Chapter 6: Exceptions.

Understanding Exceptions.

The role of exceptions. When you write a method, you can either deal with the exception or mate the calling code's problem. The role is to deal largely with the mistakes in the program. The key point to remember is that exceptions alter the program's flow.

Exception Types.
Java has a Throwable superclass for all objects that represent these events. Not all of them have the word *exception* in their classname, which can be confusing. Hierarchy is the following:

```
java.lang.Object → java.lang.Throwable →
                   java.lang.Error
                   java.lang.Exception → java.lang.RuntimeException
```

Error (subclass of Throwable) means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive "disappeared".

Runtime exception (subclass of Exception) include RuntimeException class and its subclasses, and tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. They are also known as *unchecked exceptions*. Note: compile errors are not runtime / unchecked exceptions.

A *checked exception* includes Exception class and all subclasses that do not extend RuntimeException. They tend to be more anticipated. For example, trying to read a file that doesn't exist. Why is it called checked? Java has the rule called the handle or declare rule, in which you need to handle or declare the checked exceptions in the method signature:

```
void fall() throws Exception { throw new Exception(); }
```

Keyword throws is used above, with the classname of the Exception / or any subclass that extend Exception. This method above might or might not throw an exception.

Throwing an exception.
There are two way of throwing an exception. As an example, something that's wrong in the code will throw an exception, as follow:

```
String[] animals = new String[0];
System.out.println(animals[0]);
```

This will throw an ArrayIndexOutOfBoundsException.
The second one, is to explicitly tell java to throw an exception, as follow:

```
throw new Exception();             throw new RuntimeException();
throw new Exception("Ow! I fell."); throw new RuntimeException("Ow! I fell.");
```

Using a Try Statement.

Java uses a try statement to separate the logic that might throw an exception, from the logic to handle it.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
}
```

If an exception is thrown in the code inside of the try statement, the code stops in that line, the catch clauses attempt to catch it. If no exception is thrown, the catch clause won't run. The *exception_type* is the type you are trying to catch, and the identifier refers to the caught exception object. Curly braces are required, even if there is only one statement in the block.

Adding a finally block: Java lets you add a *finally* clause that will execute whether or not an exception is thrown. Note: catch is not required if finally is present.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
} finally {
        // finally block
}
```

Catching various types of exceptions.

Custom exceptions are out of scope in OCA, but necessary to recognize them. Examples:

```
class AnimalsOutForAWalk extends RuntimeException { }
class ExhibitClosed extends RuntimeException { }
class ExhibitClosedForLunch extends ExhibitClosed { }
```

All of them are unchecked, as they directly or indirectly extend from RuntimeException. See more examples of how to catch multiple exceptions on CatchingMultipleExceptions.java.

Throwing a second exception.

Inside a catch or finally clause, any valid java code is acceptable, even another try and catch block. This is how you can throw a second exception.

```
try { // try block or protected code
} catch (exception_type identifier) {
        // exception handler
} finally {
        try { // try block or protected code
        } catch (exception_type identifier) {
              // exception handler
        }
}
```

See SecondExceptionSample1.java and SecondExceptionSample2.java for examples.

Recognize Common Exception Types.

Runtime Exceptions.

They extend from *RuntimeException*. They don't have to be handled or declared and can be thrown by programmer or the JVM.

- ***ArithmeticException***. Thrown by the JVM when code attempts to divide by zero.
- ***ArrayIndexOutOfBoundsException***. Thrown by the JVM when code uses an illegal index to access an array.
- ***ClassCastException***. Thrown by the JVM when an attempt is made to cast an object to a subclass of which it is not an instance. E.g.
  ```
  String type = moose;
  Object obj = type;
  Integer number = (Integer) obj; // Throws ClassCastException
  ```
- ***IllegalArgumentException***. Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument. E.g.
  ```
  public void setNumberEggs(int numberEggs) {
        if (numberEggs < 0)
           throw new IllegalArgumentException("# eggs must not be negative");
        this.numberEggs = numberEggs;
  }
  ```
- ***NullPointerException***. Thrown by the JVM when there is a null reference where an object is required.
- ***NumberFormatException***. Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format. It's a subclass of *IllegalArgumentException*.

Checked Exceptions.

They extend from *Exception*. They must be handled or declared and can be thrown by the programmer or the JVM.

- ***FileNotFoundException***. Thrown programmatically when code tried to reference a file that does not exist.
- ***IOException***. Thrown programmatically when there's a problem reading or writing a file.

Errors.

They extend the Error class. Thrown only by the JVM and should not be handled or declared. They are rare.

- ***ExceptionInInitializerError***. Thrown by the JVM when a static initializer throws an exception and doesn't handle it. Java fails to load the whole class.
  ```
  static {
  int[] countsOfMoose = new int[3];
  int num = countsOfMoose[-1]; } // Throws java.lang.ExceptionInInitializerError
  Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
  ```
- ***StackOverflowError***. Thrown by JVM when a method calls itself too many times (infinite recursion). The Stacks gets full / overflows.
- ***NoClassDefFoundError***. Thrown by JVM when a class that the code uses is available at compile time but not runtime.

Calling Methods That Throw Exceptions.

When calling a method that may throw a checked exception, the exception must be handled / declared in the method where the call is made. See Bunny.java and Bunny2.java classes for examples.

Subclasses.

- When a class overrides a method from a superclass or implements a method form an interface, it's not allowed to add new checked exceptions to the method signature. Example:

```
class CanNotHopException extends Exception { }
class Hopper {
     public void hop() { } }
class Bunny extends Hopper {
     public void hop() throws CanNotHopException { } // Does not compile
}
```

- A subclass is allowed, thought, to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them. Example:

```
class Hopper {
     public void hop() throws CanNotHopException { }  }
class Bunny extends Hopper {
     public void hop() { }
}
```

- A class is allowed to declare a subclass of an exception type. The idea is the same. The superclass or interface has already taken care of the broader type. Example:

```
class Hopper {
     public void hop() throws Exception { }  }
class Bunny extends Hopper {
     public void hop() throws CanNotHopException { }  }
```

All these rules apply to checked exceptions only. In the first example, declaring a runtime exception in the Bunny class would be perfectly legal.

Printing an Exception.

Three ways: you let Java print it, print just the message or print where the stack trace comes from. The first one prints the default: exception type and message. Second option prints message. The last one prints the stack trace, showing where the exception occurred in each method that it passed through. See PrintingAnException.java for examples.