Chapter 15 – Exception Handling

15.1 Throwing Exceptions

- What is a good program? A program that is reliable.
- Not just giving correct answer on correct input
- Should protect against possible errors (invalid password, not a picture file, etc)

Good programs

- Need a robust program, reliable to the user
- It should not ("cannot") crash easily

How do we do this?

- Option 1 – return a special value to determine if the method succeeded or failed
  - Ex. Make withdraw return true or false
- What’s wrong with this?
  - Calling method may not check answer
  - Calling method may not know what to do

Catching Exceptions

- Exception – an error condition that can occur during the normal course of a program execution
- In Java, exceptions are objects themselves
- Exception handling is another form of control structure (like ifs and switch statements)
  - When an error is encountered, the normal flow of the program is stopped and the exception is handled

Exceptions

- We say an exception is thrown when it occurs
- When the exception-handling code is executed, the error is caught
- Examples:
  - Divide by zero
  - Access a null object
  - Array Index Out of Bounds
Exception Program Flow

- What happens when exceptions occur?
  - An Exception object is thrown

- What happens when an Exception is thrown?
  - normal execution stops and exception handling begins

- What does the Exception object know?
  - the name of the problem
  - the location where it occurred
  - and more...

Why have exception handling?

- consistency (everyone else does it)
  - Java API classes use exceptions.
  - Other programming languages do too!

- flexibility
  - Programmer can decide how to fix problems.

- simplicity
  - Easy to pinpoint problems.

What exceptions can occur?

```java
FoodIterator fi = afc.getAllFood();
while (!hasFood() || !fi.hasNext()) {
  Food f = fi.nextFood();
  if (f.getLocation().equals(myLoc))
    myFood = f;
}
```

FoodIterator

```java
public double divide( int data1, int data2 )
{
  return data1 / data2;
}
```

Arithmetic Exception: divide by zero

What exceptions can occur?

```java
nullPointerException
```

```
exceptions
```

if (user enters "abc123")

```java
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.throwFor(Scanner.java:819)
at java.util.Scanner.next(Scanner.java:1431)
at java.util.Scanner.nextInt(Scanner.java:2040)
at java.util.Scanner.nextInt(Scanner.java:2000)
at Test.main(Test.java:5)
```

How to handle Exceptions?

- Do nothing
  - program crashes if the exception occurs!

- Propagate (throws) it
  - tell the method’s caller about it and let them decide what to do

- Resolve (try-catch) it in our method
  - fix it or tell the user about it and let them decide what to do

Exceptions

- So far, we have let the system handle exceptions

```java
int score = in.nextInt();
```

If the user enters "abc123"
Says in English:
- System has caught an error described as a `InputMismatchException`
- Thrown because a String cannot be converted to an integer
- When system handles, we often get a program crash
- Instead of the system, we can handle to improve robustness

Throwing Exceptions
- If there is an error in the value of a parameter, we can throw exceptions to make the user accountable

1. Decide what type of exception to throw
2. Test for condition, and throw the exception if condition is violated

Example
- Throw an exception object to signal an exceptional condition

Example: What do we do if the amount to withdraw is greater than the balance?
- `IllegalArgumentException`: illegal parameter value

Solution #1

```java
class BankAccount {
    public void withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Amount exceeds balance");
            throw exception;
        }
        balance = balance - amount;
    }
}
```

Solution #2

```java
class BankAccount {
    public void withdraw(double amount) {
        if (amount > balance) {
            throw new IllegalArgumentException("Amount exceeds balance");
        }
        balance = balance - amount;
    }
}
```
15.2 Checked/Unchecked Exceptions

- Checked Exception – checked at compile time
  - Compiler ensures that you are handling a possible problem
  - Due to external circumstances that the programmer cannot prevent
  - Majority occur when dealing with input and output
  - For example, IOException

- Unchecked Exception – Runtime Exceptions
  - Extend the class RuntimeException Of Error
  - They are the programmer’s fault
  - Examples of runtime exceptions:
    - NumberFormatException
    - IllegalArgumentException
    - NullPointerException
  - Optional to deal with these
  - Example of error: OutOfMemoryError
  - Can’t do anything about these catastrophic problems, so don’t deal with it

Why the difference

- Unchecked Exceptions result from deficiencies in your code, so should check on own
  - Null object reference
  - Sending a negative value to Math.sqrt()

- Checked Exceptions are not the fault of the coder
  - Problems with the file format, user input, etc.

Categories aren’t perfect:
- Scanner.nextInt throws unchecked InputMismatchException
- Programmer cannot prevent users from entering incorrect input
- This choice makes the class easy to use for beginning programmers

Deal with checked exceptions principally when programming with files and streams
- For example, use a Scanner to read a file
  ```java
  String filename = ...;
  FileReader reader = new FileReader(filename);
  Scanner in = new Scanner(reader);
  ```
- But, FileReader constructor can throw a FileNotFoundException
How do we deal with a checked Exception?

1. Handle the exception
2. Tell compiler that you want method to be terminated when the exception occurs
   - Use throws specifier so method can throw a checked exception

```java
public void read(String filename) throws FileNotFoundException
{
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    ... 
}
```

This tells the compiler to “pass the buck” to the method that called this method.

- Can propagate multiple exceptions:
  ```java
  public void read(String filename) throws IOException, ClassNotFoundException
  {
      FileReader reader = new FileReader(filename);
      Scanner in = new Scanner(reader);
      ... 
  }
  ```

- Can also group using hierarchy
  - If method can throw an IOException and FileNotFoundException, only use IOException

Why propagate?

- Why not handle ourselves?
  - We may not know how to
  - Let use of my code decide

- Better to declare exception than to handle it incompetently

15.3 Catching Exceptions

- At some point, an exception should be dealt with
  - If not, program terminates with error message

- Professional code requires more sophistication – cannot just allow errors to kill program
  - What would happen if all of my.wisc.edu turned off if you entered wrong password?

Solution

- Install exception handlers in your code to deal with possible exceptions
- Handlers are try/catch statements

Try-Catch

- Put statement(s) that could cause an error in the try block
- Error handling code goes in catch block
  - Only is executed if there was an error
- Can have multiple catch blocks, one for each possible type of exception
try-catch Syntax

```java
try {
    <try block>
} catch ( <ExceptionClass> <name> ) {
    <catch block>
} catch ( <ExceptionClass> <name> ) {
    <catch block>
}
```

3 types

- 3 types of error can be thrown
  - FileNotFoundException is thrown by FileReader constructor → caught by IOException clause
  - NoSuchElementException is thrown by Scanner.next → not caught, thrown to caller
  - NumberFormatException is thrown by Integer.parseInt() → caught by second clause

Execution Flow

- If there are no errors, the catch block is skipped
- If an exception is thrown, the try block stops executing immediately, jumps to catch block

Exception objects

- Why do we create an instance of the exception?
  - We can get information on the specific exception cause
- 2 methods defined (from the Throwable superclass):
  - getMessage() (what happened?)
  - printStackTrace() (where did it happen?)
getMessage

- Returns the data that cause the error
- Example:

```java
...)
catch (NumberFormatException e) {
    System.out.println(e.getMessage());
}
```

printStackTrace

- Prints out a trace of methods that caused the error starting at the root of the error
- Where did the exception occur? What method called this code to execute..etc.
- What the System does when an exception is thrown
- Example:

```java
catch (NumberFormatException e) {
    e.printStackTrace();
}
```

Catch ALL exceptions (bad idea)

```java
public double average(String data)
{
    try {
        int sum = 0;
        for (int i=0; i < data.length(); i++)
            sum += Integer.parseInt(data.charAt(i));
        return sum/data.length();
    }
    catch (Exception e) // catch ALL exceptions
    {
        System.out.println(e.getMessage());
        return 0;
    }
}
```

15.4 finally clause

- What if there is some code we want to execute regardless of exception or not?
- `finally` block is used

Throwing Exceptions

- What if an exception is thrown and there is no catch block to match?
  - System handles it (terminates program and prints stack)

Throwable Exceptions

- IMPORTANT! Order of catch blocks matters

```java
public double average(String data)
{
    try {
        int sum = 0;
        for (int i=0; i < data.length(); i++)
            sum += Integer.parseInt(data.charAt(i));
        return sum/data.length();
    }
    catch (Exception e) // catch ALL exceptions
    {
        System.out.println(e.getMessage());
        return 0;
    }
    catch (NumberFormatException e) {
        System.out.println("" + e + "not valid input, Please use digits only");
    }
}
```

- You should go specific to generic
- NumberFormatException is a specific type of the class Exception (inheritance)
try{
    distance = Double.parseDouble(str);
    if (distance < 0){
        throw new Exception("Negative distance is not valid");
    }
    return distance;
} catch (NumberFormatException e){
    System.out.println("Failed to parse: 
" + e.getMessage());
} finally {
    System.out.println("Done");
}

Throwing Exceptions

- finally is executed NO MATTER WHAT
- Even if there is a break or return in try block
- Good for "cleanup" of a method

**Exception Inheritance Hierarchy**

```
Throwable
  Exception
    RuntimeException
      NegativeArrayIndexException
      ArithmeticException
      IndexOutOfBoundsException
    IOException
      IOException
      FileNotFoundException
      EOFException
```

**Try-Catch Control Flow**

- code before try
- try block
- no exceptions occur
- code after try

**Try-Catch Control Flow**

- code before try
- try block
- exception occurs
- catch block
- code after try
**try-catch** Control Flow

- Code before try
- Try block
- **no exceptions occurred**
- Finally block (if it exists)
- Code after try

**try-catch** Control Flow

- Code before try
- Try block
- Exception occurs
- Catch block
- Finally block (if it exists)
- Code after try

**try-catch** Control Flow

- Code before try
- Try block
- Exception occurs
- **true** matches first catch block?
- **false**
- 1st catch block
- **true** matches next catch block?
- **false**
- 2nd catch block
- Finally block (if it exists)
- Finally block (if it exists)
- Code after try
- Exception thrown to caller

**Propagating Exceptions**

- When a method may throw an exception, either directly or indirectly, we call the method an exception thrower.
- Every exception thrower must be one of two types:
  - catcher.
  - propagator.

**Propagating Exceptions**

- An exception catcher is an exception thrower that includes a matching catch block for the thrown exception.

- An exception propagator does not contain a matching catch block.

- A method may be a catcher of one exception and a propagator of another.
Propagating Exceptions

- Do not catch an exception that is thrown as a result of violating a condition set by the client programmer (a precondition).

- Instead, propagate the exception back to the client programmer’s code and let him or her handle it.