CSC 143 Java

Errors and Exceptions

Reading: Ch. 15

What Can Go Wrong With Programs?

- Programs can have bugs and try to do things they shouldn't. E.g., try to send a message to null.
- Users can ask for things that they shouldn't (we can't control the user). E.g., try to withdraw too much money from a bank account.
- The environment may not be able to provide some resource that is needed.
  - Program runs out of memory or disk space
  - Expected file is not found
  - Extreme network examples: Thousands to millions of tiny sensors (one or more sensors break down), Interplanetary Internet (a server is down)

Coping Strategies

- Check all user input! (Not doing this has led to many insecurities.) But what should the program do if it's wrong?
- Be able to test whether resources were unavailable. But what should the program do if they weren't?
- Other strategies?

Reporting Errors with Status Codes

- If a method cannot complete properly because of some problem, how can it report it to the rest of the program?
- One approach: return a status code (error code)
  - Boolean status flags are very common
    - A boolean flag: true means OK, false means failure
  - Integers or other types could be used
    - An integer flag: 0 means OK, 1 means error of kind #1, etc.
    - For object return types: null could mean error, non-null could mean success
- What's bad about using this idea of returning a status code?

Status Codes in BankAccount

- From the original design of the bank account operations:
  ```java
  public boolean deposit(double amount) { return this.updateBalance(amount); }
  public boolean withdraw(double amount) { return this.updateBalance(-amount); }
  private boolean updateBalance(double amount) {
      if (this.balance + amount < 0) {
        System.out.println("Sorry, you don't have that much money to withdraw.");
        return false;
      } else {
        this.balance = this.balance + amount;
        return true;
      }
  }
  ```
- What do you think?

Status Codes: Pro and Con

- Easy to program, in the method that detects the error
  ```java
  MyObject methodThatMightFail(...) {
      if (weirdErrorCondition()) { return null; }
      else {
        //continue and create an object to return
      }
  }
  ```
- Can be bothersome for callers (why?)
- Can be unreliable (why?)
An Alternative: Throwing Exceptions

• Java (and C++, and many modern languages) include exceptions as a more sophisticated way to report and handle errors
• If something bad happens, program can throw an exception
  • A throw statement terminates the throwing method
  • throw sends back a value, the exception itself.
• So far it sounds a lot like the return statement
  • A return statement terminates the method
  • return can send a value back to the caller

Revised BankAccount Methods

public void deposit(double amount) { this.updateBalance(amount); }
public void withdraw(double amount) { this.updateBalance(-amount); }
private void updateBalance(double amount) {
  if (this.balance + amount < 0) {
    throw new IllegalArgumentException("insufficient funds");
  } else {
    this.balance = this.balance + amount;
  }
}

• Methods now have void return type, not boolean
• Error message and “return false” replaced with throw of new exception object
• Callers can chose to ignore the exception, if they don’t know how to cope with it
  • It will be passed on to the caller’s caller, and so on, to some caller that can cope

Return vs Throw

• A return takes the execution right back to where the method was called
  • Sometimes referred to as the “call site”
• A throw takes the execution to code (the handler) designated specifically to deal with the exception
  • The handler is said to catch the exception
• The handler might not be at or near the call site
• The calling (client) module might not even have a handler
• If a handler doesn’t exist somewhere, the program aborts

Throw Statement Syntax

• To throw an exception object, use a throw statement
  • Syntax pattern:
    throw <expression> ;
• The expression must be an object of type throwable
  • There are many such classes already defined
  • BankAccount example used IllegalArgumentException
  • The expression can’t be omitted
  • But it doesn’t just return to the caller, but ends execution of the caller, and its caller, and so on, until a handler is found (explained later), or the whole program is terminated
    • It’s bad practice for a complete program to die with an unhandled exception

Exception Objects In Java

• Exceptions are regular objects in Java
• Exception are subclasses of the predefined Throwable class
• Some predefined Java exception classes:
  • RuntimeException (a very generic kind of exception)
  • NullPointerException
  • IndexOutOfBoundsException
  • ArithmeticException (e.g. for divide by zero)
  • IllegalArgumentException (for any other kind of bad argument)
• Most exceptions have constructors that take a String argument

Throwable/Exception Hierarchy

- Throwable
  - Error
    - RuntimeException
  - Exception
    - ArithmeticException
    - NullPointerException
    - IllegalArgumentException
What about Handlers?

• As we said, return and throw have some similarities
• When a method ends as a result of a throw...
  • If the caller has a handler, that’s where execution continues
  • If the caller doesn’t have a handler, then its caller is checked to see if there is a handler.
• This checking of callers proceeds up the line, until a handler is found; if there isn’t one anywhere, the program aborts.
• That’s the big picture. A few details later.

Specifying an Exception Handler

• If a caller knows how to cope with an exception, then it can specify an appropriate handler using a try-catch block

```java
try {
    mySavingsAccount.withdraw(100.00);
    myCheckingAccount.deposit(100.00);
} catch (IllegalArgumentException exn) {
    System.out.println("Transaction failed: " + exn.getMessage());
}
```

• The catch part of the block constitutes the handler.
• If an exception is thrown anywhere inside the body of the try block, that is an instance of IllegalArgumentException or a subclass, then the exception is caught and the catch block is run

Try-Catch Blocks: Syntax

• Syntax:

```java
try {
    <body, a sequence of statements>
} catch (<exception type1> <name1>) {
    <handler1, a sequence of statements>
} catch (<exception type2> <name2>) {
    <handler2, a sequence of statements>
} ...
```

• Can have one or more catch clauses for a single try block

Try-Catch Blocks: Semantics

• First evaluate <body>
• If no exception thrown during evaluation of body, or all exceptions that are thrown are already handled somewhere inside body, then we’re done with the try-catch block: skip the catch blocks
• Otherwise, if an exception is thrown and not handled, then check each catch block in turn
  • See if the exception is an instance of <exception type1>
  • If so, then the exception is caught: Bind <name1> to the exception; execute <handler1>; skip remaining catch blocks and go to the code after the try-catch block.
  • If not, then continue checking with the next catch block (if any)
• If no catch block handles the exception, then continue searching for a handler, e.g. by exiting the containing method and searching the caller for a try-catch block surrounding the call

Example

• Implement a robust transferTo method on BankAccount, coping properly with errors that might arise

```java
public class BankAccount {
    ...
    public void transferTo(BankAccount otherAccount, double amount) {
```