Prepared by Linan Qiu <lq2137@columbia.edu>

This note is prepared with extensive lifting from Java Lessons

Exception Handling

Exceptions are used to signify shit happening.

Try Catch Requirements

However, let's be sophisticated: just like in life, there are different kinds of shits in Java.

- Shit that you expect and should take care of. For example, you know that you're going to be turned upside down on a roller coaster. Hence, you wear seatbelts. In Java, we call these **checked exception** because you are expected to take care of them. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for `java.io.FileReader`. Normally, the user provides the name of an existing, readable file, so the construction of the `FileReader` object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name. You are expected to take care of these using `try/catch`, and if you don't your code will not compile.
- Shit that you could expect, but don't really need to take care of. For example, you can choose to buy a super awesome health insurance or you can choose a basic health insurance. The difference is in the big illnesses that have a slimmer chance of happening. In Java, we call these **errors**. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw `java.io.IOError`. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace (that bunch of lines that print tracing the line of code that causes the error) and exit.
- Shit that you can't possibly expect and you don't need to take care of. For example, you tripping while walking. You simply can't predict that. In Java, we call these **Runtime Exceptions**. These are exceptional conditions

that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for `FileReader`. If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Hierarchy of Exceptions

Now all exceptions extend `java.lang.Exception`. The three different types above form the 3 direct subclasses. For example, `FileNotFoundException`, which is a **checked exception** has the following superclasses:

```
java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.io.IOException
              extended by java.io.FileNotFoundException
```

whereas `ArrayIndexOutOfBoundsException` which is thrown when you try to access beyond the last index of an array is an **Runtime Exception** and extends `RuntimeException`

```
java.lang.Object
  extended by java.lang.Throwable
      extended by java.lang.Exception
          extended by java.lang.RuntimeException
              extended by java.lang.IndexOutOfBoundsException
                  extended by java.lang.ArrayIndexOutOfBoundsException
```

Handling Exceptions

There are two ways of handling exceptions: resolve them immediately or get someone else to resolve it.

Let's say we have the following methods:

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

```java
public static void methodA() {
  methodB();
}

public static void methodB() {
  throw new FileNotFoundException();
}
```

What's happening here is that the main method is calling `methodA`, which then calls `methodB`. Within method B, we instantiate a new exception of the type `FileNotFoundException` and throw it. Throwing has a special meaning: code stops executing and one of two things happen: the exception has to be handled immediately or be resolved by someone else.

Try Catching

We can choose to handle the exception immediately. The `try-catch` block does that:

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

public static void methodA() {
  try {
    methodB();
  } catch (FileNotFoundException e) {
    System.out.println("File not found");
    // do something else like ask for another file?
  }
}

public static void methodB() {
  throw new FileNotFoundException();
}
```

When the code inside the try portion of the `try-catch` block runs, Java will look out for exceptions thrown. If the exception thrown matches the type of the exception that you're catching, then the code will skip over everything after it and go straight to the portion in the catch block. Notice that now we are catching `FildNotFoundException e`, which means that only exceptions of the type `FileNotFoundException` will be caught. If `methodB` throw another kind of exception (say `ArrayIndexOutOfBoundsException`) we will not be able to catch it.

Now, when the code runs, we would see that `File not found` is printed. You can think of `methodB` as any procedure that may generate the `FileNotFoundexception` such as looking for a specific file on your computer.

Throwing Exceptions

Another way to handle this is to tell `methodA` to shrug when the exception hits and simply throw the responsibility to the method that called it. In this case, that's the main method. In other words, an exception that is thrown by `methodB` will in turn be thrown by `methodA`. We need to denote this by:

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

public static void methodA() throws FileNotFoundException {
  methodB();
}

public static void methodB() {
  throw new FileNotFoundException();
}
```

By adding the `throws FileNotFoundException` to `methodA`'s signature, we are telling other methods that use `methodA` to be prepared for `FileNotFoundExceptions`. In this case, if we leave the code as above, our code won't compile because we haven't dealt with the `FileNotFoundException` yet – we only procrastinated. Instead, the method calling `methodA` now needs to catch the `FileNotFoundException`.

```
public static void main(String[] args) {
  try {
    methodA();
  } catch (FileNotFoundExecption e) {
    system.out.println("File not found");
    // do something else
  }
}

public static void methodA() throws FileNotFoundException {
  methodB();
}
```

```java
public static void methodB() {
  throw new FileNotFoundException();
}
```

Remember that the rules of inheritance applies as well. Hence, I can change the catch clause to:

```java
public static void main(String[] args) {
  try {
    methodA();
  } catch (IOException e) {
    system.out.println("File not found");
    // do something else
  }
}

public static void methodA() throws FileNotFoundException {
  methodB();
}

public static void methodB() {
  throw new FileNotFoundException();
}
```

and the exception would be caught as well. In fact, the `FileNotFoundException` and all exceptions that extend `IOException` will be caught. This is actually not too good, because we always want to be specific with our exception (so that when users see exceptions, they know exactly what's causing the error.)

Hence, if we want to catch multiple exceptions, we usually leave the most specific ones (subclasses) at the top, then the more general ones (superclasses) at the top like this:

```java
try {
  methodA();
} catch (FileNotFoundException e) {
  // all FileNotFoundExceptions will be handled here
} catch (IOException e) {
  // all IOExceptions other than FileNotFoundException
  // will be handled here
}
```

This means that `FileNotFoundException` would be handled by the upper catch clause first.