Week 4 - Wednesday

COMP 2000

Last time

- What did we talk about last time?
- Exceptions
- Catching exceptions
- **`finally`** blocks
- The **`throws`** keyword

Questions?

Project 1

An application of exception handling

- Sometimes you need to convert a **`String`** to an **`int`** (or **`double`**)
- But you don't always know that the **`String`** is a properly formatted representation of an **`int`**

```
String number = "eggplant"; // Not a number!
int value = Integer.parseInt(number); // Fails!
```

- In these situations, it can be useful to catch a **`NumberFormatException`** and ask for another **`String`**

Application continued

```
int value = 0;
boolean success = false;
while(!success) {
   try {
      System.out.print("Enter a number: ");
      String number = in.next();
      value = Integer.parseInt(number);
      success = true;
   }
   catch(NumberFormatException e) {} // Don't need to do anything
}
```

- How do we know **`parseInt()`** can throw a **`NumberFormatException`**?
  - Read the Java API!

System.exit()

- **System.exit()** is a method that will shut down the entire JVM

```
if(trouble) {
  System.exit(-1); // Exits with error code -1
  System.out.println("You will never reach this line.");
}
```

- It's roughly equivalent to the **exit()** function in C
- You should **never** use **System.exit()**
- Exception handling is a **much better** way to end a program
- **System.exit()** doesn't give a chance for other threads to clean themselves up

Custom Exceptions

Creating your own exceptions

- If you're creating a framework of code, you might want to create your own exceptions
- For the Uno game, I created:
  - **`IllegalCardException`**
  - **`IllegalDrawException`**
  - **`EmptyDeckException`**
- You shouldn't create exceptions often, but they're useful if you want to name a particular program error

Creating an exception class

- Exceptions are classes like any other in Java
- They can have members, methods, and constructors
- All you need to do is make a class that extends **`Exception`**, the base class for all exceptions

```
public class SimpleException extends Exception {
}
```

- That's it.
- Although it makes them long, it's good style to put the word **`Exception`** at the end of any exception class name

Adding special information

- In some cases, you might want a constructor that lets you explain why the exception was created

```
public class TooManyEyeballsException extends Exception {
  private final int eyeballs;
  public TooManyEyeballsException(int eyeballs) {
     this.eyeballs = eyeballs;
  }
  public int getEyeballs() {
     return eyeballs;
  }
}
```

Getting information from exceptions

- When you catch an exception, you get a reference to the exception object itself
- It's customary (but not required) to call this reference **e**

```
try {
  if(eyeballs > 2)
     throw new TooManyEyeballsException(eyeballs);
}
catch(TooManyEyeballsException e) {
  System.out.println("Ugh! " + e.getEyeballs() +
     " is too many eyeballs!");
}
```

Messages in exceptions

- All **`Exception`** objects have a **`String`** message inside of them
- If you want your custom exception to have a message, you have to call the **`Exception`** constructor that takes a **`String`**

```
public class DoomException extends Exception {
  public DoomException(String prophecy) {
     super(prophecy); // Uses prophecy for message
  }
}
```

Throwing Exceptions

`throw` keyword

- The **`throw`** keyword is used to start the exception handling process
- You simply type **`throw`** and then the exception object that you want to throw
- Most of the time, you'll create a new exception object on the spot
  - Why would you have one lying around?

```
throw new CardiacArrestException();
```

- Don't confuse it with the **`throws`** keyword!

Throwing exceptions

- Code you write will seldom throw exceptions explicitly
- Remember than an exception is thrown when something has gone wrong
- Most of the time, your code will catch exceptions and deal with them
- If you write a lot of library code, you *might* throw exceptions to signal problems
- If you throw a checked exception in your method, you have to mark it with a matching **`throws`** descriptor

Exception throwing example

- Here's a method that finds the integer square root of an integer

```
public static int squareRoot(int value) {
  if(value < 0)
     throw new IllegalArgumentException("Negative value!");
  int root = 0;
  while(root*root <= value) {
     ++root;
  }
  return root - 1;
}
```

- If **`value`** is negative, an **`IllegalArgumentException`** will be thrown

Exceptions and Inheritance

Exceptions and inheritance

- One way that exceptions interact with inheritance is that all exceptions inherit from **`Exception`**
- Remember that you can use a child class **anywhere** you can use a parent class
- A catch block for a parent will catch a child exception
- If **`NuclearExplosionException`** is a child of **`ExplosionException`**, an **`ExplosionException`** catch block will catch **`NuclearExplosionException`**

Multiple catches with inheritance

- Because a parent **`catch`** will catch a child, you have to organize multiple **`catch`** blocks from most specific to most general:

```
try {
   dangerousMethod();
}
catch(FusionNuclearExplosionException e) {
   System.out.println("Fusion!");
}
catch(NuclearExplosionException e) {
   System.out.println("Nuclear!");
}
catch(ExplosionException e) {
   System.out.println("Explosion!");
}
catch(Exception e) { // Don't do this!
   System.out.println("Some arbitrary exception!");
}
```

Rules of thumb

- Always make exceptions as specific as possible so that you don't catch exceptions you didn't mean to
- Always put code in your exception handlers so that something happens
  - Otherwise, code will fail silently
- Never make a catch block for **`Exception`**
  - That will catch everything!

Rules for overriding methods

- In COMP 1600, we said that you can only override a parent method if you write a method that has the same name, takes the same parameters, and returns the same type
- That was a lie.
- You can change the parameters and the return type slightly, if you follow certain rules

Hoare's consequence rule

- Hoare's consequence rule says that a method can override a parent method as long as:
  1. Its parameters are broader (or the same)
  2. Its return value is narrower (or the same)
- In other words, it will take even more kinds of input but will give back fewer kinds of output

Hoare's consequence in action

```
public class SandwichShop {
  public Sandwich getSandwich(Dollars dollars) {
     return new Sandwich(dollars);
  }
}
```

Narrower

Broader

```
public class SubmarineSandwichShop extends SandwichShop {
  @Override
  public SubmarineSandwich getSandwich(Money money) {
     return new SubmarineSandwich(money);
  }
}
```

Application to exceptions

- Hoare's consequence rule applies to the exceptions that a method can throw as well
- If a method overrides a parent method, it can **only** throw exceptions that are the same or subtypes of the exceptions that the parent method throws
- Otherwise, someone might call the method on a child object and get exceptions they didn't expect

Exceptions example

```
public class Vehicle {
  public Trip ride(String destination) throws
     CrashException, NauseaException {
     return new Trip(destination);
  }
}
```

Narrower

```
public class Helicopter extends Vehicle {
  @Override
  public HelicopterTrip ride(String destination) throws
     HelicopterCrashException {
     return new HelicopterTrip(destination);
  }
}
```

Quiz

Upcoming

Next time…

- Review
- Look over chapters 1 – 6, 8 – 12, and 17

Reminders

- **Finish Project 1**
  - **Due Friday!**
- Exam 1 is Monday