1. $f(n) = 100n$, $g(n) = 3n^2$.
2. $f(n) = 19n \log n$, $g(n) = 4n^2$.
3. $f(n) = n^2$, $g(n) = n!$.
4. $f(n) = n^{17}$, $g(n) = 2^n$.