EXCEPTIONS

Prof. Chris Jermaine
cmj4@cs.rice.edu

Before We Begin

- Good programmers always ask, with every line of code they write
  - — "What could possibly go wrong here?"
- Example from the class project
  - — To imp our ML over docs, we'll be adding DoubleVectors together
  - — "vec1" might be the row of one matrix
  - — "vec2" might be the column of another
  - — What if you just wrote:

```
for (i = 0; i < vec1.getLength (); i++) {
  vec1.setItem (vec1.getIem (i) + vec2.getItem(i), i);
}
```

Before We Begin

- Example from the class project
  - — To imp our ML over docs, we'll be adding DoubleVectors together
  - — "vec1" might be the row of one matrix
  - — "vec2" might be the column of another
  - — What if you just wrote:

```
for (i = 0; i < vec1.getLength (); i++) {
  vec1.setItem (vec1.getIem (i) + vec2.getItem(i), i);
}
```

  - — Fine if you didn't screw up the extraction of the two vecs
  - — But what if got "vec2" from the wrong matrix?
  - — Might go thru fine, but your answer is wrong and you have no idea why
  - — If you'd actually checked the lengths, you'd catch this error!

Exceptions

- In FORTRAN, C...
  - — Your program would simply crash
- In more modern languages, we have **exceptions**
  - — At high level, “exception” is language-provided mechanism for handling errors

Exceptions

- In FORTRAN, C...
  - — Your program would simply crash
- In more modern languages, we have **exceptions**
  - — At high level, “exception” is language-provided mechanism for handling errors
- Can divide use cases 50/50
  - — Often, programmers use exceptions like a fancy “assert”
  - — Exit the program when something bad happens
  - — Exceptions give you a nice framework for controlling/documenting the abort

Exceptions

- In FORTRAN, C...
  - — Your program would simply crash
- In more modern languages, we have **exceptions**
  - — At high level, “exception” is language-provided mechanism for handling errors
- Can divide use cases 50/50
  - — Often, programmers use exceptions like a fancy “assert”
  - — Exit the program when something bad happens
  - — Exceptions give you a nice framework for controlling/documenting the abort
- Other common use: ability to recover from unexepected events
  - — File suddenly disappearing, network unavailable, no input for 5 minutes...
  - — Be a bit careful with this one (exceptions as control flow is generally bad)

Exceptions in Java

- Are two kinds
- A “checked exception”
  - — Is a message that is passed from a callee back up the call stack
  - — When a caller gets such a message, normal execution is stopped
  - — And special code is executed in the caller
  - — “Checked” exceptions are unique to Java
- An “unchecked exception”
  - — Is a message that is passed from a callee back up the call stack
  - — When a caller gets such a message, normal execution is stopped
  - — And special code is executed in the caller...

    -OR-
  - — ...the caller does not supply such code, and the program exits (crashes!)

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
   // some stuff in here
} catch (ExceptionOne e) {
   // some stuff in here
} catch (ExceptionTwo e) {
   // some stuff in here
} catch (ExceptionThree e) {
   // some stuff in here
} finally {
   // some stuff in here
}
```

- A few rules:
  - — Need at least one “catch” in a “try” statement
  - — Can have zero or one “finally” clauses in a “try” statement

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
   // some stuff in here
} catch (ExceptionOne e) {
   // some stuff in here
} catch (ExceptionTwo e) {
   // some stuff in here
} catch (ExceptionThree e) {
   // some stuff in here
} finally {
   // some stuff in here
}
```

- A few rules:
  - — "ExceptionOne", "ExceptionTwo", "ExceptionThree" name class types
  - — Must all be subclasses of "Throwable"

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
    // some stuff in here
} catch (ExceptionOne e) {
    // some stuff in here
} catch (ExceptionTwo e) {
    // some stuff in here
} catch (ExceptionThree e) {
    // some stuff in here
} finally {
    // some stuff in here
}
```

First, this is executed

- What does this do?

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
    // some stuff in here
} catch (ExceptionOne e) {
    // some stuff in here
} catch (ExceptionTwo e) {
    // some stuff in here
} catch (ExceptionThree e) {
    // some stuff in here
} finally {
    // some stuff in here
}
```

First, this is executed

If an exception is created somewhere below here in the call stack...

...and it makes it up this far...

...Java tries to match it it's type with one of these...

...if it can't, up to caller.

- What does this do?

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
   // some stuff in here
} catch (ExceptionOne e) {
   // some stuff in here
} catch (ExceptionTwo e) {
   // some stuff in here
} catch (ExceptionThree e) {
   // some stuff in here
} finally {
   // some stuff in here
}
```

If a match is found, then corresponding code is run...

The exception object is visible to that code...

- What does this do?

The Try-Catch-Finally Clause

- In Java, can always write:

```
try {
   // some stuff in here
} catch (ExceptionOne e) {
   // some stuff in here
} catch (ExceptionTwo e) {
   // some stuff in here
} catch (ExceptionThree e) {
   // some stuff in here
} finally {
   // some stuff in here
}
```

In either case (exception or not) the “finally” block is run after you finish

- What does this do?
  - — “finally” always executes after the “try”, no matter what
  - — why is this useful?

Now Let’s Look at an Example

```
public void writeList(Vector <Integer> v) {
   PrintWriter out = null;

   try {
     out = new PrintWriter(new FileWriter("OutFile.txt"));
     for (int i = 0; i < SIZE; i++)
       out.println("Val at: " + i + " = " + v.elementAt(i));
   } catch (ArrayIndexOutOfBoundsException e) {
     System.err.println("Caught "
       + "ArrayIndexOutOfBoundsExcept: " + e.getMessage());
   } catch (IOException e) {
      System.err.println("Bad IO: " + e.getMessage());
   } finally {
     if (out != null) {
       System.out.println("Closing PrintWriter");
       out.close();
     } else {
       System.out.println("PrintWriter not open");
     }
   }
}
```

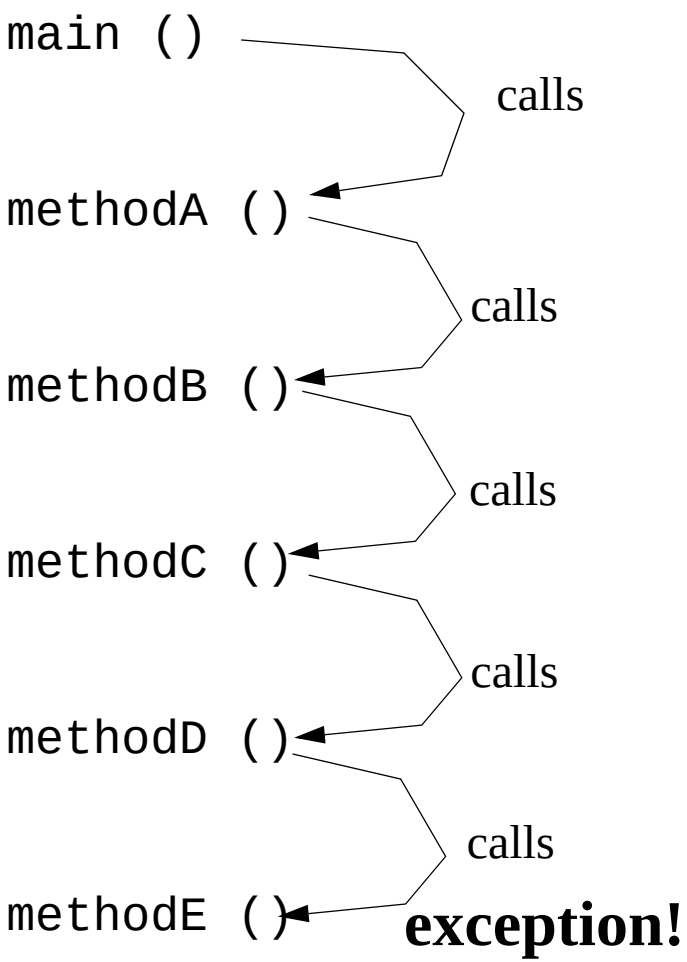
To Throw an Exception In Your Code

- Easy:

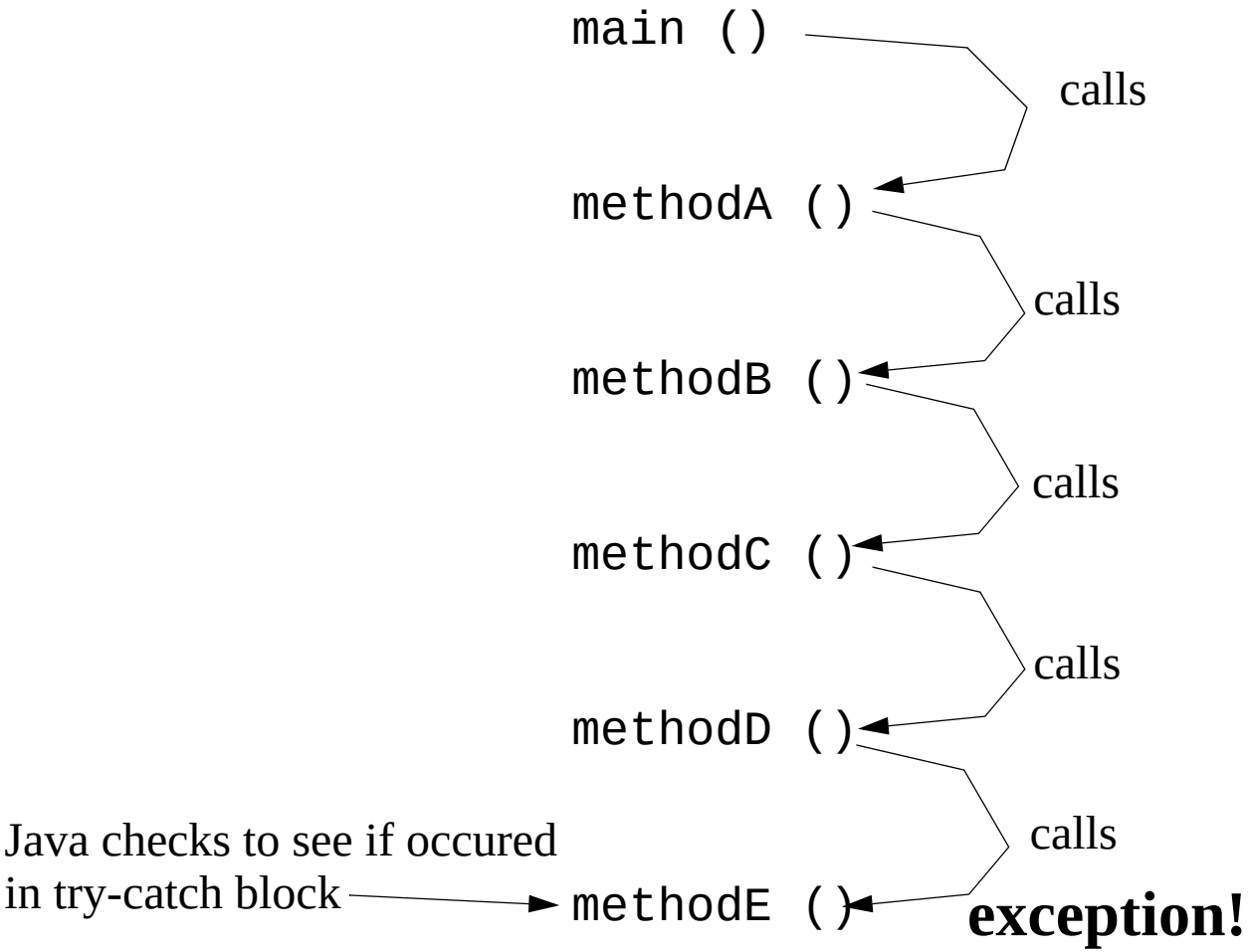
```
myException = new doNotLikeYourFaceException (faceLen);
throw myException;
```

- When execution hits a “throw”, it returns to caller’s “catch”

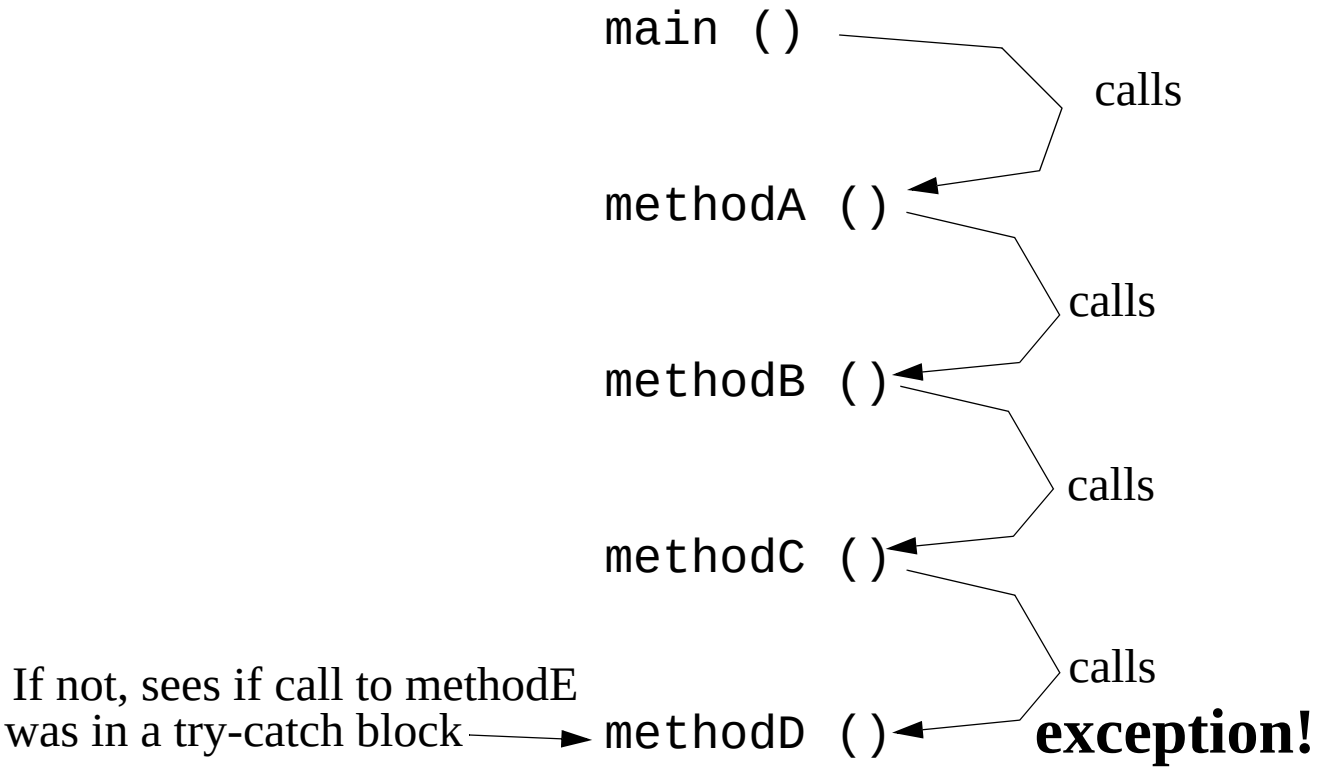
Exceptions and the Call Stack



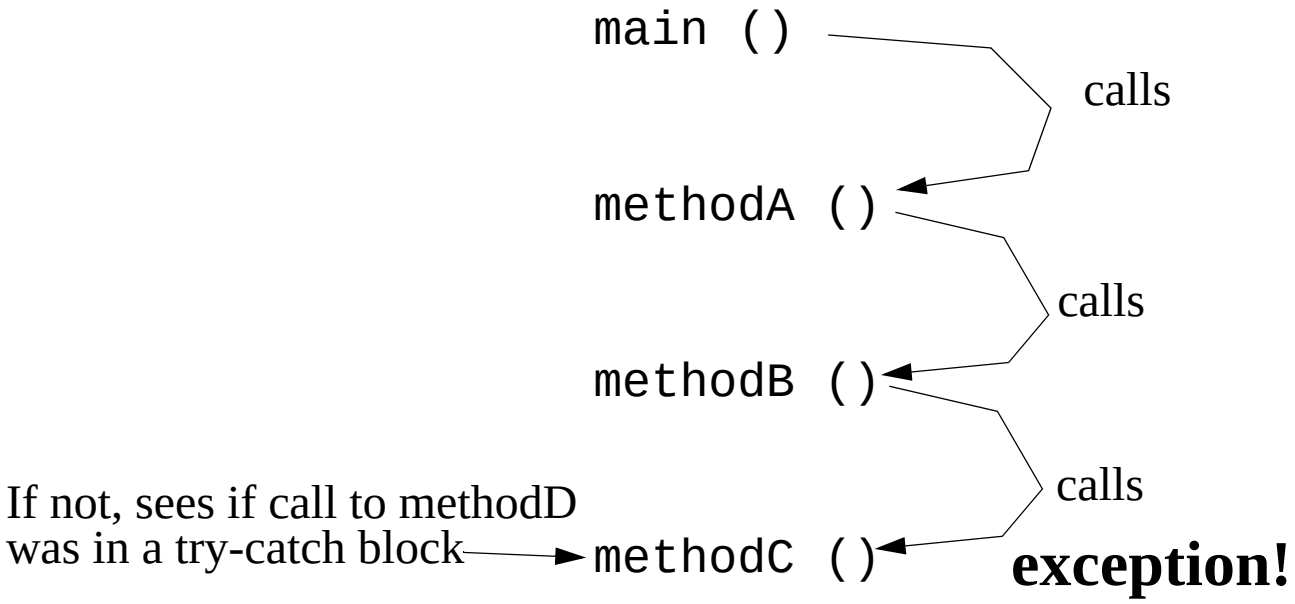
Exceptions and the Call Stack



Exceptions and the Call Stack



Exceptions and the Call Stack



Exceptions and the Call Stack

Sees if call to methodA
was in a try-catch block

```
main ()
```

Goes all the way back
up to main () if needed

Exceptions and the Call Stack

Sees if call to methodA was in a try-catch block

```
main ()
```

If not?

The program crashes...

Checked and Unchecked Exceptions

- Methods can be spec'ed to "throw" exceptions:

```
public void writeList() throws IOException,
    ArrayIndexOutOfBoundsException {}
```

- If you call a method that "throws" a "checked" exception
  - — You **MUST** catch it
  - — Or you must declare that you throw it yourself
  - — Or compiler will barf
  - — Note: if, at runtime, you don't catch a particular exception, you throw it yourself
- If you call a method that "throws" an "unchecked" exception
  - — You need not handle it
  - — If you don't, and exception thrown, you will throw it yourself
  - — Can explicitly "throw" an unchecked exception yourself, w/o declaring it

Are Two Hierarchies of UnCh. Exceptions

- class “Error”
  - — Generally meant to correspond to case when outside world intrudes
  - — Out of memory, a resource seems to disappear, etc.
- class “RuntimeException”
  - — Programmer error

Note: when you need to throw an exception, ask:

- — Does an existing exception type exist?
- — If not, where do I create one?
- — If I need to create one, do I need additional methods?

Are Two Hierarchies of UnCh. Exceptions

- class “Error”
  - — Generally meant to correspond to case when outside world intrudes
  - — Out of memory, a resource seems to disappear, etc.
- class “RuntimeException”
  - — Programmer error

  Note: when you need to throw an exception, ask:
  - — Does an existing exception type exist?
  - — If not, where do I create one?
  - — If I need to create one, do I need additional methods?
- everything else is checked!

Are Several Useful Methods in Throwable

- Allow you to do things like “chain” exceptions

```
catch (ExceptionTypeOne e1) {
    ...
    e1.printStackTrace ();
    ...
    ExceptionTypeTwo e2 (e1);
    ...
    throw e2;
}
```

- Suggest you check it out!

Example of Exceptions As Error Reporting

- Why are exceptions a nice idea in general?
  - — The provide a natural way for a method to fail, and to give back important info

Example

- Imagine a method with the signature:

```
Packet getNextPacket ()
```

- This wraps up a network connection
- Returns a packet upon completion
- Many reasons this might not work
  - — connection down, timeout, checksum bad...
- Without exceptions, things can be inconvenient
  - — You'd end up adding "error" objects to everything
  - — Like the following...

```
class PacketContainer {
    Packet returnVal;
    ConnectionError myError;

    PacketContainer (Packet inputReturnVal) {
        returnVal = inputReturnVal;
    }
    PacketContainer (ConnectionError inputError) {
        myError = inputError;
    }
    Packet getReturnVal () {
        return returnVal;
    }
    ConnectionError getMyError () {
        return myError;
    }
    boolean gotPacket () {
        return myError == null;
    }
}
```

Then, To Use the getNextPacket ()…

```
PacketContainer result = getNextPacket ()

if (result.gotPacket ()) {
    Packet myPacket = result.getReturnVal ();
    ...
} else {
    ConnectionError myError = result.getMyError ();
    ...
}
```

- Hmm... this looks **exactly** like a try-catch
- **AND** I had to spend 10 minutes writing a stupid class to do this!
- Any time it’s a pain to do something, less likely it’s done
  - — Classc C method: have some global error variable that gets set
  - — This is why it is very nice to have exceptions in Java

One Thing About Exceptions…

- Do not use them for control flow!
- Here's some bad code:

```
try {
  for (int i = 0; true; i++) {
    myVec.setItem (myVec.getItem (i) + 1, i);
  }
} catch (outOfBoundsException e) {
  return;
}
```

Now That I've Argued Exceptions are Good

- Onto (perhaps) the biggest controversy in Java-world...

  When Should You Use Checked Exceptions??

Now That I've Argued Exceptions are Good

- Onto (perhaps) the biggest controversy in Java-world...

  When Should You Use Checked Exceptions??

Really gets your blood boiling, huh?!?!?!?

Like if I say: "The tea party movement... good or bad? Discuss!"

Now That I've Argued Exceptions are Good

- Onto (perhaps) the biggest controversy in Java-world...

  When Should You Use Checked Exceptions??

- It appears there are two camps here

Always **Never**

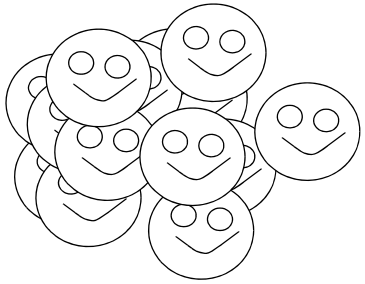
Now That I've Argued Exceptions are Good

- Onto (perhaps) the biggest controversy in Java-world...

  When Should You Use Checked Exceptions??

- It appears there are two camps here

Always



Never

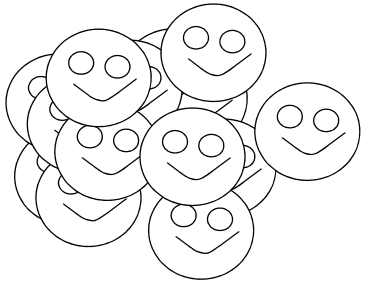
Now That I've Argued Exceptions are Good

- Onto (perhaps) the biggest controversy in Java-world...

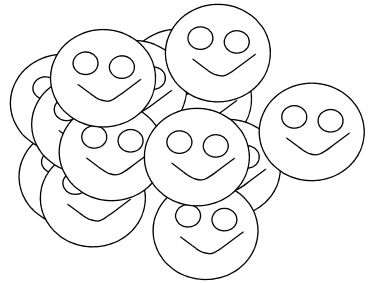
  When Should You Use Checked Exceptions??

- It appears there are two camps here

Always



Never



Now That I've Argued Exceptions are Good

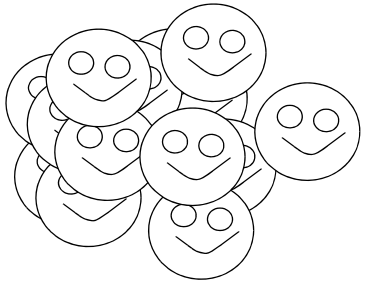
- Onto (perhaps) the biggest controversy in Java-world...

  When Should You Use Checked Exceptions??

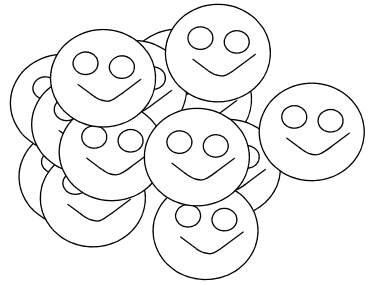
- It appears there are two camps here

  — And then there is Chris

Always



Never



Let's Examine the Common "Pro" Position

- It says:
  - — Use checked exceptions for almost every possible error you could encounter
  - — Except if is an error you are quite sure no one could ever recover from
- But the reason to use them in every other case is that...
  - — ...it forces people to think about all of the errors that could happen in their code

Let's Examine the Common "Con" Position

- It says:
  - — Don't use checked exceptions. Just don't do it.
- Why?
  - — It's a **HUGE** pain to write try-catch when you don't want to. The net effect is empty catch blocks all over the place, or tons of useless lines of code
  - — They question whether the benefit on the last slide really exists
  - — They ask: why subject people to this without empirical study?
- Plus, as I indicated before...
  - — Checked exceptions can be abused as a strange form of message passing...
  - — ...where you send messages up multiple hops in the call stack

see http://www.mindview.net/Etc/Discussions/CheckedExceptions for a nice discussion of some of this

Let's Examine the "Chris" Position

- It says:
  - — Don't use unchecked exceptions. Just don't do it.
- Utility even in things like "arrayIndexOutOfBounds"
  - — Almost forces printing/logging info re what program was doing when it died
  - — Is it more useful to see:

```
"exception at ........... line 20
 called by .......
 called by .......
 called by ......."
```

  - — Or:

```
"When I was checking to see if there were any bytes
 coming across the connection I over-ran the end of
 the buffer. Perhaps I didn't resize the buffer?
 At .......... line 20...."
```

Let's Examine the "Chris" Position

- What I'm trying to say is:
  - — Checked exceptions **force** you to think about...
  - — And (hopefully) describe nicely...
  - — What could go wrong when you write code
- Plus
  - — Who is **EVER** gonna catch an unchecked exception?
  - — So if don't want to force handling, why not print the call stack, kill the program?
  - — That's what's gonna happen anyway, right?

The Moral of the Story

- The Java community...
  - — ...does not seem to know what to do with checked vs. unchecked exceptions
  - — Even the gurus seem to disagree
- I have my opinion...
  - — ...which is why you got checked exceptions in A2
- But you can take it or leave it
  - — Just understand what the issues are

Questions?