ICS 111
Java Exceptions

- Motivation for exceptions
- Java exception hierarchy
- Throwing exceptions
- Catching exceptions
- Runtime systems
- Example

Motivation for Exceptions

- It is good to write code for the expected case:
  - non-empty strings
  - files that exist
  - (sometimes) non-negative numbers
- this code must be correct, and is usually well tested
- what happens with unexpected cases?
  - answer 1: crash the program
  - answer 2: crash the program, unless the programmer adds code to handle this **exception**
    - if the programmer wishes to do so
- Java follows answer 2

Throwing Exceptions

- It is good to write code for the expected case:
  - non-empty strings
  - files that exist
  - non-zero numbers
- this code must be correct, and is usually well tested
- what happens with unexpected cases?

```
throw new IllegalArgumentException("empty string");
throw new java.io.FileNotFoundException(fileName);
throw new ArithmeticException("division by zero");
```

- notice the difference:
  - `throw` generates the exception
  - `throws` tells the compiler that it’s OK that a method may generate the exception

Throwing Exceptions: Syntax

- The keyword `throw`, then `new`, then the constructor for the exception
  - remembering that in Java, everything (except for the 8 basic types) is an Object
  - so Exceptions are a particular kind of objects – they are Throwable objects
  - `new` is used to reserve a space in memory for these objects

```
throw new IllegalArgumentException("empty string");
throw new java.io.FileNotFoundException(fileName);
throw new ArithmeticException();
```

- The arguments to the constructor for the exception may take the empty argument list (), or can take a string describing the reason the exception was thrown

Throwing Exceptions: Semantics

- Throwing an exception ends execution of the current block
- If the exception is not caught in the method:
  - throwing the exception ends execution of the entire method
  - and of the method that called that method,
  - and so on, until
  - throwing the exception ends execution of the entire program
- On the other hand, if the exception is caught, execution continues at the `catch` statement
  - which may be in the same method as the expression is called
  - or in one of the methods that calls that
  - and so on, until
  - the exception may be caught in the `main` method
  - or may not be caught at all
- The `catch` statement will be explained next

Catching Exceptions

- If we want to catch an exception that may be thrown by a block of code, we surround that code with `try` … `catch`:

```
try {
  Scanner in = new Scanner(new File(fileName));
  String line = in.nextLine();
} catch (java.io.FileNotFoundException e) {
  System.out.println("file not found: " + e);
}
```

- the FileNotFoundException is caught no matter where in this code it is thrown – even if the constructor for Scanner calls another method that calls another method that calls another method that throws the exception
- The block following `catch` is the **exception handler**

Catching Multiple Exceptions: explicit catch statements

- We can catch more than one exception for a given block of code:

```
try {
  Scanner in = new Scanner(new File(fileName));
  String line = in.nextLine();
} catch (java.io.FileNotFoundException e) {
  System.out.println("file not found: " + e);
} catch (java.io.java.util.NoSuchElementException e) {
  e.printStackTrace();
}
```

- just like if … else if … else if … else, the block of statements for the first matching catch is executed, the remainder are not

Catching Multiple Exceptions: more generic exceptions

- Another way of catching multiple exceptions is to catch a more generic exception:

```
try {
  Scanner in = new Scanner(new File(fileName));
  String line = in.nextLine();
} catch (Exception e) {
  System.out.println("caught exception: " + e);
}
```

- This works because every java exception is also a java.lang.Exception
  - in the same way that every value in Java (that is not one of the basic types) is an Object
- This second way doesn't tell our code what exception we caught
  - sometimes the code doesn't discriminate based on the specific exception
  - the exception is printed for the user to see

Java Exception Hierarchy

- The Java exceptions are structured in a hierarchy:
  - `java.lang.Exception` belongs to the class of `Throwable` Objects
  - `java.lang.RuntimeException` and `java.io.IOException` belong to the class of `Exception` Objects
  - There are many exceptions that belong to the class of `RuntimeException` Objects:
    - `java.lang.ArithmeticException`
    - `java.lang.IndexOutOfBoundsException`
  - and many exceptions that belong to the class of `IOException` Objects:
    - `java.io.FileNotFoundException`
- When catching an exception X which belongs to the class of exceptions Y, and Y belongs to the class of exceptions Z, catching any one of X, Y, or Z will catch the exception
- That’s why catching `Exception` will catch any exception that is thrown
  - because `Exception` is the root of the Java Exception Hierarchy

Java Runtime Exceptions

- The general rule is that exceptions that might be thrown from a method must be declared in the method header (with `throws`)
  - these are **checked exceptions**
- The exception to this rule is that Java Exceptions that belong to the class of RuntimeException Objects do not have to be declared in the method header
  - these are **unchecked exceptions**
- In general, unchecked exceptions are those that are so common that catching them, or declaring them with `throws`, would make too many programs unnecessarily complicated

try with resources

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

- Sometimes you create a variable and you want to make sure it is closed properly
- try-with-resources will close the resource associated with the variable declared in the try:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

- the resource will be closed both if an exception is thrown, and if it is not
- try-with-resources is especially useful for closing output files!
  - example in book, special topic 7.5

Runtime System

- What happens to an uncaught exception?

```
Exception in thread "main" java.lang.StackOverflowError
    at x.infiniteRecursion(x.java:6)
    at x.infiniteRecursion(x.java:6)
```

- *something* outside of main catches the exception and prints the call stack
- this *something* is the same code that called main in the first place
- this code is called the **runtime system**, and its jobs include:
  - doing all the necessary setup to start your program, including
    - setting up System.out and System.in
    - getting the command-line arguments
  - calling main
  - catching and handling any uncaught exceptions in main

How the Runtime System Fits In

- The runtime system is in some sense part of the compiler, but in another sense is a program separate from the compiler
  - the compiler runs at compile time,
  - the runtime system runs every time you run your program
- When you run your program from Eclipse, the runtime system is different from the one I use to run the same program from the command line
- But all programs that you run from Eclipse use the same runtime system

Exceptions: an Example

```
public static void main(String[] args) {
  // throws IndexOutOfBoundsException
  Scanner in = new Scanner(args[0]);
  // throws InputMismatchException, NoSuchElementException, IllegalStateException
  int value = in.nextInt();
  // throws ArithmeticException
  System.out.println("10 / " + value + " is " + 10 / value);
}
```

- Let’s catch each of these and print appropriate messages

Example: Just catch Exception

```
public static void main(String[] args) {
  try {
    Scanner in = new Scanner(args[0]);
    int value = in.nextInt();
    System.out.println("10 / " + value + " is " + 10 / value);
  } catch (Exception e) {
    System.out.println("something went wrong!");
  }
}
```

- This is not helpful
  - Even letting the runtime system catch the exception is more informative
  - you can at least print the exception:

    ```
    System.out.println("something went wrong! exception " + e);
    ```

Example:
catch exceptions you care about

```
public static void main(String[] args) {
  try {
    Scanner in = new Scanner(args[0]);  // throws IndexOutOfBoundsException
    int value = in.nextInt(); // throws InputMismatchException
    System.out.println("10 / " + value + " is " + 10 / value); // throws ArithmeticException
  } catch (IndexOutOfBoundsException e) {
    System.out.println("Error: must have at least one argument");
  } catch (InputMismatchException e) {
    System.out.println("Error: argument must be an integer");
  } catch (ArithmeticException e) {
    System.out.println("Error: argument cannot be zero");
  } // the runtime system catches any remaining exceptions
}
```

- We could also do this with if statements:

Example:
avoid exceptions you care about

```
public static void main(String[] args) {
  if (args.length < 1) {
    System.out.println("Error: must have at least one argument");
    return;
  }
  Scanner in = new Scanner(args[0]);
  if (! in.hasNextInt()) {
    System.out.println("Error: argument must be an integer");
    return;
  }
  int value = in.nextInt();
  if (value == 0) {
    System.out.println("Error: argument cannot be zero");
    return;
  }
  System.out.println("10 / " + value + " is " + 10 / value);
}
```

So we see that there are multiple ways to solve this particular problem...

Philosophy:
when to throw an exception

- When:
  - you know your assumptions may not hold, and
    - i.e. there are inputs that you cannot handle
  - you don't know what to do about it
    - i.e. you can't handle the situation gracefully

  then it's a good time to throw an exception
- especially if someone else lower in the call stack might know what to do
- make sure the exception name is meaningful

Philosophy: when (not) to catch an exception

- When:
  - you know what to do with the exception
    - especially if your handling of the exception actually fixes the problem
  - the buck stops here: if you are in the main method, and your user(s) might not understand the exception

  then it is a good idea to catch and handle the exception
- it is not usually a good idea to catch exceptions that are caused by programming errors
  - except during debugging
  - because before you release your program to your users (or the TA), you should take care of these programming errors
- if in doubt, it is OK to handle an exception
  - this is “defensive programming”, akin to “defensive driving” – it is safer
- but as the book points out (programming tip 7.1), it is often better to let the caller handle the exception

Summary

- We finally see what to do exceptions:
  - throw to raise the exception
    - include new and any parameters to the constructor
  - catch to handle the exception
  - finally for code that must be executed whether or not an exception occurred
- Exceptions are a useful mechanism
  - some exceptions can be avoided with more conditionals – whether that’s worth it, is up to the programmer