CSE 1223: Introduction to Computer Programming in Java
Chapter 7 – Exception Handling
Exception Handling

- Recall the try-catch framework from dealing with File I/O:

```java
try {
    // statements possibly raising exception
} catch (exception declaration) {
    // statements dealing with exception
}
```

- Are there other kinds of exceptions it would be useful to be able to handle this way?
Exception Handling

There are three classes of Exceptions that Java programs can throw

- **Errors**
  - Errors internal to the Java Virtual Machine – OutOfMemoryError as an example
  - Don’t worry about catching these – these are fatal errors that tell you something is wrong with your machine

- **Runtime Exceptions (Unchecked Exceptions)**
  - Runtime errors that programmers should be able to avoid
    - DivisionByZero, IndexOutOfBoundsException, etc.
  - Don’t use try/catch for these – fix your code!
Exception Handling

- Runtime Exceptions (Unchecked Exceptions)
  - Runtime errors that programmers should be able to avoid
  - DivisionByZero, IndexOutOfBoundsException, etc.
  - Typically you don’t use try/catch for these – fix your code instead!
  - However, sometimes checking for these kinds of exceptions is useful
    - You can use a try/catch block to handle these kinds of exceptions
    - But Java does not require you to handle these exceptions
Exception Handling

- Checked Exceptions
  - All of the other kinds of exceptions
  - IOException, ClassNotFoundException, etc.
  - These *must* all be handled in your code if you use objects/methods that can possibly *throw* them
    - Your code either needs to *throw* them as well (up to the calling method)
    - Or it needs to handle them in a *try/catch* block
Checked vs. Unchecked Exceptions

- Why are some exceptions unchecked?
  - Unchecked exceptions can be checked by regular Java code
    - Check the length of an array before you access it
    - Check to see if your denominator is zero before you divide by it
    - The programmer knows what can possibly happen at compile time and can check for it
  - Checked exceptions cover circumstances outside of the programmer’s control
    - No way to test for a file’s existence without trying to open it
Using try/catch

- Suppose we want to write a program that:
  - Reads integers one at a time from the keyboard
  - Until it reads a negative number
  - Then it displays the average
    - A familiar program for this class
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int value = promptForNumber(in);
    int count = 0;
    int total = 0;
    while (value >= 0) {
        count = count + 1;
        total = total + value;
        value = promptForNumber(in);
    }
    System.out.println("Average: "+(total/count*1.0));
}

public static int promptForNumber(Scanner in) {
    int value = in.nextInt();
    return value;
}
Using try/catch

- Suppose we want to write a program that:
  - Reads integers one a time from the keyboard
  - Until it reads a negative number
  - Then it displays the average
    - A familiar program for this class
  - Now with a twist!
    - If the user enters something that isn’t an integer, it should display an error message and ask for new input
    - How will we do that?
      - We can add a try/catch to promptForNumber
public static int promptForNumber(Scanner in) {
    int value=0;
    boolean done=false;
    while (!done) {
        try {
            System.out.print("Enter an integer: ");
            value = in.nextInt();
            done=true;
        }
        catch(InputMismatchException e) {
            System.out.println("ERROR!  Input not an integer!");
            in.nextLine(); // clear input to start of a line
        }
    }
    return value;
}
public static int promptForNumber(Scanner in) {
    int value=0;
    boolean done=false;
    while (!done) {
        try {
            System.out.print("Enter an integer: ");
            value = in.nextInt();
            done=true;
        }
        catch(InputMismatchException e) {
            System.out.println("ERROR!  Input not an integer!");
            in.nextLine(); // clear input to start of a line
        }
    }
    return value;
}
public static int promptForNumber(Scanner in) {
    int value=0;
    boolean done=false;
    while (!done) {
        try {
            System.out.print("Enter an integer: ");
            value = in.nextInt();
            done=true;
        } catch (InputMismatchException e) {
            System.out.println("ERROR! Input not an integer!");
            in.nextLine(); // clear input to start of a line
        }
    }
    return value;
}
Multiple Exceptions

What happens if a piece of code can generate multiple exceptions, and you want to handle them differently?

- What if we want a program that prompts the user for a filename and then:
  - Opens the file for reading
  - If the file does not exist, display an error message that the file does not exist and exit
  - If the file does exist, read one integer at a time from the file into an ArrayList and then display those numbers
  - If we ever see a line that is not an integer, display an error message and exit
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a filename: ");
    String fname = keyboard.nextLine();
    ArrayList<Integer> list = new ArrayList<Integer>();
    try {
        Scanner in = new Scanner(new File(fname));
        while (in.hasNext()) {
            int value = in.nextInt();
            list.add(value);
        }
        in.close();
    } catch (IOException e) {
        System.out.println("ERROR! File "+fname+" not found!");
        System.exit(0);
    } catch (InputMismatchException e) {
        System.out.println("ERROR! Non integer value in file "+fname);
        System.exit(0);
    }
    for (int i=0; i<list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```java
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a filename: ");
    String fname = keyboard.nextLine();
    ArrayList<Integer> list = new ArrayList<Integer>();
    try {
        Scanner in = new Scanner(new File(fname));
        while (in.hasNext()) {
            int value = in.nextInt();
            list.add(value);
        }
        in.close();
    } catch (IOException e) {
        System.out.println("ERROR! File "+fname+" not found!");
        System.exit(0);
    } catch (InputMismatchException e) {
        System.out.println("ERROR! Non integer value in file "+fname);
        System.exit(0);
    }
    for (int i=0; i<list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a filename: ");
    String fname = keyboard.nextLine();
    ArrayList<Integer> list = new ArrayList<Integer>();
    try {
        Scanner in = new Scanner(new File(fname));
        while (in.hasNext()) {
            int value = in.nextInt();
            list.add(value);
        }
        in.close();
    } catch (IOException e) {
        System.out.println("ERROR! File " + fname + " not found!");
        System.exit(0);
    } catch (InputMismatchException e) {
        System.out.println("ERROR! Non integer value in file " + fname);
        System.exit(0);
    }
    for (int i=0; i<list.size(); i++) {
        System.out.println(list.get(i));
    }
}
Throwing Exceptions

- Recall that we could choose to throw an exception instead of handling it
  - This allows us to pass the handling of the exception up to a calling method
    - All the way up to the main method, or it can pass responsibility on to the Java Virtual Machine to handle it
  - When we do this, the exception immediately ends our method and we return to our calling method
    - Our calling method then needs to handle the exception, or throw it up to another method
    - Why would we want to do this?
Let’s look again at our program that:
- Reads integers one at a time from the keyboard
- Until it reads a negative number
- Then it displays the average

If the user enters something that isn’t an integer, it should display an error message and ask for new input.

How will we do that?
- Instead of adding a try/catch to promptForNumber, make promptForNumber throw an Exception
- Add the try/catch to the main method
```java
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int value=0, count=0, total=0;
    while (value>=0) {
        try {
            value=promptForNumber(keyboard);
            if (value>=0) {
                count=count+1;
                total=total+value;
            }
        } catch (InputMismatchException e) {
            System.out.println("ERROR!  Input not an integer!");
            keyboard.nextLine(); // clear input to beginning of new line
        }
    }
    System.out.println("Average: "+(total/count*1.0));
}

public static int promptForNumber(Scanner in) throws InputMismatchException {
    System.out.print("Enter an integer: ");
    int value = in.nextInt();
    return value;
}
```
public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int value=0, count=0, total=0;
    while (value>=0) {
        try {
            value=promptForNumber(keyboard);
            if (value>=0) {
                count=count+1;
                total=total+value;
            }
        } catch(InputMismatchException e) {
            System.out.println("ERROR! Input not an integer!");
            keyboard.nextLine(); // clear input to beginning of new line
        }
    }
    System.out.println("Average: "+(total/count*1.0));
}

public static int promptForNumber(Scanner in) throws InputMismatchException {
    System.out.print("Enter an integer: ");
    int value = in.nextInt();
    return value;
}
Throwing Exceptions

- Why would we choose to do this?
  - It makes our code *more versatile*
  - Think about the first approach – handling the exception in the method
    - It gets handled in exactly one way – an error is put to the screen and the user must enter a new value
    - Suppose we want to use this code for a new problem where if the user enters a non-integer, the program gracefully exits with an error message instead of asking for a new value
  - Do we need to rewrite our method code for the first approach? What about for the second?
  - Delaying the handling of an exception as long as possible makes it easier to *reuse* previously written code to solve new problems
throwing Exceptions

- Because this approach is useful for code reuse, we sometimes want to write methods that generate their own exceptions
  - Rather than just throwing exceptions that have been already generated
- We can use the **throw** statement to do this
Suppose we want to write a method that takes two parameters – a Scanner and an integer n

- The method should read n integers off the Scanner, store them in an array, and return the array
This works, but what happens if there aren’t enough elements in the file to fill the array?

We might want to throw an exception, to indicate to the calling program that an error has occurred.
public static int[] readInts(Scanner in, int n) throws IOException {
    int[] arr = new int[n];
    int count = 0;
    while (count < n && in.hasNext()) {
        arr[count] = in.nextInt();
        count = count + 1;
    }
    if (count < n) {
        throw new IOException("Not enough elements in file");
    }
    return arr;
}

This will now throw an IOException to the calling method. Since IOException is a checked exception, the calling method must handle it explicitly.

Notice that we added a throws declaration to the method signature to tell the compiler to pass that IOException up to the calling method.
Your Turn

- Modify the method `readInts()` to handle the case where the file has too many elements in it rather than too few

```java
public static int[] readInts(Scanner in, int n) throws IOException {
    int[] arr = new int[n];
    int count=0;
    while (count<n && in.hasNext()) {
        arr[count]=in.nextInt();
        count=count+1;
    }
    if (count < n) {
        throw new IOException("Not enough elements in file");
    }
    return arr;
}
```
Your Turn Part Two

- Write a main method that:
  - Prompts the user for a file and a number
  - Uses readInts() to read that many numbers from the file into an array
  - Prints the elements of that array
  - Make sure that your main method handles the IOExceptions properly

```java
public static int[] readInts(Scanner in, int n) throws IOException
```
Your Turn Part Two

public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.print("Enter a filename: ");
    String fname = keyboard.nextLine();
    System.out.print("Enter number of digits to read: ");
    int count = keyboard.nextInt();
    try {
        Scanner inFile = new Scanner(new File(fname));
        int[] myArray = readInts(inFile, count);
        for (int i=0; i<myArray.length; i++) {
            System.out.println(myArray[i]);
        }
    } catch (IOException e) {
        System.out.println("ERROR! A file error has occurred");
    }
}

How could we modify this to handle a FileNotFoundException differently from the IOException thrown by the readInts() method?
Question: Why is InputMismatchException *unchecked* instead of *checked*?

- Remember – if it were *checked*, Java would *require us* to handle it somehow
  - We’ve been using `nextInt()` all semester without handling it!
- There is actually a way to rewrite this code *without* using exceptions
  - You need to use a method we haven’t used yet – `hasNextInt()`
public static int promptForNumber(Scanner in) {
    int value=0;
    boolean done=false;
    while (!done) {
        System.out.println("Enter an integer: ");
        if (in.hasNextInt()) {
            value=in.nextInt();
            in.nextLine(); // clear input to start of a line
            done=true;
        }
        else {
            System.out.println("ERROR!  Input not an integer!");
            in.nextLine(); // clear input to start of a line
        }
    }
    return value;
}
public static int promptForNumber(Scanner in) {
    int value=0;
    boolean done=false;
    while (!done) {
        System.out.println(“Enter an integer: “);
        if (in.hasNextInt()) {
            value=in.nextInt();
            in.nextLine(); // clear input to start of a line
            done=true;
        }
    }
    return value;
}
Unchecked Exceptions

So what does this mean for us?

- If we find that our code is throwing an exception and we have not already been required to handle it …

- … then there’s probably a way to deal with the error in our program without putting in exception handling

- Time to research the Java Standard Library documentation and look for a solution

- But if we are getting an “Unhandled exception” syntax error message, then we must handle it.

- There won’t be some other way to deal with it in our program