Exceptions

CS 99 – Summer 2000
Michael Clarkson
Lecture 11

Administration

- Lab 10 due tomorrow
- No lab tomorrow
  - Work on final projects
- Remaining office hours
  - Rick: today 2-3
  - Michael: Thursday 10-noon, Monday 10-noon
- Final projects due Tuesday by noon via FTP
- You will receive your final grade from Cornell, not me

Agenda

- "Leftovers"
- Exceptions
- Course evaluations

"Leftovers"

- Topics we didn't get to cover:
  - Multi-dimensional arrays
  - Advanced String operations
  - Packages
  - Using the debugger
  - Files

When Things Go Wrong

- In the real world:
  - Input isn't nicely formatted
  - Hardware devices fail
  - Memory is limited
  - Code you rely on is buggy
- What to do?
  - Give up and abort the program
  - Return codes
  - Exceptions

Handling Errors

- Abort:
  - Calculator program, given division by 0
  - Poor choice, should at least:
    - Return to safe state and enable user to execute other commands, or:
    - Allow user to save work and terminate program gracefully
- Data validation:
  - Most any lab since we learned loops
- What if you can't reinput a value, though?
  - Example: user calls a method with illegal parameters

Return Codes

- *Return code*: value returned from a method indicating success or failure
- Usually a boolean or integer
  - If integer, constants often defined
- Example: Time class
  - Needed to setTime, but parameters could be invalid
  - Our solution: use default values
  - Another solution: use return codes

Return Codes [2]

```
boolean setTime(int h, int m, int s) {
  boolean success;
  if (isValidHour(h) && isValidMin(m) && isValidSec(s)) {
      hour = h;
      min = m;
      sec = s;
      success = true;
  } else {
      success = false;
  }
  return success;
}
```


Return Codes [3]

```
// input h, m, s from user
boolean ret = t.setTime(h, m, s);
if (ret) {
  // continue with program
} else {
  // reinput h, m, s and try again
}
```


Return Codes [4]

- Problems:
  - What if we also want to return something in addition to the return code?
  - Users of the method can ignore return code.
  - Can't use for constructors
  - Doesn't fit OOP model to return numbers/booleans when we're really trying to indicate an error

Exceptions

- Used to handle exceptional situations in code
- Instead of returning normally, a method can *throw* an object containing information about the situation (error)
- That object is an *exception*
- Method exits immediately after throwing an exception
  - Does *not* return normally (return any value)
  - Does *not* return to where method was called

Example Exception

```
class DivBy0 {
  public static void main(String[] args) {
      int num = 10, den = 0;
      System.out.println(num / den);
      System.out.println("Will never get here");
  }
}
```

```
Exception in thread "main"
  java.lang.ArithmeticException: / by zero
  at Zero.main(Zero.java:4)
```

Exception Output

- Type of exception
  - `java.lang.ArithmeticException`
- Descriptive message
  - `/ by 0`
- Stack trace
  - List of methods that had been called to get to that point in the program where the exception occured
  - `Zero.main(Zero.java:4)`

Handling Exceptions

- Users would rather see a nice error message, not exception output
- Can prevent output of exception by using a try-catch block:

```
try {
  // code that could throw an exception
} catch (ExceptionType objectName) {
  // code to execute if exception occurs
}
```


Exception Example, revisited

```
class DivBy0 {
  public static void main(String[] args) {
      int num = 10, den = 0;
      try {
            System.out.println(num / den);
      } catch (ArithmeticException e) {
            System.out.println("You can't divide by"
                               + "zero!");
      }
      System.out.println("Will always get here");
  }
}
```


There's always a `catch`

- Every `try` block must have at least one `catch`
- Can be multiple `catch` clauses:

```
try {
    ...
} catch (...) {
    ...
} catch (...) {
    ...
} ...
```

- Each clause must catch a different type of exception
- The `try` block must be able to throw each type of exception

`finally`...

- After all the `catch` clauses can be a single `finally` clause:

```
try {
    ...
} catch (...) {
    ...
} finally {
    ...
}
```

- `finally` code is executed after everything else

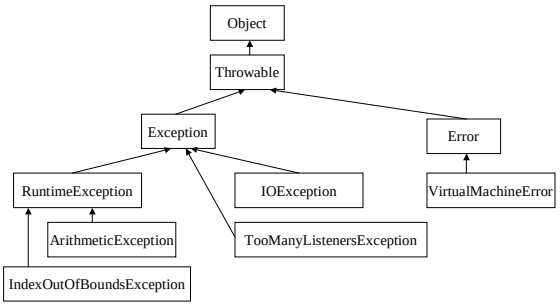
Flow of Control

- Exception doesn't occur:
  - Entire try clause is executed
  - No catch clause is executed
  - Finally clause is executed
  - After try block, execution proceeds with first statement after catch/finally clauses
- Exception occurs:
  - As soon as it occurs, control jumps to the single appropriate catch clause (*exception handler*)
  - After clause is executed, execution proceeds with finally, then the first statement after catch/finally clauses

Causes of Exceptions

1. Calling a method that can throw an exception
2. Detecting an error and throwing an exception yourself
3. Making a programming error, such as `a[-1]=0`
4. An internal error occurs in Java

Partial Exception Hierarchy



Checked vs. Unchecked

- A *checked* exception is checked by the compiler to make sure you catch it
  - All subclasses of `Exception` other than `RuntimeException`
- An *unchecked* exception is not checked by the compiler – you don't have to catch it
  - All subclasses of `Error` and `RuntimeException`

Advertising Exceptions

- Every method must declare the checked exceptions it can throw:

```
void myMethod() throws IOException, ... {
    ...
}
```

- Doesn't have to declare:
  - Unchecked exceptions
  - Exceptions it catches itself
  - `main`: any exceptions

Exception Propagation

- Exceptions *propagate* up the call stack until they are caught
  - So if one method doesn't catch an exception, Java checks the method that called it for a handler, all the way up to main
  - If no handler can be found, the default error message is printed and the program aborts

Propagation Example

- Exception generated by `ExceptionProp.level3()`
  - No handler in level3
  - No handler in level2
  - Handler in level1
- What is the output?

Propagation Example [2]

```
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.
Arithmetic exception occurred.
Level 1 ending.
Program ending.
```


Throwing Exceptions

- You can throw exceptions yourself using the `throw` statement:

```
throw exceptionReference;
```

- You can get a reference to an exception by creating one:

```
ArithmeticException e
   = new ArithmeticException("message");
throw e;
```


`setTime`, revisited

- Instead of default values or return code, throw an exception
- `IllegalArgumentException` is a `RuntimeException` used to indicate illegal values were passed to a method
- Since unchecked, responsibility is on programmer to make sure illegal values not passed to constructor or `setTime`

`setTime`, revisited [2]

```
public Time(int h, int m, int s) {
    setTime(h, m, s);
}
public void setTime(int h, int m, int s) {
    if (isValidHour(h) && isValidMin(m) && isValidSec(s)) {
        hour = h;
        min = m;
        sec = s;
    } else {
        throw new IllegalArgumentException("Hour, "
             + " minute, or second was illegal");
    }
}
```


Creating Exceptions

- Can create your own types of exceptions by declaring classes that extend `Exception`
- Extremely useful for handling errors in large programs
- Add {fields, methods} to exception to {store data, take action} on errors

Course Evaluations