COP 3330 – Object Oriented Programming in Java
Catching Exceptions thrown by Java

Exceptions - Options

If there is a problem with a Java statement, then a Java program may generate an exception or an error. Although an error represents a situation that the program can not recover from, an exception can be handled.

Here are your three options when dealing with a thrown exception:

1) Not handle the exception
2) Handle the exception where it occurs
3) Handle the exception at another point in the program.

Option #1

If you choose not to handle an exception, your program will terminate abnormally (called a run-time error) and produce an error message telling you which statement caused the exception to be thrown. This message is very helpful for debugging purposes. It will print out a call stack trace that tells you exactly which line in which method caused the exception. This printout includes the entire set of method calls leading to the exception. (e.g. main called func1 which called parseInt, which threw an exception.)

Option #2

In order to handle an exception, we must use the try statement. Here is the syntax of the try statement:

```
try {
     <stmts>
}
catch (<Class> <var>) {
     <exp_stmts>
}
```

You may have more than one catch clause. (You can have 0 or more.) Also, you may have a finally clause, but this must go at the end:

```
try {
     <stmts>
}
catch (<Class> <var>) {
     <exp_stmts>
}
finally {
     <fin_stmts>
}
```

If the statements inside of the try block execute normally, then execution skips to the finally block after finishing the try block. (If no finally block exists, then execution skips to right after the last catch block.)

However, if one of the statements in the try block throws an exception, then the rest of the statements in the try block are NOT executed. Rather, the class of the exception is noted. If this matches any of the catch clauses (these are processed just like an if statement, where only the first catch that matches the exception gets executed), then execution is transferred to the appropriate catch block. After that code is executed, execution transfers to right after the last catch block.

The whole benefit of catching an exception is that you allow your program to continue even if a line of code has thrown an exception. Here's a short example of a code segment that avoids the overall program crashing when a user doesn't enter an integer when they are supposed to:

```
boolean done = false;
int x;
while (!done) {

    try {
        System.out.println("Enter an integer.");
        x = Integer.parseInt(stdin.readLine());
        done = true;
    }
    catch (NumberFormatException exception) {
        System.out.println("Sorry, try entering an int!");
    }
}
```

This piece of code will continue to prompt the user for an integer until they enter a valid one. Notice that without the try and catch in the while loop, this program would terminate after the first time data of the wrong format was entered.

Notice that a block of code may have the possibility of throwing more than one exception. This is why you may have mutliple catches after a try block.

Finally clause

Execution is ALWAYS transferred to the finally clause whether an exception was thrown or not. It is executed last.

An example of when one would like to use a finally clause is when regardless of whether an exception occurs or not, you definitively want to complete an action, such as closing a file. (We haven't talked about files yet, but that lecture is coming soon!)

Exception Propagation

If an exception is not caught immediately in the method that it is thrown, the method will propagate or throw the exception to its calling method. It is possible them for this calling method to handle the exception.

Consider the following situation:

1) Main calls method level1.
2) level1 calls method level2 inside of a try block.
3) level2 calls method level3.
4) A statement in level3 throws an exception that is not caught.
5) The exception is propagated back to level2.
6) Since level2 doesn't handle the exception, it's propagated to level1
7) level1 handles the exception by printing out some pertinent information about the exception.

In the example posted (CircleExcptEx.java), main has a try block in it, and inside of that try block there is a call to a method getInput(). getInput throws an Exception, which is then caught by the try/catch block in main. Here the exception just gets propagated from getInput to main. However, the chain can go further. We could easily make getInput call another method, and have this method throw the exception to getInput, and then have getInput throw the exception back to main.

One way to think about this is that you have a person at work run into a problem (say a computer problem). He doesn't want to deal with it, so he throws it to his supervisor. He doesn't want to deal with it either, so he throws it to his supervisor. Then, she decides to fix (catch) the problem at her level.

Let's look at the code example further. First, let's look at the getInput method, which doesn't handle the exception but just throws it to the method that called it:

```
public static int getInput() throws InputMismatchException {
    Scanner stdin = new Scanner(System.in);
    System.out.println("What is the radius of your pizza(int)?");
    return stdin.nextInt();
}
```

The nextInt() method could throw the InputMismatchException. We choose to simply have getInput() report that this is a possibility. From here, it's up to some method in the call chain that we're currently in to catch and handle the exception.

Now, let's look at the main method, which calls getInput() from inside of a try-catch clause and handles the InputMismatchException (on the next page):

```
public static void main(String[] args) {

    int radius=0;
    boolean done = false;

    while (!done) {

        try {
            radius = getInput();
            done = true;
        }
        catch (InputMismatchException e) {

            e.printStackTrace();
            System.out.println("Try entering an integer this time.");
        }
    }

    double area = Math.PI*radius*radius;
    System.out.print("The area of your pizza with radius "+radius);
    System.out.println(" is "+area);
}
```

We initialize variables before the while loop to safe values. Then the while loop sets up to run as long as the user hasn't entered an input of the right type. We use the boolean variable done to control when we exit the loop. We try to read the input from the user when we call getInput() in the try clause. If that method throws the InputMismatchException, we are able to catch it and ask the user to enter an integer this time. When the user does enter an integer, no Exception will be thrown and the try clause will execute the statement that changes done to true, and then the while loop will be exited.

As can be seen, in order to write more fault tolerant code, we definitely need to increase the complexity of our code (as well as the nesting of our blocks of code. The original version of this program has a print, a nextInt() method call, an assignment statement and a print, just four lines. This version is non-trivially longer. While this is important for real-life code, it's inelegant and takes away from teaching problem solving in the beginning, which is why Exceptions are taught later in the course.

Checked vs. Unchecked Exceptions

A compiler can determine whether or not there is a possibility of a checked exception occurring. An example of a checked exception is a FileNotFoundException. At compile time, we can look at code and determine whether or not this exception is a possibility. This is also true about an IOException with a BufferedReader, but not true about the Exceptions that the Scanner methods could through.

For all checked exceptions, Java requires you to either catch them, or to report them with a throws clause. The throw clause must be included at the beginning of any method that has the possibility of throwing a checked Exception. We did this with main when we used a BufferedReader object to speed up the run-time of the Kattis Problem CD.

You might have noticed that you never bother throwing ArithmeticException, for example. That is because ArithmeticException is an unchecked exception. It is one that the compiler can't determine whether or not it might occur.

An unchecked one is not required to be thrown or caught. It may still be in the best interest of the programmer to try to catch these and track them down, but the compiler does not require it.

Method printStackTrace()

This method is defined in the Throwable class, from which Exception inherits. Thus, anytime you have an Exception object, you can call this method on that object. It will print out the stack of calls in the chain that created originally threw the exception. These are the SAME EXACT messages that the Java compiler prints when a regular run-time error (which occurs by an exception being thrown) happens.