Comp 401 Exceptions

Instructor: Prasun Dewan

Prerequisite

- Inheritance
- Interfaces
- Input
- Iterator

Exceptions: What?

- Exceptions have to do with error handling
  - Run time
  - Custom
- Error kinds
  - Internal errors in a program (e.g. off –by-one)
  - External errors
    - User input
    - File system
    - Other distributed programs

Why Study Error Handling

- Better user experience
- Security

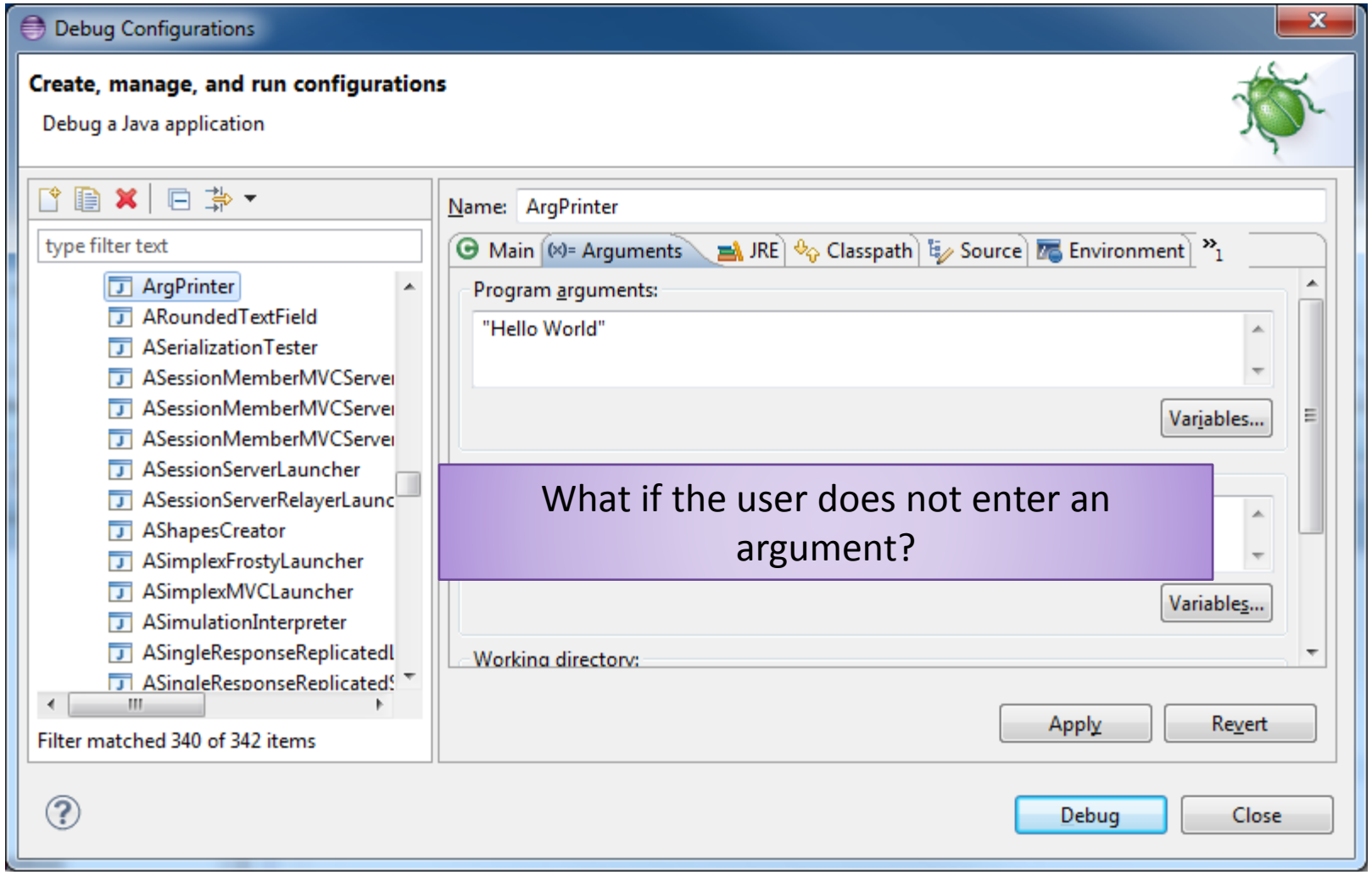
Why Exception Support for Error Handling

- Error handing can be done without exceptions
- Exceptions promote software engineering principles on error handling
  - Easier to program
  - Efficiency
  - Separation of concerns (modularity)
    - Ease of change
    - Ease of understanding
  - Concerns and exceptions?
    - Non erroneous code
    - Erroneous code

Argument Printer

```
public class ArgPrinter {
  public static void main(String[] args) {
      System.out.println(args[0]);
  }
}
```

Correct Usage



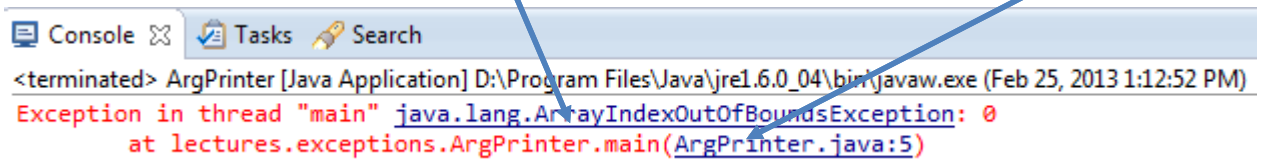

```
<terminated> ArgPrinter [Java Application] D:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Feb 25, 2013 1:17:29 PM)
Hello World
```

User does not enter Argument: Default Exception Handling

```
package lectures.exceptions;
public class ArgPrinter {
  public static void main(String[] args) {
      System.out.println(args[0]);
  }
}
```

Type of exception (array index exception) reported to the user

Trace of all methods in the call chain (stack) shown



Implies Java is checking each array element access for correct index (subscript checking) rather than simply going to some unanticipated (valid or invalid) memory address. Core dumped in some languages when hardware detects that memory address is out of range

Message understandable to programmer but not user

Safe Arg Printer with Error Check

```
package lectures.exceptions;
  public class SafeArgPrinter {
    public static void main(String[] args) {
```

```
  }
}
```

Extra check

Java checks for subscript error.

Regular and error code undifferentiated for reader

May want to ignore errors on first pass.

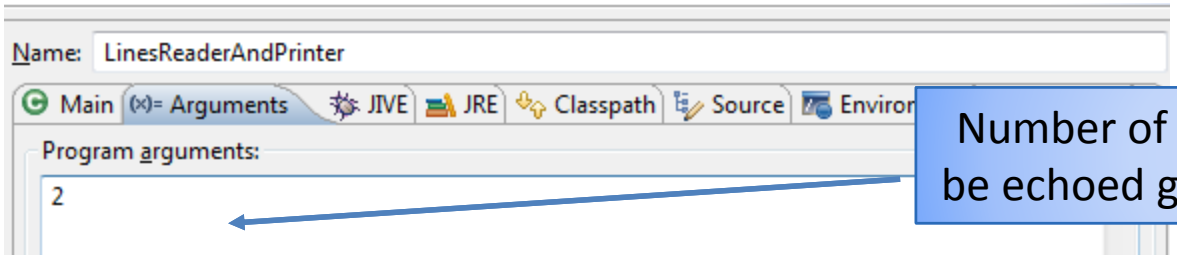
Safe Code with Try-Catch

```
package lectures.exceptions;
public class ExceptionHandlingArgsPrinter {
  public static void main(String[] args) {
    try {
      [redacted]
    }
  }
}
```

No extra subscript check

Try-catch separates regular and error-handling code.

Multiple Line Echoer



Number of input lines, N, to be echoed given as argument

Programs echoes N input lines and terminates

```
<terminated> LinesReaderAndPrinter [Java Application] D.
Hello world
Hello world
Goodbye world
Goodbye world
```

User may enter no argument

User argument may be a non number

User may enter more than N lines?

User may enter less than N lines

And enter the EOF character

Errors?

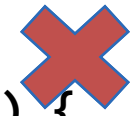
Processing of program argument and user input in different methods

Call Chains and Exceptions

```
public class LinesReaderAndPrinter {
  public static void main (String args[])  {
    echoLines(numberOfInputLines(args));
  }
  …
}
```

Erroneous numberOfInputLines Function

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter an argument.");
    }
}
```



Ignoring NumberFormalException to simplify example

Return value?

Multiple catch blocks can be associated with a try block

Error recovering numberOfInputLines Function

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter an argument.");
        return  0;
     }
}
```

Error recovery: Program not halted

This caller of function want to assume 1 instead of 0

Caller should decide what to do

ECHOLINES: GIVE UP

```
static BufferedReader input =
        new BufferedReader(
                new InputStreamReader(System.in));
static void echoLines (int numberOfInputLines) {
  try {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
  }catch (IOException e) {
    System.out.println("Did not input " + numberOfInputLines +
            " input strings before input was closed. ");
    System.exit(-1);
  }
}
```

Decision to halt without full context

Maybe echoing fewer lines is ok for this caller and only a warning can be given or a confirmation can be asked from user.

Caller should decide what to do

Moral: Separate error detection and handling

- In this example
  - Let echoLines() and numberOfInputLines() not do the error handling.
  - All they should do is error detection.
  - Main should handle error reporting and associated UI

How to Pass the Buck?

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter an argument.");
        return  0;
     }
}
```

Passing the buck: numberOfInputLines: Error Code

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        return  -1;
     }
}
```

Caller can decide what to do

Does not work if -1 was legal value

Does not work if function returned unsigned value (supported in some languages and simulatable in Java)

How to Pass the Buck for Procedure?

```
static BufferedReader input =
  new BufferedReader (new InputStreamReader(System.in));
static void echoLines (int numberOfInputLines) {
  try {
    for (int inputNum = 0;inputNum <
numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
  } catch (IOException e) {
    System.out.println("Did not input " + numberOfInputLines +
            " input strings before input was closed. ");
    System.exit(-1);
  }
}
```

Procedure does not return any value

Can convert it into a function that returns an error code

Converting Procedure to Function: Returning Error Codes

```
static BufferedReader input =
  new BufferedReader (new InputStreamReader(System.in));
static int echoLines (int numberOfInputLines) {
  try {
    for (int inputNum = 0;inputNum <
numberOfInputLines;inputNum++)
      System.out.println(input.readLine());
    return 0;
   } catch (IOException e) {
      return -1;
   }
```

Fake function with side effects

Passing the Back: Error code solution

- Pass back error codes to caller
  - Procedure
    - "Works" as we can make it return value instead of void
    - Programmer cannot distinguish between true and fake procedure
  - Function
    - Either unsigned integer (int, short, long) converted into much larger signed integers
    - Or error code may be legal return value if integer function returns all possible values

Global variable solution?

- Store error codes in common variables
  - Does not work when there are multiple calls to the same method
    - A call may overwrite value written by another call
  - Variable may accessed by other methods sharing its scope
    - Least privilege violated

How to Propagate Exception?

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter an argument.");
        return  0;
     }
}
```

Do not handle it.

Exception propagation

- Java lets exceptions be "returned" instead of regular values.
  - No need to overload function return value or make procedure return value
- These propagate through call chain until some method responds to them by catching them
- Caller should know that it should catch exception

Propagating echoLines

```
static void echoLines(int numberOfInputLines) {
  for (int inputNum = 0; inputNum < numberOfInputLines;
inputNum++)
    System.out.println(input.readLine());
  }
}
```

How to tell caller that it should catch exception?

Propagating echoLines

```
static void echoLines(int numberOfInputLines)
                  throws IOException {
  for (int inputNum = 0; inputNum < numberOfInputLines;
inputNum++)
    System.out.println(input.readLine());
  }
}
```

Tells caller that passing it or throwing the exception, a special return value

Propagating numberOfInputLines

```
static int numberOfInputLines(String[] args)
                        throws ArrayIndexOutOfBoundsException {
  return Integer.parseInt(args[0]);
}
```

Tells caller that passing it or throwing the exception

Handling in Caller

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Did not enter an argument. Assuming a single input
line.");
      echoLines(1);
  } catch (IOException e) {
    System.out.println("Did not input the correct number of input strings
before input was closed. ");
  }
}
```

IO exception not caught

Has context

Propagating echoLines (Review)

```
static void echoLines(int numberOfInputLines)
                  throws IOException {
  for (int inputNum = 0; inputNum < numberOfInputLines;
inputNum++)
    System.out.println(input.readLine());
  }
}
```

Tells caller that passing it or throwing the exception, a special return value

Propagating numberOfInputLines (Review)

```
static int numberOfInputLines(String[] args)
                        throws ArrayIndexOutOfBoundsException {
  return Integer.parseInt(args[0]);
}
```

Tells caller that passing it or throwing the exception

Handling in Caller (Review)

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Did not enter an argument. Assuming a single input
line.");
      echoLines(1);
  } catch (IOException e) {
    System.out.println("Did not input the correct number of input strings
before input was closed. ");
  }
}
```

IO exception not caught

Has context

Nested Exceptions

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Did not enter an argument. Assuming a single input
line.");
    try {
      echoLines(1);
    } catch (IOException ioe) {
      System.out.println("Did not input the one input string, which is the
default in case of missing argument, before input was closed. ");
   }
 } catch (IOException e) {
    System.out.println("Did not input the correct number of input strings
before input was closed. ");
 }
}
```

Parameters in nested trys must have different names

Parameters in sibling trys can have same name

Implicit Throwing, Multiple Thrown Exceptions

```
public static void main(String args[]) throws IOException,
ArrayIndexOutOfBoundsException {
   echoLines(numberOfInputLines(args));
}
```

Bad idea as interpreter's messages may be meaningless to the user

Not catching and hence indirectly throwing an exception that was thrown to it

Multiple exceptions can be "thrown"

Both thrown exceptions acknowledged

ArrayIndexBounds Alternatives

```
static int numberOfInputLines(String[] args) {
    try {
        return Integer.parseInt(args[0]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter an argument.");
    }
}
```

Throws clause is a form of documentation like comments, annotations, interfaces, and assertions.

Purely a documentation feature, exception propagated regardless of whether it exists

```
static int numberOfInputLines(String[] args)
                    throws ArrayIndexOutOfBoundsException {
  return Integer.parseInt(args[0]);
}
```

```
static int numberOfInputLines(String[] args){
  return Integer.parseInt(args[0]);
}
```

Caller does not know what it must handle

It may not have code to handle

Java knows method is not catching a potential exception but does not complain about lack of documentation

Omitting IOException in Throws clause

```
public static void main(String args[]) throws
ArrayIndexOutOfBoundsException {
   echoLines(numberOfInputLines(args));
}
```

IOException not acknowledged

```
static void echoLines(int numberOfInputLines) {
  for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
    System.out.println(input.readLine());
  }
}
```

Java complains IOException neither handled nor acknowledged

Java has two kinds of exceptions

- Unchecked exceptions
  - Uncaught exceptions need not be acknowledged in method headers (if they can be thrown)
- Checked exceptions
  - Uncaught exceptions must be acknowledged in header of **class and interface method (**if they can be thrown)
  - Tells caller that passing it or throwing the excep[illegible]a special return value
- Rationale for division?

```
public void echoLines(int numberOfInputLi[illegible]s IOException
```

Acked in class method header

```
public void echoLines(int numberOfInputLines) throws IO[illegible]eption {
  for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
    System.out.println(input.readLine());
  }
}
```

Acked in class method header

Not caught in body

Lack of Uniformity

```
static int numberOfInputLines(String[] args){
  return Integer.parseInt(args[0]);
}
```

ArrayIndexOutOfBounds need not be acknowledged

```
static void echoLines(int numberOfInputLines) {
  for (int inputNum = 0; inputNum < numberOfInputLin
    System.out.println(input.readLine());
  }
}
```

IOException must be acknowledged

Misleading header

```
static void safeArrayIndexer() throws ArrayIndexOutOfBoundsException {
    String args[] = {"hello", "goodbye"};
    System.out.println(args[1]);
}
```

Array index out of bounds guaranteed to not happen

- Array indexing does not imply exception
- It implies it may happen
- Halting problem prevents Java from knowing if an exception will really be thrown
  - In this case it can but does not
- Its exception checking assumes the worst case – if an exception could be thrown (not if it is actually thrown)
- In the case of checked exceptions, if there is an operation that may throw it cries wolf
- Making all exceptions checked would make coding of such methods painful and mislead the reader
- Principles for determining which exceptions should be checked?

Checked /Unchecked vs. Kinds of Errors

- User and other external error
- Internal error

Which are preventable?

Errors vs. Prevention

- User and other external error
  - Programmer cannot prevent it
  - Should be acked as form of documentation
- Internal error
  - Programmer can prevent
  - A method that can be erroneous probably is not really erroneous
  - Acking is probably misleading and needlessly increases programming overhead

Justification of Java Rules

Java rules justified if:

- Checked (Non-runtime) exceptions = user and other external errors
- Unchecked (runtime) exceptions = internal errors

Problems with Java rules

```
static int numberOfInputLines(String[] args) {
  return Integer.parseInt(args[0]);
}
```

User error causes unchecked exception

Approach 1: Voluntarily list exception

```
static int numberOfInputLines(String[] args)
                        throws ArrayIndexOutOfBoundsException {
  return Integer.parseInt(args[0]);
}
```

```
public static void main(String args[]) throws IOException {
   echoLines(numberOfInputLines(args));
}
```

No way to force every caller that does not handle it to ack it.

Approach 2: Convert to existing checked exception

```
static int numberOfInputLines(String[] args) throws
IOException {
  try {
        return Integer.parseInt(args[0]);
  } catch (ArrayIndexOutOfBoundsException e) {
      throw new IOException ("First argument missing");
  }
}
```

Exception object thrown explicitly

message

```
public static void main(String args[]) throws IOException {
   echoLines(numberOfInputLines(args));
}
```

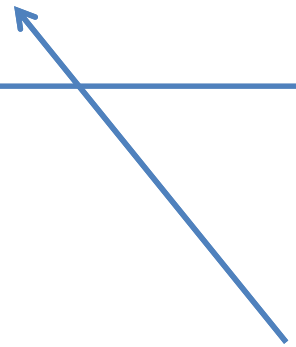
Forces every caller that does not handle it to ack it.

Exception message distinguishes it from read exception

Must query exception and do string equals() to type it

Approach 3: Convert to new checked exception

```
static int numberOfInputLines(String[] args) throws
AMissingArgumentException {
try {
        return Integer.parseInt(args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
       throw new AMissingArgumentException("First argument
missing");
}
```



Our own exception

Type of exception rather than message explains the exception

Programmer-Defined Exception Class

```
public class AMissingArgumentException extends
IOException{
  public AMissingArgumentException(String message) {
    super(message);
  }
}
```

No interface, as Java does not have one for Exceptions

Not adding any methods

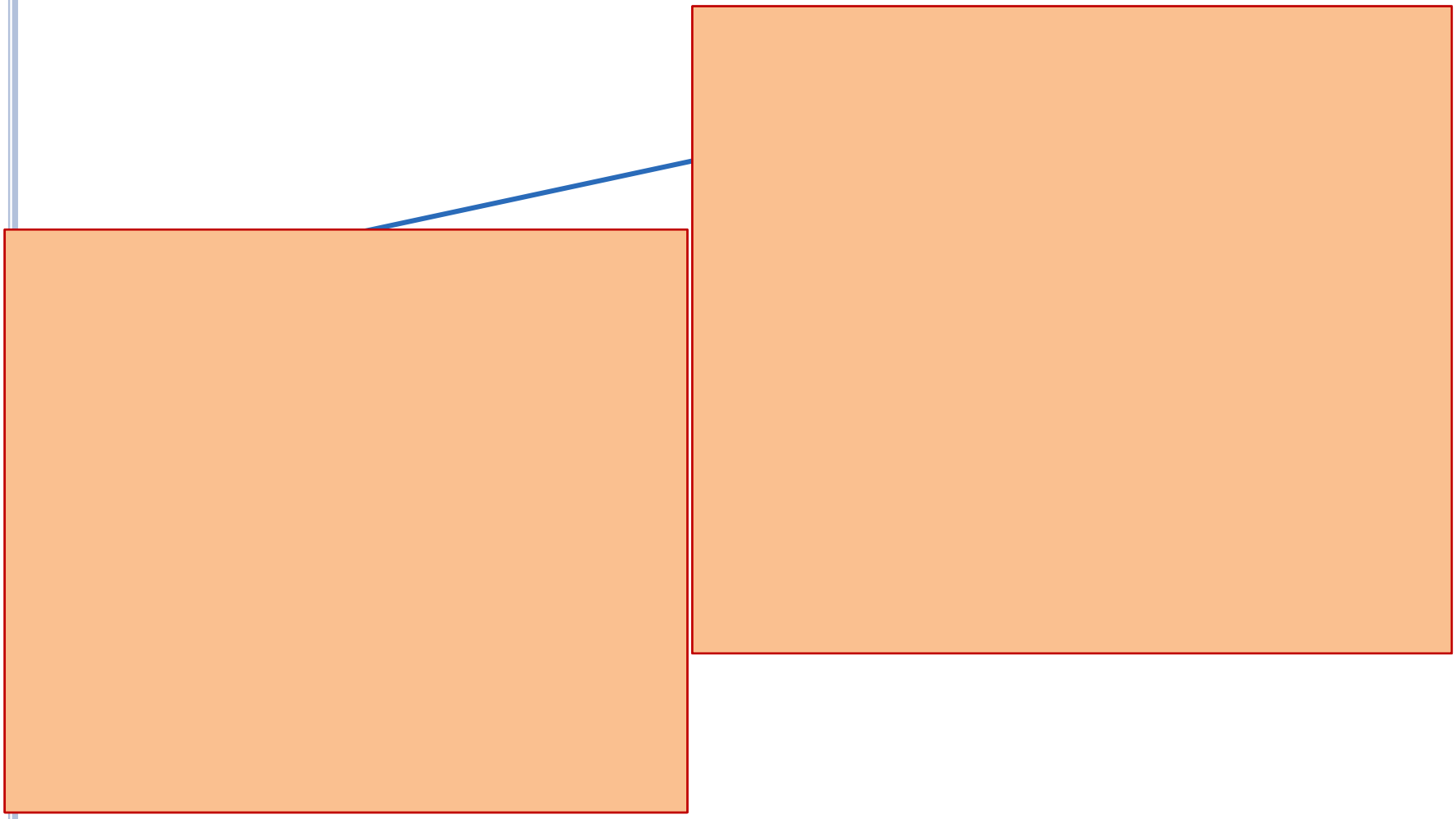
Checked?

A missing argument error is an input/output error

Checked vs. Unchecked Programmer-defined Exceptions

- An exception class must be subclass of existing exception classes
- (Direct or indirect) subclasses of RuntimeException are unchecked
  - Also called “runtime”, but all exceptions are runtime!
  - (Direct or indirect) subclasses of RuntimeException
    - E.g. ArrayIndexOutofBoundsException
- All other exception classes are checked

Part of Exception Hierarchy



Unchecked

Checked

Error Handling → Error Recovery and reporting

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (AMissingArgumentException e) {
    System.out.println("Did not enter an argument. Assuming a
single input line.");
    try {
      echoLines(1);
    } catch (IOException ioe) {
      System.out.println("Did not input the one input string,
which is the default in case of missing argument, before input
was closed. ");
   }
  } catch (IOException e) {
     System.out.println("Did not input the correct number of
input strings before input was closed. ");
  }
}
```

Exception object not used

Error Handling → Reporting Error Message, Using Catch Parameter

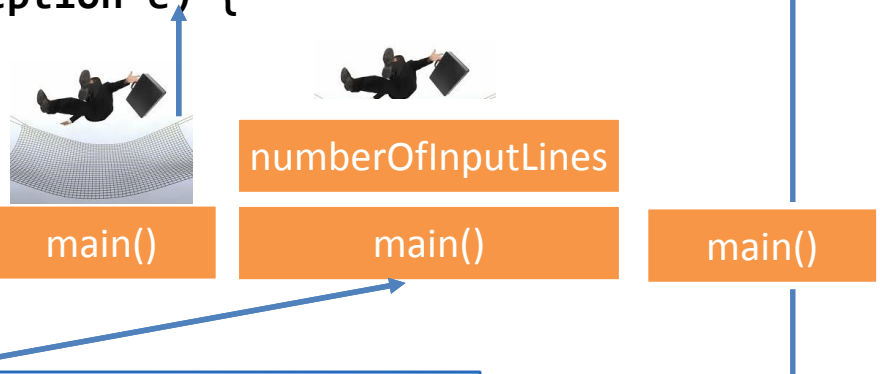
```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
   } catch (AMissingArgumentException e) {
    System.out.println(e);
    System.exit(-1);
   } catch (IOException e) {
    System.out.println(e);
    System.exit(-1);
 }
}
```

e.getMessage()

```
<terminated> LinesReaderAndPrinterUsingProgrammerDefinedException [Java Application] D:\Program Files\Java\jre1.6.0_04\
lectures.exceptions.AMissingArgumentException: First argument missing
```

Printing Stack Trace, Using Catch parameter

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
   } catch (AMissingArgumentException e) {
    e.printStackTrace();
    System.exit(-1);
   } catch (IOException e) {
    e.printStackTrace();
    System.exit(-1);
 }
}
```



Stack when exception is thrown, not when it is printed.

```
<terminated> LinesReaderAndPrinterUsingProgrammerDefinedException [Java Application] D:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Feb 25, 2013 10:15:23 PM)
lectures.exceptions.AMissingArgumentException: First argument missing
        at lectures.exceptions.LinesReaderAndPrinterUsingProgrammerDefinedException.numberOfInputLines(LinesReaderAndPrinterUsingProgrammerDefinedExceptio
        at lectures.exceptions.LinesReaderAndPrinterUsingProgrammerDefinedException.main(LinesReaderAndPrinterUsingProgrammerDefinedException.java:11)
```

Remove code duplication?

Removing Code Duplication

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (IOException e) {
    e.printStackTrace();
    System.exit(-1);
 }
}
```

AMissingArgumentException IS-A (IS-Subclass of) IOException

Superclass before Subclass

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (IOException e) {
     e.printStackTrace();
     System.exit(-1);
  } catch (AMissingArgumentException e) {
     System.out.println(e);
     System.exit(-1);
  }
}
```

List exception subclass before superclass

AMissingArgumentException processed here

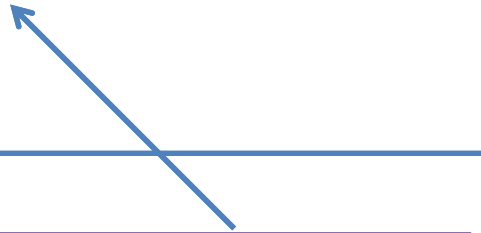
Unreachable block

SUBCLASS BEFORE SUPERCLASS

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
  } catch (AMissingArgumentException e) {
     System.out.println(e);
     System.exit(-1);
  } catch (IOException e) {
     e.printStackTrace();
     System.exit(-1);
  }
}
```

Other Exceptions?

```
static int numberOfInputLines(String[] args) {
  return Integer.parseInt(args[0]);
}
```



Exception?

NumberFormatException

Uncaught Exception?

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
} catch (IOException e) {
    e.printStackTrace();
    System.exit(-1);
 }
}
```

NumberFormatException not caught

More Robust Handling

```
public static void main(String args[]) {
  try {
    echoLines(numberOfInputLines(args));
} catch (Exception e) {
    e.printStackTrace();
    System.exit(-1);
 }
}
```

NumberFormatException caught

Exceptions and Initialization Approaches

```
int numberOfInputLines = 5;
```

Initializing Declaration

Initialize while declaring

```
int numberOfInputLines;
```

Un-initializing Declaration

Declare and then initialize in constructor (or some other method)

```
numberOfInputLines = 5;
```

Constructor initialization

Exceptions and Initialization Approaches

```
public class InitializationAndCheckedExceptions {
  int  numberOfInputLines = numberOfInputLines(new String[]
{"45"});
   …
}
```

Checked exception not being handled or acknowledged

```
public class InitializationAndCheckedExceptions {
int  numberOfInputLines;
  public InitializationAndCheckedExceptions() {
    try {
      numberOfInputLines = numberOfInputLines(new String[]
{"45"});
    } catch (AMissingArgumentException e) {
      e.printStackTrace();
    }
   }
}
```

Must call functions that can throw unchecked exceptions in constructors rather than during initialization

Exceptions Example

```
static int numberOfInputLines(String[] args) throws AMissingArgumentException
{
  try {
    return Integer.parseInt(args[0]);
  } catch (ArrayIndexOutOfBoundsException e) {
    throw new AMissingArgumentException("Missing first argument");
  } catch (NumberFormatException e) {
    System.out.println("Non number argument. Returning 0.");
    return 0;
  }
}
```

How to determine how long the method takes?

System.currentTimeMills()

```
static int numberOfInputLines(String[] args) throws AMissingArgumentException
{
  try {
    int retVal = Integer.parseInt(args[0]);
    return retVal;
  } catch (ArrayIndexOutOfBoundsException e) {
    AMissingArgumentException missingArgumentException =
      new AMissingArgumentException("Missing first argument");
    throw missingArgumentException;
  } catch (NumberFormatException e) {
    System.out.println("Non number argument. Returning 0.");
    return 0;
  }
}
```

Code duplication

FINALLY

```
static int numberOfInputLines(String[] args) throws AMissingArgumentException
{
  long startTime = System.currentTim
  try {
    return Integer.parseInt(args[0])
  } catch (ArrayIndexOutOfBoundsExcep
    System.out.println(System.currentTimeMillis() - startTime);
    throw new AMissingArgumentException("Missing first argument");
  } catch (NumberFormatException e) {
    System.out.println("Non number argument, Returning 0.");
    return 0;
  }
  finally {

  }
}
```

All paths that lead from try block end up in finally. Finally executed after the try block and before the code following the try block

Finally without catch?

Multiple Paths Through an IF

```
public static int factorial(int n) {
   System.out.println("Started factorial:"+ n);
   if (n <= 1) return 1;
    return n * factorial(n-1);
  }
}
```

Try and Finally without Exceptions

```
public static int factorial(int n) {
  System.out.println("Started factorial:"+ n);
  try {
    if (n <= 1) return 1;
    return n * factorial(n-1);
  }
  finally {
     System.out.println("Ended factorial:" + n);
  }
}
```

All paths that lead from try block end up in finally. Finally executed after the try block and before the code following the try block

Finally and Postconditions?

Finally for PostConditions and Invariants

```
public boolean preSetWeight (double newWeight) {
  return newWeight > 0;
}
public void setWeight(double newWeight) {
  assert preSetWeight(newWeight);
  try {
    if (!preSetWeight(newWeight)) return;
    weight = newWeight;
  } finally {
    assert preSetWeight(newWeight); // invariant – post and pre condition
  }
}
```

Intra-method propagation

```
static Object[] list = {5, "hello", "goodbye" };
```

```
for (int i = 0; i < list.length; i++) {
  try {
    System.out.println((String) list[i]);
  } catch (ClassCastException e) {
    System.out.println(e);
  }
```

```
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
hello
goodbye
```

```
try {
  for (int i = 0; i < list.length; i++) {
    System.out.println((String) list[i]);
  }
} catch (ClassCastException e) {
  System.out.println(e);
}
```

```
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

Println terminated, catch executed, and loop continues

println terminated and exception propagated to enclosing loop, which is also terminated, and catch executed

Terminating Program vs. Continuing

- Independent errors can be collected
  - Scanning: **int** 5a = 50
- Dependent errors cannot be:
  - Parsing: 5 + 2 4 / - 2

Try Catch Semantics

- A try catch block has
  - One try block
  - One or more parameterized catch blocks
  - Zero or one finally block
- When an exception occurs in a try block
  - Remaining code in the try block is abandoned
  - The first catch block that can handle the exception is executed
- The try catch block terminates when
  - The try block executes successfully without exception
  - A catch block finishes execution
    - Which may throw its own exceptions that may be caught or not through try blocks
- The finally block is called after termination of the try catch block and before the statement following the try catch block
  - All paths from the associated try catch block lead to it

Checked Exception Review

- Checked exceptions
  - Uncaught exceptions must be acknowledged

Catching vs. Acknowledging

```
static void echoLines (int numberOfInputLines) {
  try {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
  }catch (IOException e) {
```

```
static void echoLines (int numberOfInputLines) throws IOException {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
```

```
static void echoLines (int numberOfInputLines) {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
}
```

```
static void echoLines (int numberOfInputLines) throws IOException{
  try {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
  }catch (IOExcepti
    System.out.pri                                    utLines +
            " input strings before input was closed. ");
    System.exit(-1);
  }
}
```

Allowed as caller will still work, though it will have extra handling

In real-life, analogous rules exist

Real Life Analogy

Should overstate rather than understate bad side effect
Dizziness, headache

If you are bad, you should not say you are good.
People will be disappointed
If you are good, you can say you are bad
You don't let people down

Makes sense if there is some uncertainty

Why Allow False Positives

```
static void echoLines (int numberOfInputLines) throws IOException {
  try {
    for (int inputNum = 0;inputNum < numberOfInputLines;inputNum++)
        System.out.println(input.readLine());
  }catch (IOException e) {
    System.out.println("Did not input " + numberOfInputLines +
            " input strings before input was closed. ");
    System.exit(-1);
  }
}
```

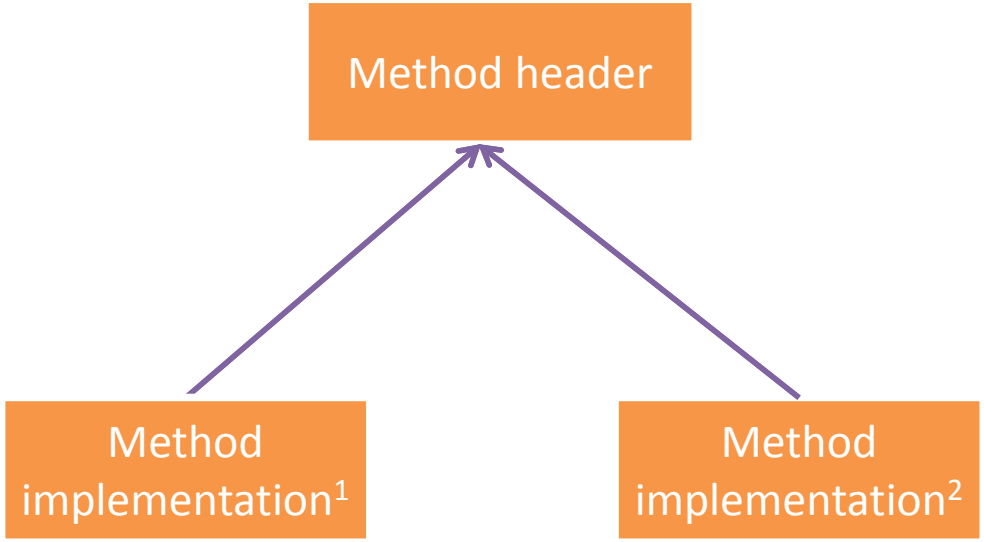
Uncertainty in this case?

No path can lead to an IO exception

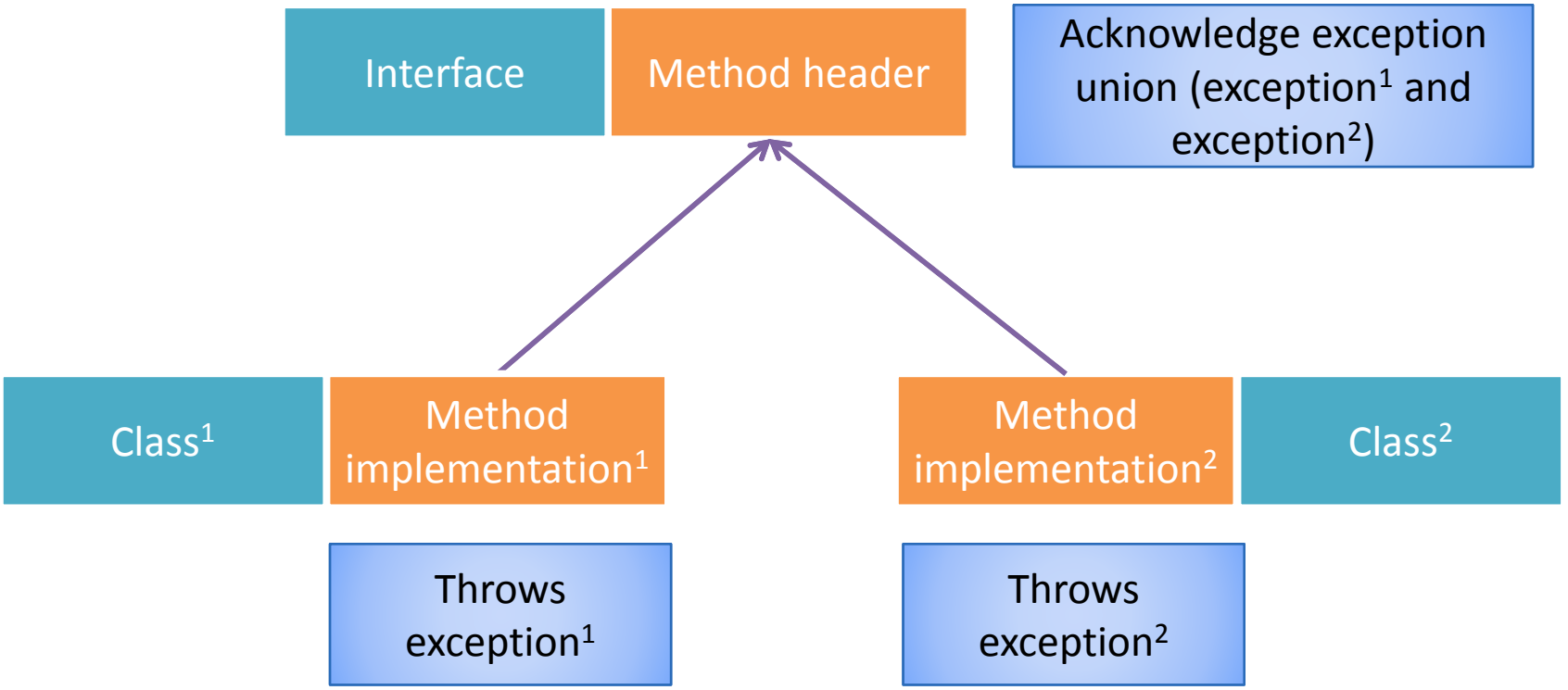
In general, Java cannot tell if an exception throwing path is taken

Also method body may evolve (from stub to full implementation)

Instance Method variations



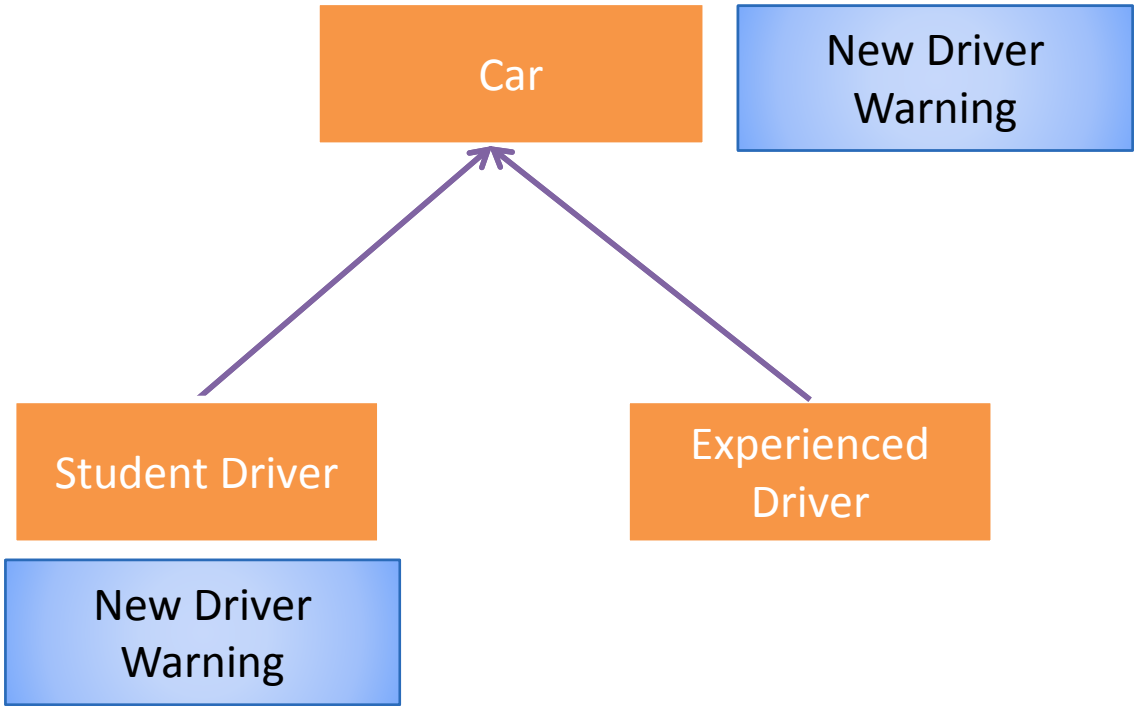
Interface Instance Method Variations



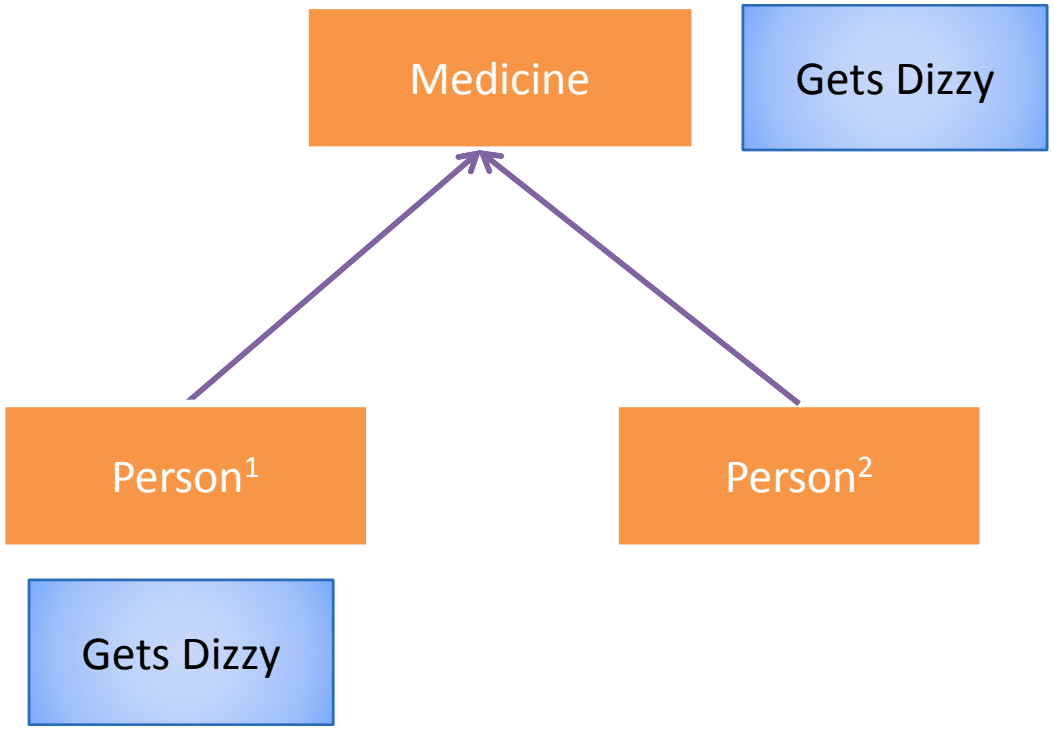
Real Life Analogy

A car used by experienced drivers could have this sign without doing harm.

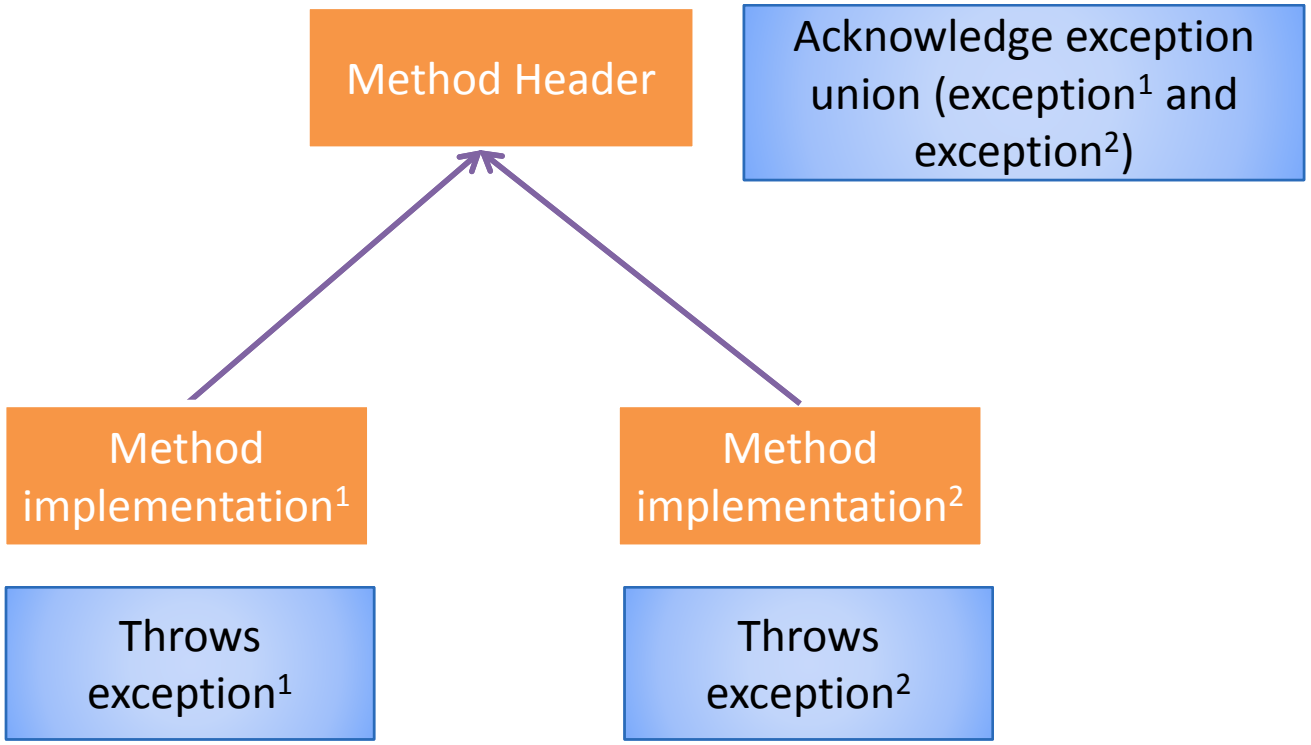
Car Driven by Experienced and New driver



Medicine used by Reactive and Non Reactive Person



Instance Method Variations Throwing Different Exceptions



An Iterator Implementation

```
public class AllUpperCaseLettersInOrder implements CharIterator {
  char nextLetter = 'A';
  public boolean hasNext() {
    return nextLetter <= 'Z';
  }
  public char next() {
   char retVal = nextLetter;
   nextLetter = (char) (nextLetter + 1);
   return retVal;
  }
}
```

Exceptions?

```
public interface CharIterator {
  public char next();
  public boolean hasNext();
}
```

Matching header for each interface method

A Safer Implementation

```
public class AllUpperCaseLettersInOrderThrowingException
             implements CharIterator {
  char nextLetter = 'A';
  public boolean hasNext() {
    return nextLetter <= 'Z';
  }
  public char next() throws NoSuchElementException {
    if (!hasNext())
      throw new NoSuchElementException();
    char retVal = nextLetter;
    nextLetter = (char) (nextLetter + 1);
    return retVal;
  }
}
```

Good idea to throw this exception when no more elements so caller knows what went wrong

Checked or unchecked?

Unchecked, as most users expected to call hasNext() before next()

Unmatched Headers

```
public class AllUpperCaseLettersInOrderThrowingException
             implements CharIterator {
  char nextLetter = 'A';
  public boolean hasNext() {
    return nextLetter <= 'Z';
  }
  public char next() throws NoSuchElementException {
    if (!hasNext())
      throw new NoSuchElementException();
    char retVal = nextLetter;
    nextLetter = (char) (nextLetter + 1);
    return retVal;
  }
}
```

```
public interface CharIterator {
  public char next();
  public boolean hasNe
}
```

Interface and class headers do not match

OK as exception is unchecked

Catching Expected Events

Less efficient, extra check

```
try {
  for (;;) {
    if (!scanner.hasNext()) break;
    System.out.println(Integer.parseInt(scanner.nextLine()));
  }
} catch (NumberFormatException |   e) {
    e.printStackTrace();
}
```

Better style

Differentiating between expected and unexpected events

```
try {
  for (;;) {
    System.out.println(Integer.parseInt(scanner.nextLine()));
  }
} catch (NoSuchElementException e) {
} catch (NumberFormatException e) {
    e.printStackTrace();
}
```

Better style trumps over efficiency

Exceptions

- Support typed errors
- Provide a way to customize error handling
  - By default Java will terminate program
- Allows more efficient error processing
- Allows separation of error-handling and normal-processing code
- Allows error handling and error detection to be in separate classes and methods
  - Without passing legal non erroneous values (null/-1)
- Allows separate classes to provide error UI

Exceptions

- Errors may be internal or external
- Exceptions allow custom error handling to be done while following software engineering principles
- Try and catch blocks allow programmers to easily separate error handling and non error handling code
- Sometimes error handing should be distributed among error detecting method and its callers
- Need a way for error information to be passed to caller, that is, propagate error information'
- Checked and unchecked exceptions

Try Catch Semantics

- A try catch block has
  - One try block
  - One or more parameterized catch blocks
  - Zero or one finally block
- When an exception occurs in a try block
  - Remaining code in the try block is abandoned
  - The first catch block that can handle the exception is executed
- The try catch block terminates when
  - The try block executes successfully without exception
  - A catch block finishes execution
    - Which may throw its own exceptions that may be caught or not through try blocks
- The finally block is called after termination of the try catch block and before the statement following the try catch block
  - All paths from the associated try catch block lead to it