Week 4 - Monday

COMP 2000

Last time

- What did we talk about last time?
- Dynamic binding and static methods
- Final methods and classes
- Abstract methods and classes
- The **`instanceof`** keyword and **`getClass()`** methods
- UML class diagrams

Questions?

Project 1

Exceptions

Errors

- Let's say that a method could cause an error
  - What should happen?
- In C, functions that cause errors return an error code, usually `-1`
- But that sucks!
  - Everything has to return an `int` that could be an error code
  - You have to check every single method return value to see if it's an error
- Wouldn't it be great if there was a general way to handle errors whenever they come up?

Exceptions

- Instead of checking every method, Java has a general way of handling errors (and other exceptional situations)
- The name for this system is **exception handling**
- When an error happens, code will **throw** an exception
  - Throwing an exception usually means something went wrong
- A special block of code **catches** the exception
- When you catch an exception, you can
  - Deal with the problem and move on
  - Throw the same (or a new) exception and make someone else deal with it

Catching an exception

- The **`risky()`** method has a chance of destroying the world
- If the world is destroyed, execution will jump into the **`catch`** block

```
try {
  System.out.println("About to do something risky!");
  risky();
  System.out.println("That was worth it!");
}
catch(WorldDestroyedException e) {
  System.out.println("Whoops. We destroyed the world.");
}
```

Another example

- Dividing an integer by zero causes an **ArithmeticException**

```
try {
  System.out.println("Let's divide by zero!");
  int value = 3 / 0;
  System.out.println("This line will never print!");
}
catch(ArithmeticException e) {
  System.out.println("Don't divide by zero!");
}
```

Yet another example

- It might be more sensible to deal with the problem

```
boolean success = false;
while(!success) {
   System.out.print("Enter a number: ");
   int divisor = in.nextInt();
   try {
      int answer = 100 / divisor;
      System.out.println("100 / " + divisor + " = " + answer);
      success = true;
   }
   catch(ArithmeticException e) {
      System.out.println("Don't divide by zero!");
   }
}
```

Multiple catch statements

- If a some code can cause many different exceptions, you can use multiple catches to handle them
- When a problem happens, execution will jump to the first catch that matches

```
try {
   useNumber(100 / divisor);
   getHoney();
   stayUpAllNight();
}
catch(ArithmeticException e) {
   System.out.println("We divided by zero!");
}
catch(BeeStingException e) {
   if(allergic)
       System.out.println("We're dying!");
   else
       System.out.println("Youch!");
}
catch(ExhaustedException e) {
   System.out.println("*YAWN*");
}
```

A `finally` block

- If an exception is thrown, the remaining code inside a **`try`** won't be executed
- If an exception isn't thrown, none of the **`catch`** blocks will be executed
- If you want code that is executed no matter what, it can be put in a **`finally`** block after all the **`catch`** blocks
- **`finally`** blocks are often used to do clean-up so we're sure it gets done
  - Things like closing files or network connections

finally example

- Statements in a **finally** happen no matter what
- Even if some uncaught exception leaves the method

```
try {
  acid.juggle();
  System.out.println("I'm an amazing juggler!");
}
catch(FaceMeltException e) {
  System.out.println("I melted my face!");
}
finally { // Happens no matter what
  room.cleanUp();
  lights.turnOff();
}
```

`finally` is out of control!

- The power of a **`finally`** block is surprising
- Even if you're about to return, code in the **`finally`** will be executed (and can override whatever you're doing)
- Only killing the JVM will stop a **`finally`**

```
try {
  if(random.nextInt() % 2 == 0)
     return "Even";
  else
     return (7 / 0) + " trouble!";
}
catch(ArithmeticException e) {
  return "Ruh-roh";
}
finally {
  return "I win!"; // "I win!" will always return
}
```

Catch or specify

- Exceptions in Java come in two categories
  - Checked
  - Unchecked
- You **must** deal with checked exceptions
- If a method could throw a checked exception, you have to run that method inside of a **`try`** block with a **`catch`** that matches the exception
- **Or** you can specify that your method also throws the exception
- Essentially, you have to deal with the problem or warn other people that you can cause the same problem

Checked exceptions

- Most exceptions that come up frequently are checked exceptions:
  - **`FileNotFoundException`**
  - **`IOException`**
- Most exceptions you will design and throw will be checked
- Checked exceptions indicate that a problem has happened, but it might be possible to recover from the problem
- For example, trying to open a file that doesn't exist could cause a **`FileNotFoundException`**
  - Recovering from this exception might involve asking the user to pick another file name
- Checked exceptions inherit from the **`Exception`** class

Unchecked exceptions

- Unchecked exceptions don't require a **`try`** block
  - If they did, **almost everything** would be in a **`try`** block
- They usually mean there's a bug in the code
- Common unchecked exceptions:
  - **`ArithmeticException`** (division by zero)
  - **`ArrayIndexOutOfBoundsException`**
  - **`StringIndexOutOfBoundsException`**
  - **`ClassCastException`**
  - **`NullPointerException`**
- You don't **have** to catch these, but you can
- Unchecked exceptions inherit either from the **`Error`** class or the **`RuntimeException`** class

The throws keyword

- If a method doesn't want to catch a (checked) exception, it can be marked as throwing that exception with the **throws** keyword

```
void pet(Goat goat) throws GoatBiteException {
  goat.touch(); // can throw GoatBiteException
}
```

- This **pet()** method doesn't handle a **GoatBiteException** and thus must use the **throws** keyword to warn other code that it could throw a **GoatBiteException**

Throwing more than one exception

- A method can have an unlimited number of exceptions listed after the **`throws`** keyword
  - Separate them with commas
- Perhaps many bad things can happen in the method

```
void haveAdventures() throws LostLegException,
  PetrificationException, AcidBurnException {
  becomePirate();       // Might lose a leg
  fightMedusa();        // Might turn to stone
  killXenomorph();      // Might be burned by acid
}
```

Non-local control

- What's really powerful about exceptions is that they are a form of **non-local control**
- Local control flow means changes to program execution that happen within a method
  - Making a choice with an **`if`**
  - Repeating with a loop
- A **`return`** statement moves control back to the method that called the current method
- Like a **`return`**, if an exception isn't caught, it will go back to the method that called the current method…
  - But if that method doesn't catch the exception, it will go back to the previous
  - And so on…

What happens if an exception is never caught?

- In many cases, we want to deal with the exception and keep going
- However, if no **`catch`** statement catches an exception, it keeps unwinding methods back to the previous method and the one before that…
- Ultimately, if the **`main()`** method doesn't catch the exception, it will kill the program (or just the current thread if there's more than one)
- The JVM will print out a message about the exception and a stack trace of all the methods involved, all the way down to the method that caused the exception
  - Eclipse shows this message in **red**

`NullPointerException`

- A **`NullPointerException`** is a very common unchecked exception
- It happens whenever you try to access a method or a member of a **`null`** reference
- It's fine if a reference is **`null`**, but if you use a dot (`.`) to try to access something *inside* the **`null`** reference, your program will likely crash
- It almost never makes sense to catch a **`NullPointerException`**
  - They just mean the program has a mistake

NullPointerException examples

- Usually, we get a **NullPointerException** when we try to call a method

```
String text = null;
int length = text.length();  // NullPointerException
```

- Sometimes people get confused when they make arrays
- When it's created, an array is full of **null** references

```
Wombat[] wombats = new Wombat[100]; // 100 nulls
// NullPointerException
System.out.println(wombat[0].toString());
```

IndexOutOfBoundsException

- When trying to access an invalid index in an array, you'll get an **ArrayIndexOutOfBoundsException**

```
int[] numbers = new int[50];
numbers[-2] = 5;  // ArrayIndexOutOfBoundsException
numbers[50] = 21; // Also illegal: indexes from 0 to 49
```

- Strings have a similar **StringIndexOutOfBoundsException** when you try to access indexes they don't have

```
String distance = "a mile long";
char c = distance.charAt(12);  // Out of bounds!
String smaller = distance.substring(-4,7);  // Negative?
```

Upcoming

Next time…

- Defining your own exceptions
- Throwing exceptions

Reminders

- **Michael Thornton talk:**
  - **How to get a Software Engineering Job**
  - Tuesday, February 4, 4-6 p.m.
  - The Point 113
- Keep reading Chapter 12
- Keep working on Project 1
  - Due Friday!