Java Exceptions

Contents

Error Handling

Runtime errors can be divided into low-level errors that involve violating constraints, such as:

- dereference of a null pointer
- out-of-bounds array access
- divide by zero
- attempt to open a non-existent file for reading
- bad cast (e.g., casting an `Object` that is actually a `Boolean` to `Integer`)

and higher-level, logical errors, such as violations of a method's preconditions:

- a call to an `Iterator`'s "next" method when there are no more items
- a call to a Stack's "pop" method for an empty stack
- a call to "`factorial`" with a negative number

Logical errors can lead to low-level errors if they are not detected. Often, it is better to detect them (to provide better feedback).

Errors can arise due to:

- User error (for example, providing a bad file name or a poorly formatted input file):

A good program should be written to anticipate these situations and should deal with them. For example, given a bad file name, an interactive program could print an error message and prompt for a new name.

- Programmer error (i.e., a buggy program): These errors should be detected as early as possible to provide good feedback. For some programs it may be desirable to do some recovery after detecting this kind of error, for example, writing out current data.

Note that recovery is often not possible at the point of the error (because the error may occur inside some utility method that doesn't know anything about the overall program or what error recovery should involve). Therefore, it is desirable to "pass the error up" to a level that can deal with it.

There are several possible ways to handle errors:

- Write an error message and quit. This doesn't provide any recovery.
- Return a special value to indicate that an error occurred. This is the usual approach for C functions (which often return 0 or -1 to signal an error). However:
  - It doesn't work if the method also returns a value on normal completion and all values are possible (i.e., there is no special value that can be used to signal an error).
  - It requires that the calling code check for an error. This can reduce the efficiency of the code and is often omitted by programmers out of laziness or carelessness.
  - It can sometimes make the code more clumsy. For example, if method `g` might return an error code, one would have to write something like:

    ```
    ret = g(x);
    if (ret == ERROR_CODE) { ... }
    else f(ret);
    ```

    instead of just:

    ```
    f(g(x));
    ```

- Use a reference parameter or a global variable to hold an error code. This solves the first problem of the previous approach, but not the second or third ones.

- Use exceptions. This is the method of choice for many modern programming languages.

Exceptions

Idea:

> When an error is detected, an exception is **thrown**. That is, the code that caused the error stops executing immediately and control is transferred to the **catch clause** for that type of exception of the first enclosing **try block** that has such a clause. The `try` block might be in the current method (the one that caused the error) or it might be in some method that called the current method (i.e., if the current method is not prepared to handle the exception, it is "passed up" the call chain). If no currently active method is prepared to catch the exception, the exception ends up passed up the call chain from `main` to the Java virtual machine. At that point, an error message is printed and the program stops.

Exceptions can be built-in (actually, defined in one of Java's standard libraries) or user-defined. Here are some examples of built-in exceptions with links to their documentation:

- `ArithmeticException` (e.g., divide by zero)
- `ClassCastException` (e.g., attempt to cast a `String` object to `Integer`)
- `IndexOutOfBoundsException`
- `NullPointerException`
- `NoSuchElementException` (e.g., there are no more elements in collection or enumeration)
- `FileNotFoundException` (e.g., attempt to open a non-existent file for reading)

How to Catch Exceptions

Exceptions are caught using **try blocks**:

```
try {
    // statements that might cause exceptions
    // possibly including method calls
} catch ( ExceptionType1 id1 ) {
    // statements to handle this type of exception
} catch ( ExceptionType2 id2 ) {
    // statements to handle this type of exception
    .
    .
    .
} finally {
```

```
        // statements to execute every time this try block executes
    }
```

Notes:

1. Each `catch` clause specifies the type of one exception and provides a name for it (similar to the way a method header specifies the type and name of a parameter). Java exceptions are objects, so the statements in a catch clause can refer to the thrown exception object using the specified name.

2. The **finally clause** is optional; a `finally` clause is usually included if it is necessary to do some clean-up (e.g., closing opened files).

3. In general, there can be one or more `catch` clauses. If there ***is*** a `finally` clause, there can be zero `catch` clauses.

Example

Here is a program that tries to open a file for reading. The name of the file is given by the first command-line argument .

```
public class Test {
    public static void main(String[] args) {
        Scanner fileIn;
        File inputFile;

        try {
            inputFile = new File(args[0]);
            fileIn =   new Scanner(inputFile);  // may throw FileNotFoundException
        } catch (FileNotFoundException ex) {
            System.out.println("file " + args[0] + " not found");
        }
    }
}
```

Notes:

1. The program really should make sure there is a command-line argument before attempting to use `args[0]`.
2. Also, it probably makes more sense to put the `try` block in a loop, so that if the file is not found the user can be asked to enter a new file name and a new attempt to open the file can be made.
3. As is, if the user runs the program with a bad file name `foo`, the message "`file foo not found`" will be printed and the program will halt.
4. If there were no `try` block and the program were run with a bad file name `foo`, a more complicated message, something like this:

```
java.io.FilenotFoundException: foo
    at java.io.FileInputStream ...
    at ...
    at Test.main ...
```

would be printed. (Actually, if there were no `try`/`catch` for the `FileNotFoundException`, the program wouldn't compile because it fails to list that exception as one that might be thrown. We'll come back to that issue later.)

Checked and unchecked exceptions

Every exception is either a **checked** exception or an **unchecked** exception. If a method includes code that could cause a ***checked*** exception to be thrown, then:

- the exception must be declared in the method header using a **throws clause**, or
- the code that might cause the exception to be thrown must be inside a `try` block with a `catch` clause for that exception.

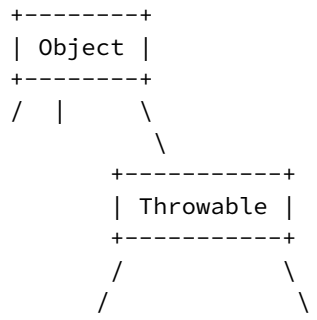
So, in general, you must always include some code that acknowledges the possibility of a checked exception being thrown. If you don't, you will get an error when you try to compile your code. (The compiler does not check to see if you have included code that acknowledges the possibility of an ***unchecked*** exception being thrown.)

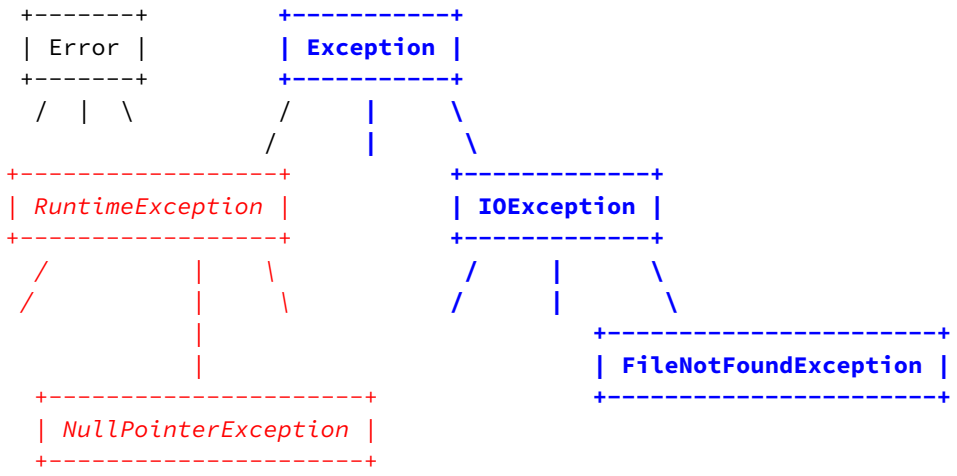
Exception hierarchy

How can you determine if an exception is checked or unchecked? The answer lies in the exception hierarchy:

- If an exception is a subclass of `RuntimeException`, then it is ***unchecked***.
- If an exception is a subclass of `Exception` but not a subclass of `RuntimeException`, then it is ***checked***.

A portion of the Java class hierarchy is shown below. Checked exceptions are in **bold**, unchecked exceptions are in *italics*.





Note that:

- Most of the built-in exceptions (e.g., `NullPointerException`, `IndexOutOfBoundsException`) are unchecked.
- I/O exceptions (e.g., `FileNotFoundException`) are checked.
- User-defined exceptions should usually be checked, so they should be subclasses of `Exception`.

Choices when calling a method that may throw an exception

If you are writing code that calls a method that might throw an exception, your code can do one of three things:

1. Catch and handle the exception.
2. Catch the exception, then re-throw it or throw another exception.
3. Ignore the exception (let it "pass up" the call chain).

As mentioned above, if your code might cause a ***checked*** exception to be thrown; i.e.,

- your code throws a checked exception, or
- your code ignores a checked exception that might be thrown by a called method

then your method must include a `throws` clause listing all such exceptions. For example:

```
public static void main(String[] args) throws FileNotFoundException, EOFException
{
    // an uncaught FileNotFoundException or EOFException may be thrown here
}
```

Only uncaught ***checked*** exceptions need to be listed in a method's `throws` clause.

Unchecked exceptions can be caught in a `try` block, but if not, they need not be listed in the method's `throws` clause.

TEST YOURSELF #1

Consider the following program (assume that comments are replaced with actual code that works as specified):

```
public class TestExceptions {

    static void e() {
      // might cause any of the following unchecked exceptions to be thrown:
      // Ex1, Ex2, Ex3, Ex4
    }

    static void d() {
      try {
          e();
      } catch (Ex1 ex) {
          System.out.println("d caught Ex1");
      }
    }

    static void c() {
      try {
          d();
      } catch (Ex2 ex) {
          System.out.println("c caught Ex2");
          // now cause exception Ex1 to be thrown
      }
    }

    static void b() {
      try {
          c();
      } catch (Ex1 ex) {
          System.out.println("b caught Ex1");
      } catch (Ex3 ex) {
          System.out.println("b caught Ex3");
      }
    }

    static void a() {
      try {
          b();
      } catch (Ex4 ex) {
          System.out.println("a caught Ex4");
          // now cause exception Ex1 to be thrown
      } catch (Ex1 ex) {
          System.out.println("a caught Ex1");
      }
    }
```

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

Assume that this program is run four times. The first time, method `e` throws exception `Ex1`, the second time, it throws exception `Ex2`, etc. For each of the four runs, say what is printed and whether any uncaught exception is thrown.

solution

Defining an exception

Java exceptions are ***objects***. We can define a new kind of exception by defining an instantiable class. This new class ***must*** be a subclass of `Throwable`; as discussed above, they are usually subclasses of `Exception` (so that they are checked). The simplest way to define a new exception is shown in this example:

```
public class EmptyStackException extends Exception { }
```

There is no need to provide any methods or fields; the class can have an empty body as shown above.

Many (if not most) of the exceptions defined in the Java API provide a constructor that takes a string as an argument (in addition to a default constructor). The purpose of the argument is to allow a detailed error message to be given when the exception object is created. You can access this message by calling the `getMessage()` method on the exception object. Providing this option in the exceptions you write is only slightly more complicated:

```
public class EmptyStackException extends Exception {
    public EmptyStackException() {
        super();
    }

    public EmptyStackException(String message) {
        super(message);
    }
}
```

Throwing an exception

At the point where an error is detected, an exception is thrown using a **throw** statement:

```
public class Stack {
        ...
    public Object pop() throws EmptyStackException {
        if (empty())
            throw new EmptyStackException();
        ...
    }
}
```

Note that because exceptions are objects, so you cannot simply throw "`EmptyStackException`" -- you must use "`new`" to create an exception object. Also, since the `pop` method might throw the (checked) exception `EmptyStackException`, that exception must be listed in `pop`'s `throws` clause.

TEST YOURSELF #2

Question 1: Assume that method `f` might throw ***checked*** exceptions `Ex1`, `Ex2`, or `Ex3`. Complete method `g`, outlined below, so that:

- If the call to `f` causes `Ex1` to be thrown, `g` will catch that exception and print "`Ex1 caught`".
- If the call to `f` causes `Ex2` to be thrown, `g` will catch that exception, print "`Ex2 caught`" and then will ***throw*** an `Ex1` exception.

```
static void g() throws ... {
    try {
        f();
    } catch ( ... ) {
        ...
    } ...
}
```

Question 2: Consider the following method.

```
static void f(int k, int[] A, String S) {
    int j = 1 / k;
    int len = A.length + 1;
    char c;

    try {
        c = S.charAt(0);
        if (k == 10) j = A[3];
    } catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("array error");
        throw new InternalError();
    } catch (ArithmeticException ex) {
        System.out.println("arithmetic error");
    } catch (NullPointerException ex) {
        System.out.println("null ptr");
```

```
        } finally {
            System.out.println("in finally clause");
        }
        System.out.println("after try block");
    }
```

Part A: Assume that variable `X` is an array of `int` that has been initialized to be of length 3. For each of the following calls to method `f`, say what (if anything) is printed by `f` and what, if any, uncaught exceptions are thrown by `f`.

```
A. f(0, X, "hi")
B. f(10, X, "")
C. f(10, X, "bye")
D. f(10, X, null)
```

Part B: Why doesn't `f` need to have a `throws` clause that lists the uncaught exceptions that it might throw?

solution

Summary

- Code that ***detects*** errors often does not know how to handle them. Therefore, we need a way to "pass errors up". The best approach is to use **exceptions**.
- Java provides both built-in and user-defined exceptions.
- Exceptions are caught using a **try block**:

```
try {
    // statements (including method calls) that might cause exceptions
} catch ( ExceptionType1 id1 ) {
    // code to handle this first type of exception
} catch ( ExceptionType2 id2 ) {
    // code to handle this second type of exception
    .
    .
    .
} finally {
    // code that will execute whenever this try block executes
}
```

- Exceptions are thrown using a **throw statement**.
- If an exception is thrown in code that is not inside a `try` block, or is in a `try` block with no `catch` clause for the thrown exception, the exception is "passed up" the call stack.
- Some exceptions are checked and some are unchecked. If a method might throw one or more checked exceptions, they must be listed in the method's **throws**

clause.

- Exceptions are objects; they are defined as ***classes*** (the class name is the name of the exception) that extend the `Exception` class.

Back

Next: Continue with Linked Lists