How Exceptions Work

- **Operative terms**
  - **Throwing** (or **raising**) an exception
    - Complain about an error
  - **Catching** an exception elsewhere in code
    - Checks for existence of an exception and "handling" it somehow
    - `try-catch` clause (more on this later)

- **In Java, Exceptions are** **Objects** **being thrown and caught**
  - These Objects carry diagnostic information about the error
  - (e.g., what caused it, was there a msg?)
How Exceptions Work

- Mechanism for exception handling:
  - An exception gets thrown "up" the call stack

- After exception is thrown in a method,
  - Java searches each caller for willingness to handle it
    - If the method "catches" and handles it:
      - Great, exception goes away!
    - If the method is not willing to handle it
      - It gets thrown up to its caller method
      - If no one handles it, the exception reaches the JVM, which terminates the program

"Call Stack"

C()  
B()  
A()  
main()  
Java Virtual Machine (JVM)
Exception Producing Code

- Study the following code (2 min)
  - Why does it crash 10% of the time? (3 min to come up with both ways)

```java
import java.util.Scanner;

public class Example {
    public static void main(String[] args) {
        // get user to input a number, convert it to int
        Scanner keyboard = new Scanner(System.in);
        int x = Integer.parseInt(keyboard.nextLine());
        System.out.println(Example.foo(x));
    }

    private static int foo(int y) {
        return bar(y-1);
    }

    private static int bar(int z) {
        return 4/z;
    }
}
```
Outline

- What Are Exceptions?
- How Exceptions Work
- Exception Handling
  - Receiving end: throws clause and try-catch block
  - Producing end: raising exceptions
- Exception Types: Unchecked vs. Checked
- Defining Custom Exceptions
- Conclusion
Exception Handling (Option 1)

- A method has two options when it receives an exception:
  - **Option 1:** *Be passive!* Make it someone else's problem

    - Syntax: `<method declaration> throws exceptionType0, exceptionType1, ...`

- Example:

  ```java
  /**
   * This method blah blah blah
   * @return some integer
   * @throws ArithmeticException if z is (fill in)
   */
  public static int foo(int z) throws ArithmeticException {
      // (code omitted)
      // (this code could produce an ArithmeticException)
  }
  ```

- Note the following:
  - A new `@throws` tag for Javadoc commenting

Whoever calls this method: You might have to deal with this exception.
Example: Throwing It up Further (Be Lazy)

- Play hot potato: `foo()` makes an announcement:
  - It itself could raise an `ArithmeticException`, or
  - It could call a method that generates an `ArithmeticException`

```java
public static void foo(int z) throws ArithmeticException {
    // (code omitted)
    // (this code could produce an ArithmeticException)
}
```
Example: Throwing It up Further (Be Lazy)

- Play hot potato: `foo()` makes an announcement:
  - It itself could raise an ArithmeticException, or
  - It could call a method that generates an ArithmeticException
- Any ArithmeticException that `foo` throws is `main`'s problem

```java
public static void main(String[] args) {
    int x = getNumber();
    foo(x); // This is a potentially "dangerous" statement,
    // but you'd never know it because it doesn't return anything
    // (remainder code)
}

public static void foo(int z) throws ArithmeticException {
    // (code omitted)
    // (this code could produce an ArithmeticException)
}
```
Example: Throwing It up Further (Be Lazy)

- Play hot potato: `foo()` makes an announcement:
  - It itself could raise an `ArithmeticException`, or
  - It could call a method that generates an `ArithmeticException`

- Any `ArithmeticException` that `foo` throws is `main`'s problem
  - Unless `main` throws it up to its caller...

```java
public static void main(String[] args) throws ArithmeticException {
    int x = getNumber();
    foo(x); // This is a potentially "dangerous" statement,
    // but you'd never know it because it doesn't return anything

    // (remainder code)
}
	public static void foo(int z) throws ArithmeticException {
	    // (code omitted)
	    // (this code could produce an ArithmeticException)
}
```
Summary: Being Passive

- Summary: Hot potato way
  - Uhh... someone will deal with the exception, just not me.

- Pros:
  - It's a simple way to make the compiler happy
    - If I'm just writing some test code that doesn't matter much...

- Cons:
  - What if no one along the way catches the exception?
    - It will reach the Java Virtual Machine...
A method has two options when it receives an exception:

- **Option 2: Be proactive:** Try to *recover* from the exception!
  - Need to have a plan to recover the program from crashing

**Syntax:**

```java
try {
    // code that could result in an exception
}
catch(ExceptionType varName) {
    // what to do if one type of exception was thrown in the "try" block?
}
```
Exception Handling (Option 2)

- A method has two options when it receives an exception:
  - **Option 2:** *Be proactive:* Try to *recover* from the exception!
    - Need to have a plan to recover the program from crashing

Syntax:
```java
try {
    //code that could result in several types of exceptions
} catch(ExceptionType varName) {
    //what to do if one type of exception was thrown in the "try" block?
}
catch(ExceptionType varName) {
    //what to do if a different type exception was thrown?
}
//maybe more
```
Example: Handling It Now (Be Proactive)

- **Exercise:** Change `main()` so that it recovers from any exceptions

  - (Naive) Recovery Plan:
    - Print out the error, then
    - Ask for the number again
    - Call `foo()` again

```java
public static void main(String[] args) {
    int x = getNumber();
    foo(x);  // potentially dangerous statement!
}

public static int foo(int x) throws ArithmeticException {
    // (this code could produce an ArithmeticException)
}
```
Example: Handling It Now (Be Proactive)

- Solution for naive (second-chance) recovery:

```java
public static void main(String[] args) {
    int x = getNumber();

    try {
        foo(x); // potentially dangerous statement!
    }
    catch(ArithmeticException ex) {
        System.out.println(ex.getMessage()); // what was the error?
        x = getNumber(); // new life! we get a second chance
        foo(x);
    }
}

public static int foo(int x) throws ArithmeticException {
    // (this code could produce an ArithmeticException)
}
```

(Can't we do better?)
Example: Handling It Now (Be Proactive)

- Solution for naive (second-chance) recovery:

```java
public static void main(String[] args) {
    int x = getNumber();

    try {
        foo(x);  // potentially dangerous statement!
    } catch(ArithmeticException ex) {
        System.out.println(ex.getMessage()); // what was the error?
        x = getNumber();  // new life! we get a second chance
        foo(x);  // ------ um... isn't this call potentially dangerous?
    }
}

public static int foo(int x) throws ArithmeticException {
    // (this code could produce an ArithmeticException)
}
```

(Can't we do better?)
A Better Recovery Plan

- Refactor the code below so that:
  - We ask for a new number *as long as* `foo()` throws an exception

```java
public static void main(String[] args) {
    int x = getNumber();

    try {
        foo(x); // potentially dangerous statement!
    }
    catch(ArithmeticException ex) {
        System.out.println(ex.getMessage()); // what was the error?
        x = getNumber(); // new life! we get a second chance
        foo(x); // <---- um... isn't this call potentially dangerous?
    }
}

public static int foo(int x) throws ArithmeticException {
    // (this code could produce an ArithmeticException)
}
A Better Recovery Plan

- Here's one way

```java
public static void main(String[] args) {
    int x = getNumber();

    while (true) {
        try {
            foo(x);
            break; // yay, no exceptions - break out of infinite loop
        }
        catch (ArithmeticException ex) {
            // caught an exception!
            // recover by getting a new number
            x = getNumber();
        }
    }
    // (remainder code)
}
```

```java
public static int foo(int x) throws ArithmeticException {
    // (this code could produce an ArithmeticException)
}
```