

# COMP 401 EXCEPTIONS

**Instructor: Prasad 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 handling 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

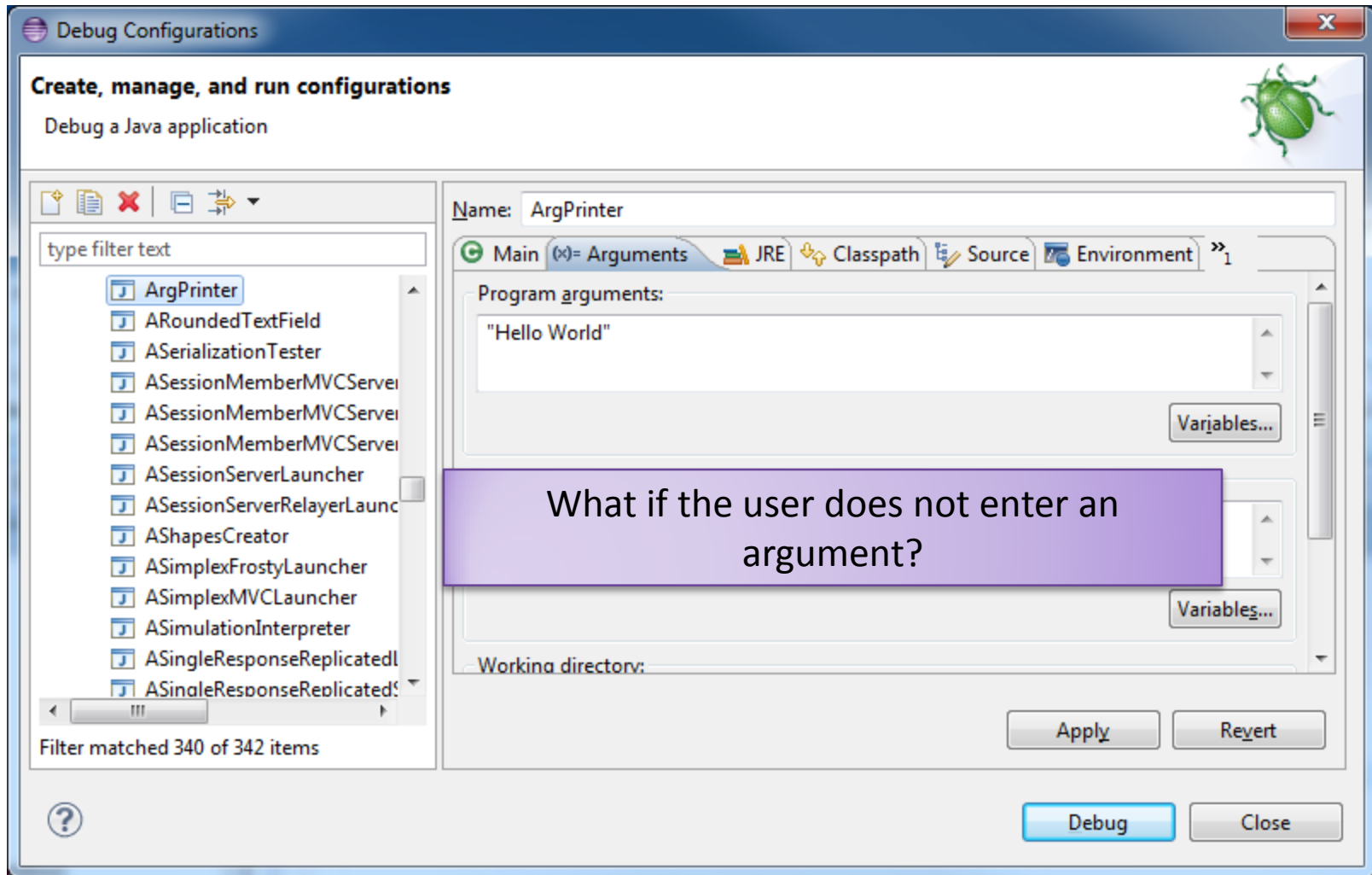


# ARGUMENT PRINTER

```
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
```

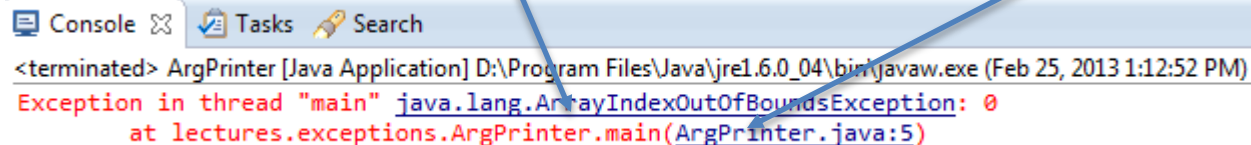


# USER DOES NOT ENTER ARGUMENT: DEFAULT EXCEPTION HANDLING

```
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



Console Tasks Search  
<terminated> ArgPrinter [Java Application] D:\Program Files\Java\jre1.6.0\_04\bin\javaw.exe (Feb 25, 2013 1:12:52 PM)  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  
at lectures.exceptions.ArgPrinter.main(ArgPrinter.java:5)

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) {
        if (args.length == 0) {
            System.out.println( "Did not specify the argument to
            be printed. Terminating program.");
            System.exit(-1);
        } else {
            System.out.println(args[0]);
        }
    }
}
```

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

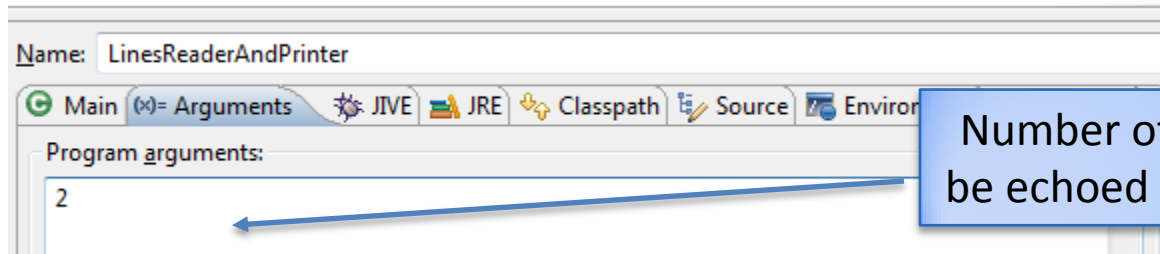
```
package lectures.exceptions;
public class ExceptionHandlingArgsPrinter {
    public static void main(String[] args) {
        try {
            System.out.println(args[0]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Did not specify the argument to be
printed. Terminating program.");
            System.exit(-1);
        }
    }
}
```

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 NumberFormatException 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) {  
     {  
        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 a) {  
        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 exception is a special return value

- Rationale for division?

```
public void echoLines(int numberOfInputLines) throws IOException
```

Acked in class  
method header

Acked in class  
method header

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

Not caught in  
body



# LACK OF UNIFORMITY

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



ArrayIndexOutOfBoundsException  
need not be  
acknowledged

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

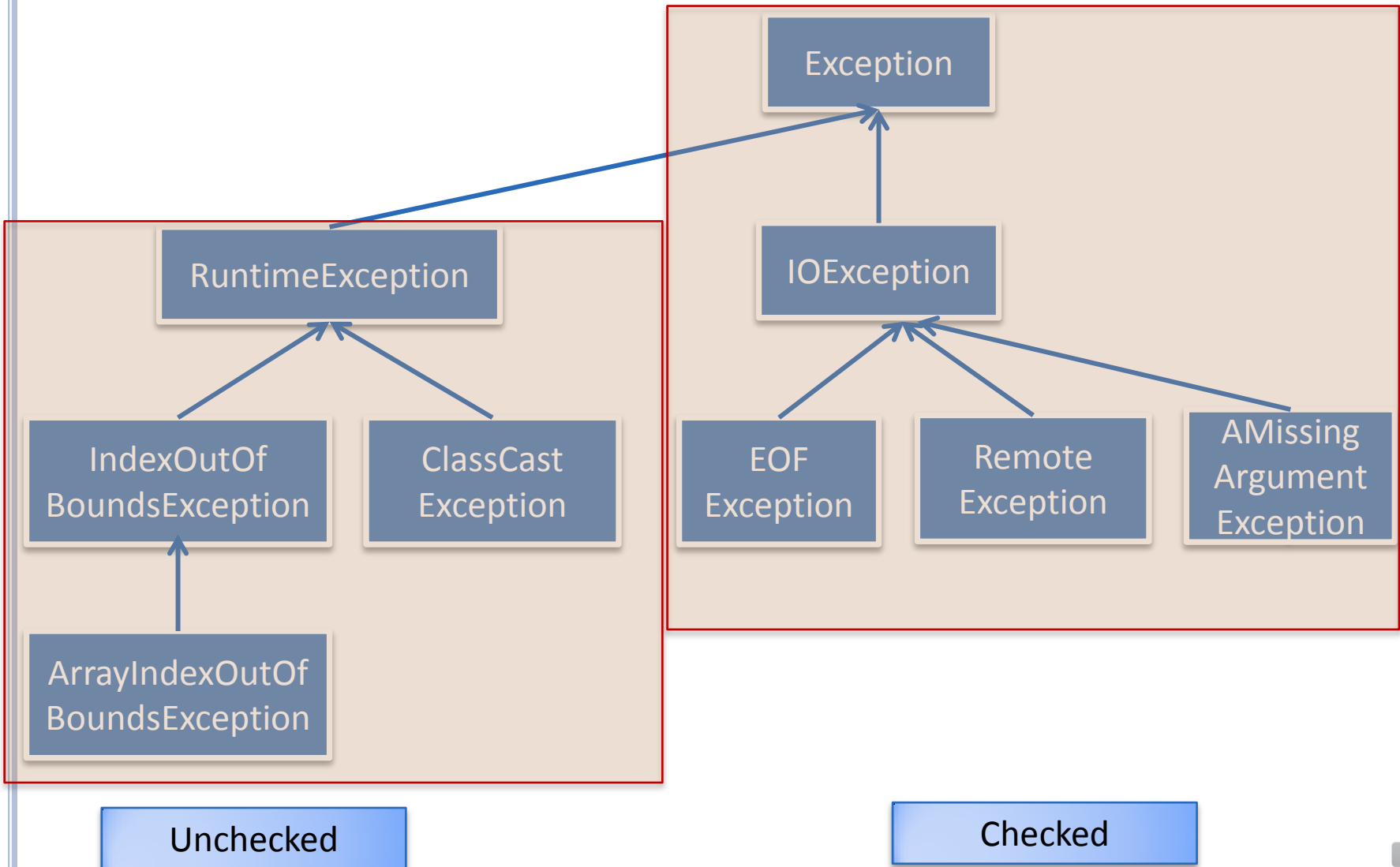


# CHECKED VS. UNCHECKED PROGRAMMER-DEFINED EXCEPTIONS

- 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





# 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()

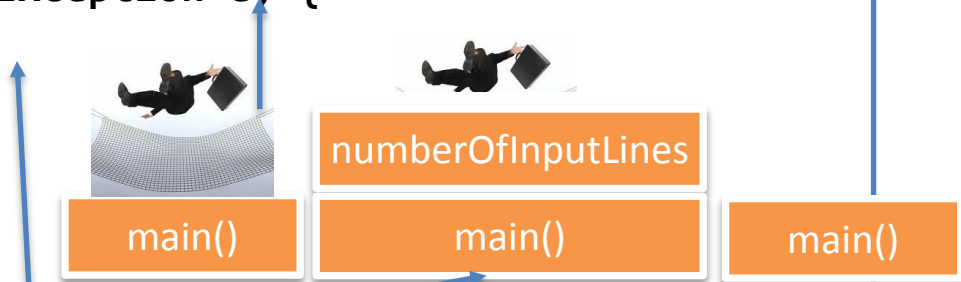


<terminated> LinesReaderAndPrinterUsingProgrammerDefinedException [Java Application] D:\Program Files\Java\jre1.6.0\_04\lectures.exceptions.AMissingArgumentException: First argument missing



# 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.

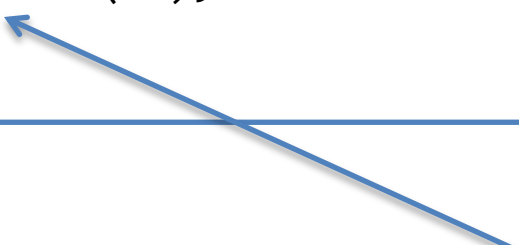
```
<terminated> LinesReaderAndPrinterUsingProgrammerDefinedException [Java Application] D:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Feb 25, 2013 10:15:23 PM)  
lectures.exceptions.AMissingArgumentException: First argument missing  
    at lectures.exceptions.LinesReaderAndPrinterUsingProgrammerDefinedException.numberOfInputLines(LinesReaderAndPrinterUsingP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



A `MissingArgumentException` 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



# NESTED EXCEPTIONS EXAMPLE REVISITED

```
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. ");
    }
}
```

How to determine how long the  
method takes?



# SYSTEM.CURRENTTIMEMILLS()

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

Code duplication

# FINALLY

```
public static void main(String args[]) {
    long startTime = System.currentTimeMillis();
    try {
        echoLines(numberOfInputLines(args));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not enter correct argument. Assuming a single input
line.");
        try {
            echoLines(1);
        } catch (IOException ioe) {
            System.out.println("Did not input the correct number of input strings
default in case of missing argument, but the program will continue to run.");
        }
        System.out.println(System.currentTimeMillis() - startTime);
    }
    } catch (IOException e) {
        System.out.println("Did not input the correct number of input strings
before input was closed. ");
    }
    finally {
        System.out.println(System.currentTimeMillis() - startTime);
    }
}
```

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

Finally without catch? is the

# 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 before the code following the try block

# 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

# CATCHING EXPECTED EVENTS

```
try {
    for (;;) {
        String nextLine = input.readLine();
        if (".".equals(nextLine)) break;
        System.out.println(Integer.parseInt(nextLine));
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Better style

Less efficient,  
extra check

Differentiating  
between expected  
and unexpected  
events

```
try {
    for (;;) {
        System.out.println(Integer.parseInt(input.readLine()));
    }
} catch (Exception e) {
}
```

Better style trumps over efficiency

# 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 (IOException e) {  
        System.out.println("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

**NEW DRIVER**

*"Give me a brake"*<sup>TM</sup>

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)

# INTERFACE ACKS BUT NOT CLASS

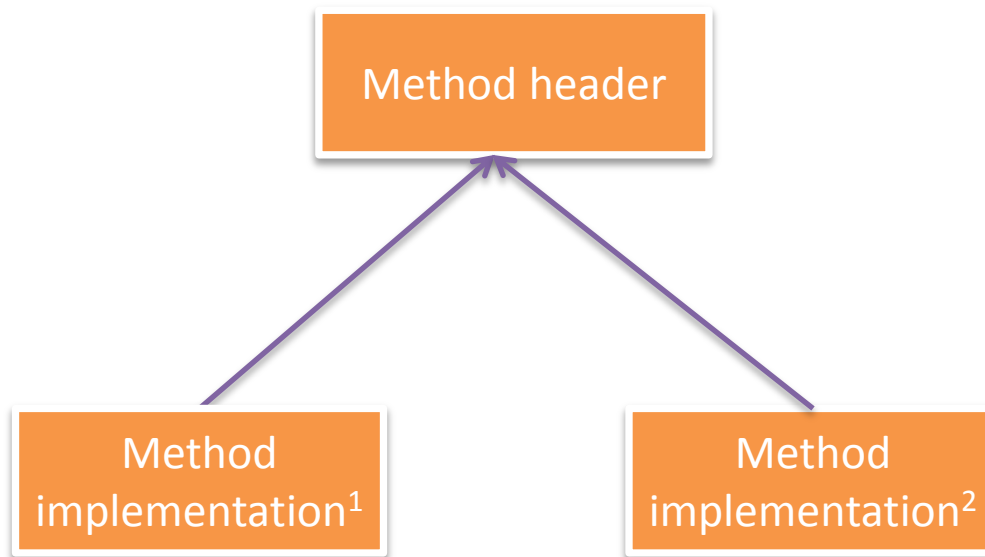
```
public void echoLines(int numberOfInputLines) throws IOException
```

```
public 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);  
    }  
}
```

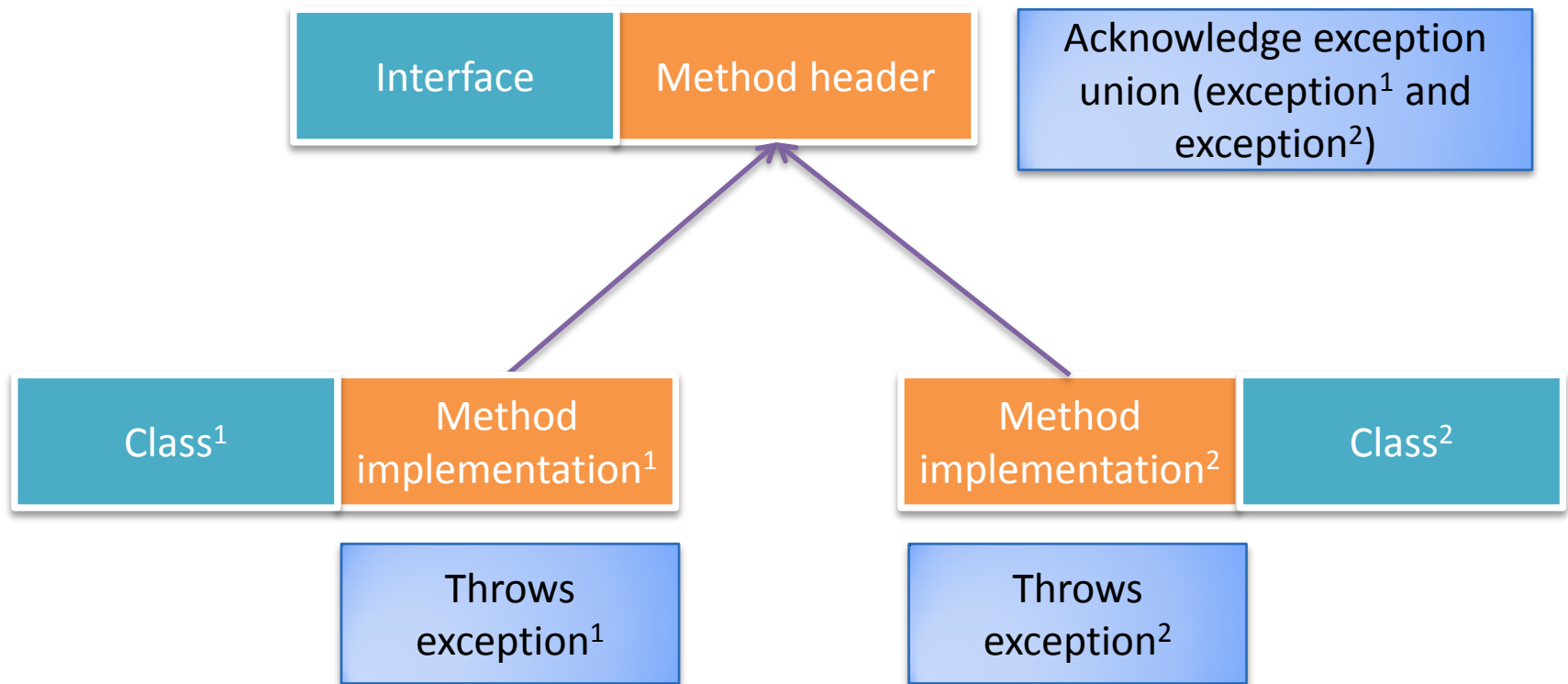
In instance methods, must also consider  
variation of a method header



# INSTANCE METHOD VARIATIONS



# INTERFACE INSTANCE METHOD VARIATIONS



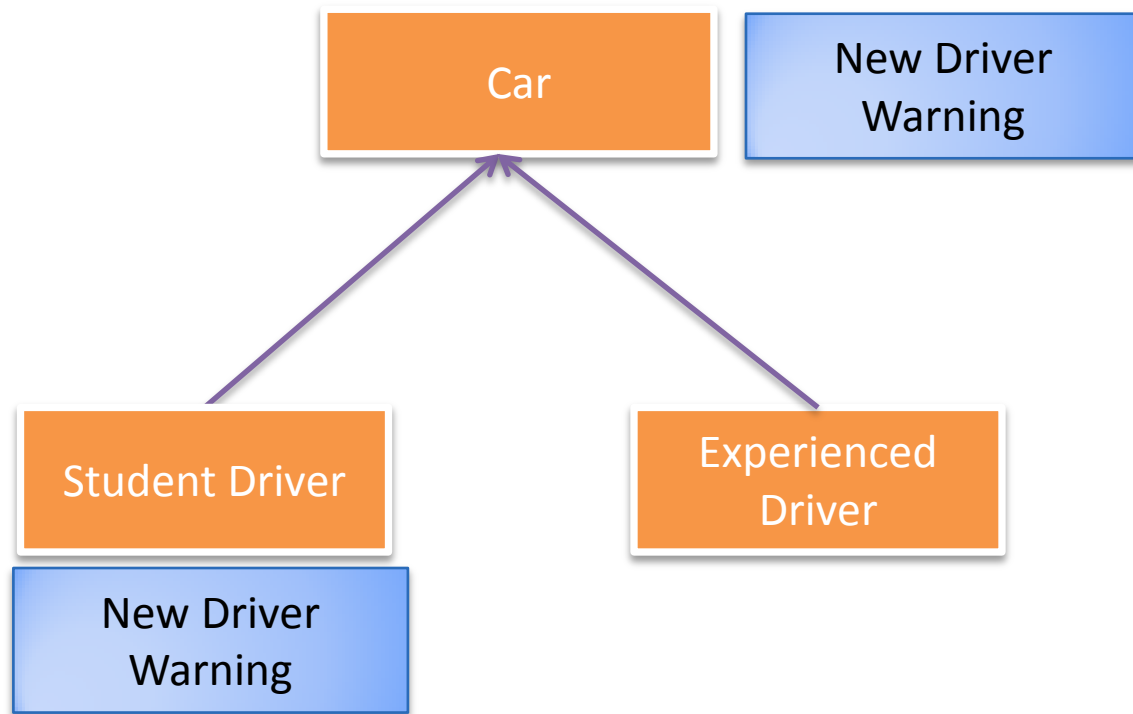
# REAL LIFE ANALOGY

**NEW DRIVER**

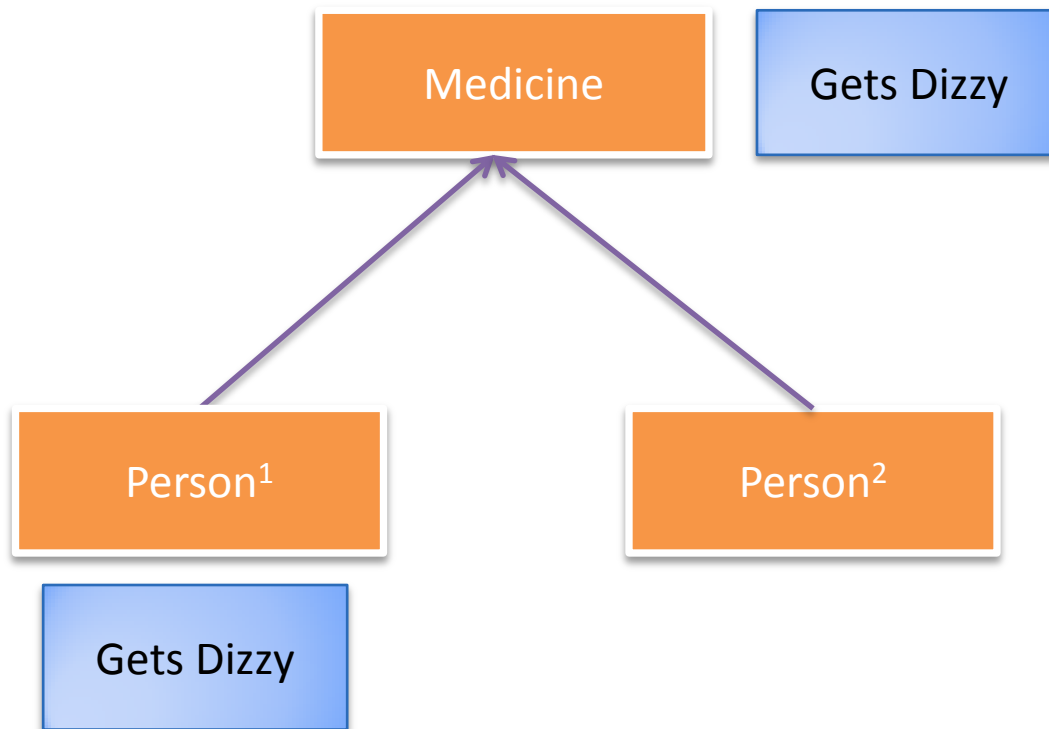
*"Give me a brake"*<sup>TM</sup>

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

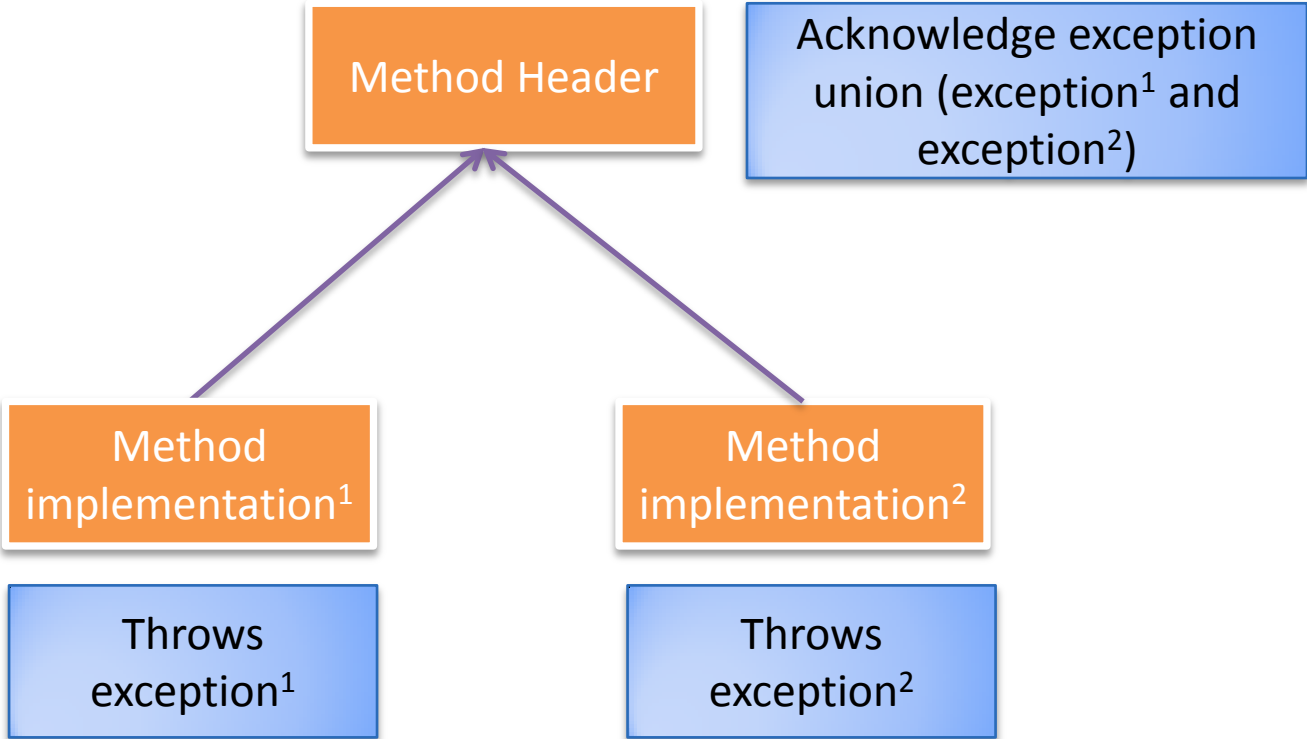
# CAR DRIVEN BY EXPERIENCED AND NEW DRIVER



# MEDICINE USED BY REACTIVE AND NON REACTIVE PERSON



# INSTANCE METHOD VARIATIONS THROWING DIFFERENT EXCEPTIONS



# EXCEPTIONS

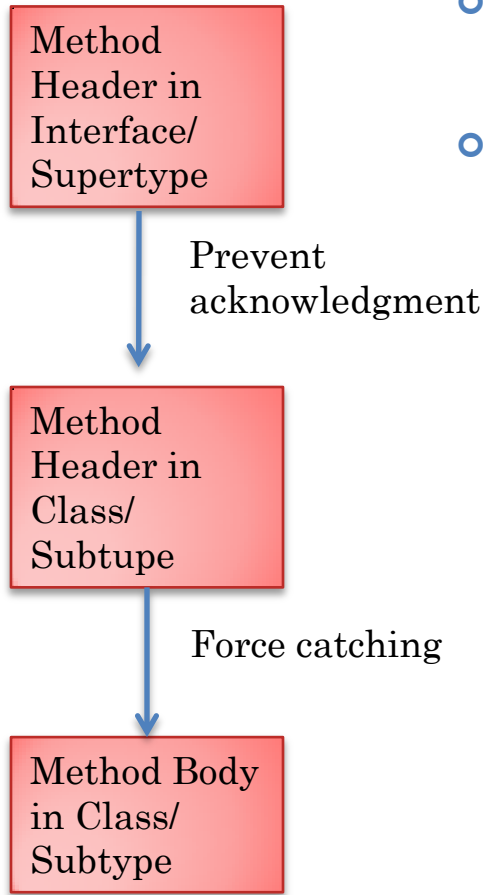
- Support typed errors
- Provide a way to customize error handling
  - By default Java will terminate program
- Allows more efficient error processing
- Allows separation of error-handling and normal-processing code
- Allows error handling and error detection to be in separate classes and methods
  - 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 handling 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



# CHECKED VS. UNCHECKED



- Unchecked
  - No rules
- Checked
  - uncaught in method body => acknowledged in method header
  - unacknowledged in method header => caught in method body (from 1)
  - unacknowledged in interface/supertype method-header => unacknowledged in class/subtype method-header
  - unacknowledged in interface/supertype method-header => caught in method body (from 2 and 3)
  - interface can be used to force method implementations to catch exceptions

# IS-A RULE FOR EXCEPTIONS

- Exception of type T1 uncaught in method body → exception of type T2, where T1 IS-A T2, acknowledged in method header
- Exception of type T1 acknowledged in interface/supertype method-header → exception of type T2, where T2 IS-A T1, acknowledged in class/subclass method-header



# IS-A RULE FOR EXCEPTIONS

- Overzealous handling not allowed
- Can add fine-grained or coarse-grained trys
- Should have exception and not normal handling in catches



# EXTRA SLIDES

# STATIC METHOD EXAMPLE

```
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);  
    }  
}
```

Need instance method example for variations

# INTERFACE AND ITERATOR

```
public interface StringIterator {  
    public String next();  
    public boolean hasNext();  
}
```

```
public class IteratedStringPrinter {  
    public static void print(StringIterator aStringIterator) {  
        while (aStringIterator.hasNext())  
            System.out.println(aStringIterator.next());  
    }  
}
```

# AN ARRAY ITERATOR IMPLEMENTATION

```
public class AnArrayIterator implements StringIterator {
    String[] elements;
    int nextIndex;
    public AnArrayIterator(String[] anElementArray) {
        elements = anElementArray;
    }
    public String next() {
        String retVal = elements[nextIndex];
        nextIndex++;
        return retVal;
    }
    public boolean hasNext() {
        return nextIndex < elements.length;
    }
}
```

```
public interface StringIterator {
    public String next();
    public boolean hasNext();
}
```

# WITH UNCHECKED EXCEPTION

```
public class AnArrayIteratorThrowingUncheckedException
    implements StringIterator {
    String[] elements;
    int nextIndex;
    public AnArrayIteratorThrowingUncheckedException(String[]
anElementArray) {
    elements = anElementArray;
}
    public String next() throws NoSuchElementException {
    if (!hasNext()) throw new NoSuchElementException();
    String retVal = elements[nextIndex];
    nextIndex++;
    return retVal;
}
    public boolean hasNext() {
    return nextIndex < elements.length;
}
}
```

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

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



# UNMATCHED HEADERS

```
public class AnArrayIteratorThrowingUncheckedException
    implements StringIterator {
    String[] elements;
    int nextIndex;
    public AnArrayIteratorThrowingUncheckedException(String[]
anElementArray) {
    elements = anElementArray;
}
public String next() throws NoSuchElementException {
    if (!hasNext()) throw new NoSuchElementException();
    String retVal = elements[nextIndex];
    nextIndex++;
    return retVal;
}
public boolean hasNext() {
    return nextIndex < elements.length;
}
}
```

Interface and class headers do not match

```
public interface StringIterator {
    public String next();
    public boolean hasNext();
}
```

OK as exception is unchecked



# UNMATCHED BODY AND HEADERS

```
public class AnArrayIteratorThrowingUncheckedException
    implements StringIterator {
    String[] elements;
    int nextIndex;
    public AnArrayIteratorThrowingUncheckedException(String[]
anElementArray) {
    elements = anElementArray;
}
public String next() {
    if (!hasNext()) throw new NoSuchElementException();
    String retVal = elements[nextIndex];
    nextIndex++;
    return retVal;
}
public boolean hasNext() {
    return nextIndex < elements.length;
}
```

Body and class headers do not match

```
public interface StringIterator {
    public String next();
    public boolean hasNext();
}
```

OK as exception is unchecked

# CHECKED EXCEPTION: INPUT READING ITERATOR (A LA SCANNER):

```
public class AnInputStreamScannerThrowingIOException
    implements ... {
    BufferedReader input =
        new BufferedReader(new InputStreamReader(System.in));
    public static final String QUIT_COMMAND = "quit";
    String nextLine;
    public AnInputStreamScannerThrowingIOException ()
        throws IOException {
        nextLine = input.readLine();
    }
    public boolean hasNext() {
        return nextLine.equals(QUIT_COMMAND);
    }
    public String next() throws IOException, NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        String retVal = nextLine;
        nextLine = input.readLine();
        return retVal;
    }
}
```

When EOF or other IOException occur

When no more elements



# SCANNER-BASED ITERATOR

```
public class AnInputStreamScannerThrowingIOException
    implements StringIterator
    {
        BufferedReader input =
            new BufferedReader(new InputStreamReader(System.in));
```

Caller of interface method not ready to handle checked exception

Interface header must report union of all (checked) exceptions thrown by implementation

```
public boolean hasNext() {
    return nextLine.equals(QUIT_COMMAND);
}
```

```
public String next() throws IOException, NoSuchElementException {
    if (!hasNext()) throw new NoSuchElementException();
    String retVal = nextLine;
    nextLine = input.readLine();
    return retVal;
}
```

```
public interface StringIterator {
    public String next();
    public boolean hasNext();
}
```

# FORCING CLASS TO CATCH

```
public class AnInputStreamScanner implements StringIterator {  
    BufferedReader input =  
        new BufferedReader(new InputStreamReader(System.in));
```

Unacknowledged in interface method-header → caught in class method body

```
        throws IOException {  
            nextLine = input.readLine();  
        }  
    public boolean hasNext() {  
        return nextLine.equals(QUIT_COMMAND);  
    }  
    public String next() {  
        if (!hasNext()) throw new NoSuchElementException();  
        String retVal = nextLine;  
        try {  
            nextLine = input.readLine();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return retVal;  
    }  
}
```

# CHANGING THE INTERFACE AND INTERFACE USER

```
public interface StringIteratorThrowingIOException {  
    public String next () throws IOException ;  
    public boolean hasNext();  
}
```

```
public static void print(StringIteratorThrowingIOException  
aStringIterator) {  
    try {  
        while (aStringIterator.hasNext())  
            System.out.println(aStringIterator.next());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

# THROWING EXCEPTION IN ONE IMPLEMENTATION

```
public class AnInputStreamScannerThrowingIOException
    implements StringIteratorThrowingIOException {
    BufferedReader input =
        new BufferedReader(new InputStreamReader(System.in));
    public static final String QUIT_COMMAND = "quit";
    String nextLine;
    public AnInputStreamScannerThrowingIOException ()
        throws IOException {
        nextLine = input.readLine();
    }
    public boolean hasNext() {
        return nextLine.equals(QUIT_COMMAND);
    }
    public String next() throws IOException, NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        String retVal = nextLine;
        nextLine = input.readLine();
        return retVal;
    }
}
```

# NOT THROWING EXCEPTION IN ANOTHER IMPLEMENTATION

```
public class AnotherArrayIteratorThrowingUncheckedException implements
StringIteratorThrowingIOException {
    String[] elements;
    int nextIndex;
    public AnotherArrayIteratorThrowingUncheckedException(String[]
anElementArray) {
        elements = anElementArray;
    }
    public String next() throws NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        String retVal = elements[nextIndex];
        nextIndex++;
        return retVal;
    }
    public boolean hasNext() {
        return nextIndex < elements.length;
    }
}
```

Interface user will still work

Interface declares union of all exceptions thrown by implementations

```
public interface StringIteratorThrowingIOException {
    public String next () throws IOException;
    public boolean hasNext();
}
```



# EVEN MORE OVERSTATEMENT

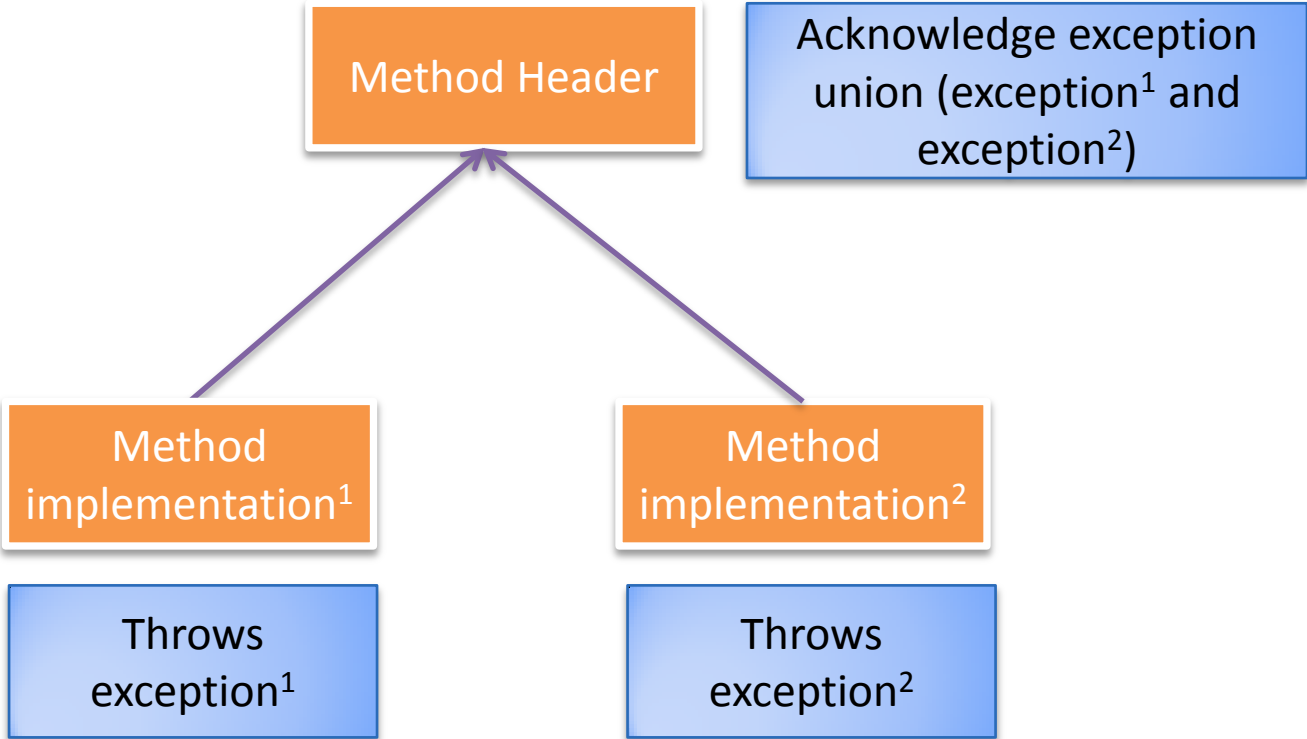
```
public class AnotherArrayIteratorThrowingUncheckedException implements
StringIteratorThrowingIOException {
    String[] elements;
    int nextIndex;
    public AnotherArrayIteratorThrowingUncheckedException(String[]
anElementArray) {
        elements = anElementArray;
    }
    public String next() throws NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        String retVal = elements[nextIndex];
        nextIndex++;
        return retVal;
    }
    public boolean hasNext() {
        return nextIndex < elements.length;
    }
}
```

Interface user will still work

Interface declares union of all exceptions thrown by implementations

```
public interface StringIteratorThrowingIOException {
    public String next () throws Exception;
    public boolean hasNext();
}
```

# INSTANCE METHOD VARIATIONS THROWING DIFFERENT EXCEPTIONS



# CATCHING THROWN EXCEPTION

```
public interface StringIteratorThrowingIOException {  
    public String next () throws IOException ;  
    public boolean hasNext();  
}
```

```
public static void print(StringIteratorThrowingIOException  
aStringIterator) {  
    try {  
        while (aStringIterator.hasNext())  
            System.out.println(aStringIterator.next());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

# OVERZEALOSUNESS: CATCHING MORE THAN THROWN EXCEPTION

```
public interface StringIteratorThrowingIOException {  
    public String next () throws IOException ;  
    public boolean hasNext();  
}
```

```
public static void print(StringIteratorThrowingIOException  
aStringIterator) {  
    try {  
        while (aStringIterator.hasNext())  
            System.out.println(aStringIterator.next());  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (ParseException re) {  
        System.out.println(e);  
    }  
}
```



Java complains  
that checked  
exception not  
thrown by any path

Overstatement allowed by  
thrower but not  
overzealousness by catcher

# CATCHING MORE THAN THROWN EXCEPTION

```
public interface StringIteratorThrowingIOException {  
    public String next () throws IOException ;  
    public boolean hasNext();  
}
```

```
public static void print(StringIteratorThrowingIOException  
aStringIterator) {  
    try {  
        while (aStringIterator.hasNext())  
            System.out.println(aStringIterator.next());  
    } catch (IOException e) {  
        e.printStackTrace();  
    } catch (Exception e) {  
        System.out.println(e);  
    }  
}
```

Super type of thrown  
exception

Exception catch can  
handle checked exceptions  
not covered by  
IOException catch

# 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;  
    }  
}
```

UI mixed in  
computation method

Error recovery: Program not halted

May want to print on pop-up box  
or in different language

May want to assume 1 instead of 0

Caller should decide what to do



# SPECIFIC BENEFITS

- 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



# REAL LIFE ANALOGY FOR OPTIONAL UNCHECKED EXCEPTIONS

Optional warnings on cars





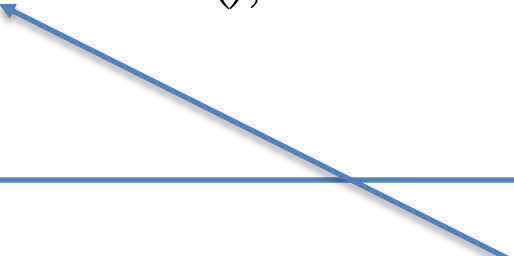
# PRINTING DEBUGGING INFORMATION

- `catch (ArrayIndexOutOfBoundsException e) {`
- `System.out.println(e);`
- `}`

`e.getMessage()`



```
catch (ArrayIndexOutOfBoundsException e) {  
    e.printStackTrace();  
}
```



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



# 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



## WITH EXCEPTION

```
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 hasNext();
}
```

Interface and class headers do not match

OK as exception is unchecked

# SCANNER-BASED ITERATOR

```
public class AnInputStreamScannerThrowingIOException
    implements ... {
    BufferedReader input =
        new BufferedReader(new InputStreamReader(System.in));
    public static final String QUIT_COMMAND = "quit";
    String nextLine;
    public AnInputStreamScannerThrowingIOException ()
        throws IOException {
        nextLine = input.readLine();
    }
    public boolean hasNext() {
        return nextLine.equals(QUIT_COMMAND);
    }
    public String next() throws IOException, NoSuchElementException {
        if (!hasNext()) throw new NoSuchElementException();
        String retVal = nextLine;
        nextLine = input.readLine();
        return retVal;
    }
}
```

When next element erroneous  
because of user error (e.g.  
scanning rules violated)

When no more elements



# SCANNER-BASED ITERATOR

```
public class AnInputStreamScannerThrowingIOException  
    implements StringIterator {
```

```
    public interface StringIterator {  
        public String next();  
        public boolean hasNext();  
    }
```

Unacknowledged in interface method-header → caught in method body

Caller of interface method not ready to handle exception

```
        public String next() throws IOException, NoSuchElementException {  
            if (!hasNext()) throw new NoSuchElementException();  
            String retVal = nextLine();  
            nextLine = inputStream.readLine();  
            return retVal;  
        }  
    }
```

# FORCING CLASS TO CATCH

```
public class AnInputStreamScanner  
    implements StringIterator {
```

```
public interface StringIterator {  
    public String next();  
    public boolean hasNext();  
}
```

```
public String next() {  
    if (!hasNext()) throw new NoSuchElementException();  
    String retVal = nextLine();  
    try {  
        nextLine = inputStream.readLine();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return retVal;  
}}
```



# OVERSTATING IN INTERFACE

```
public class AnInputStreamScanner  
    implements StringIteratorThrowingIOException {
```

```
public interface StringIterator {  
    public String next() throws IOException;  
    public boolean hasNext();  
}
```

Interface user will still work

Some other class may not catch exception

Interface declares union of all exceptions thrown  
by implementations

```
public String next() {  
    if (!hasNext()) throw new NoSuchElementException();  
    String retVal = nextLine();  
    try {  
        nextLine = inputStream.readLine();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return retVal;  
}}
```





# OVERSTATING THROUGH SUPER TYPE

```
public class AnInputStreamScannerThrowingIOException  
    implements StringIterator {
```

```
    public interface StringIteratorThrowingException {  
        public String next() throws Exception;  
        public boolean hasNext();  
    }
```

Supertype (stronger advertisement) allowed in  
interface method

Interface declares union of all exceptions thrown  
by implementations

```
        public String next() throws IOException, NoSuchElementException {  
            if (!hasNext()) throw new NoSuchElementException();  
            String retVal = nextLine;  
            nextLine = inputStream.readLine();  
            return retVal;  
        }  
    }
```

# FORCING CLASS TO CATCH

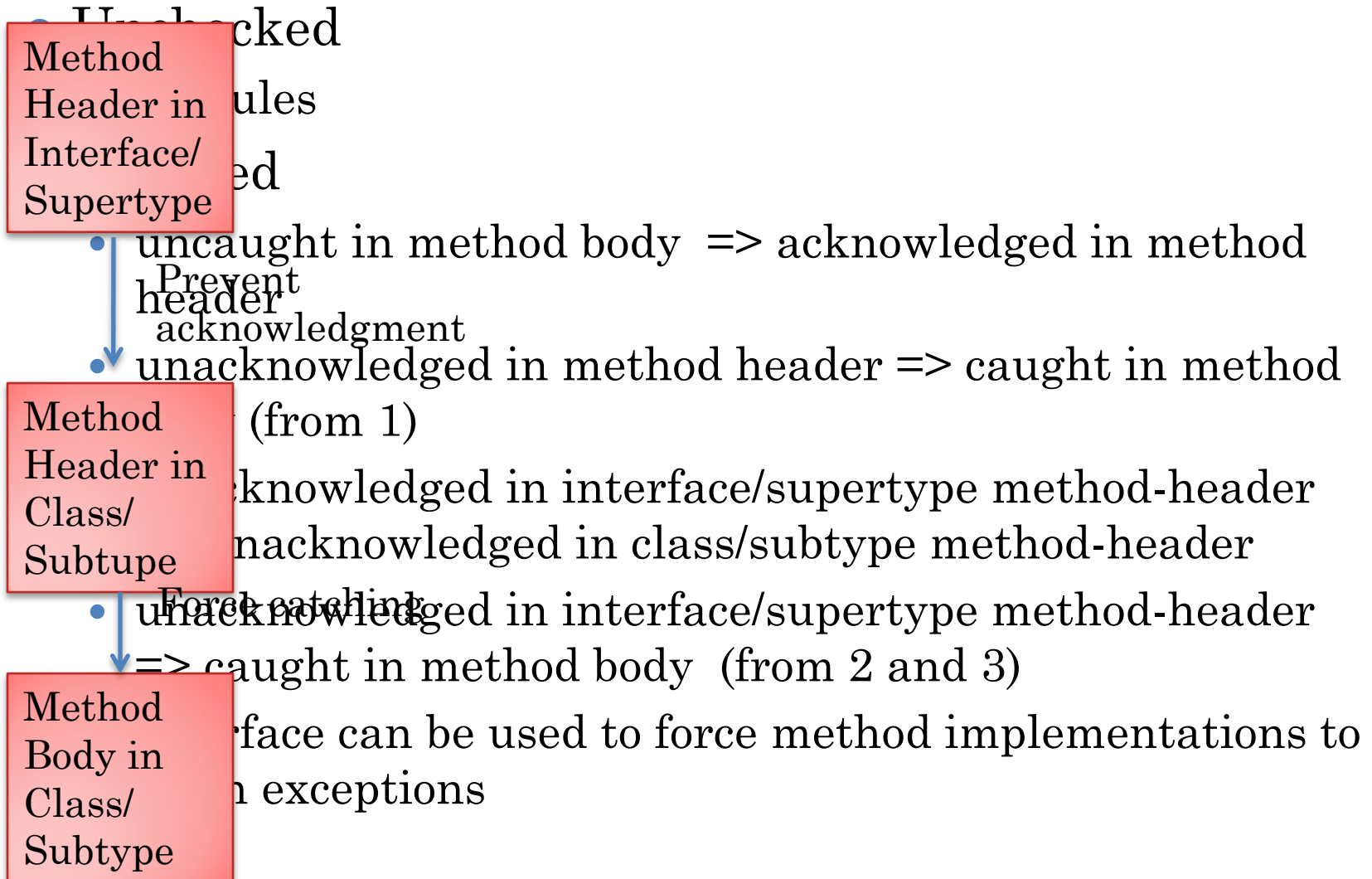
```
public class AnInputStreamScannerThrowingIOException
    implements StringIterator {
    BufferedReader input =
```

Unacknowledged in interface method-header → caught in class method body

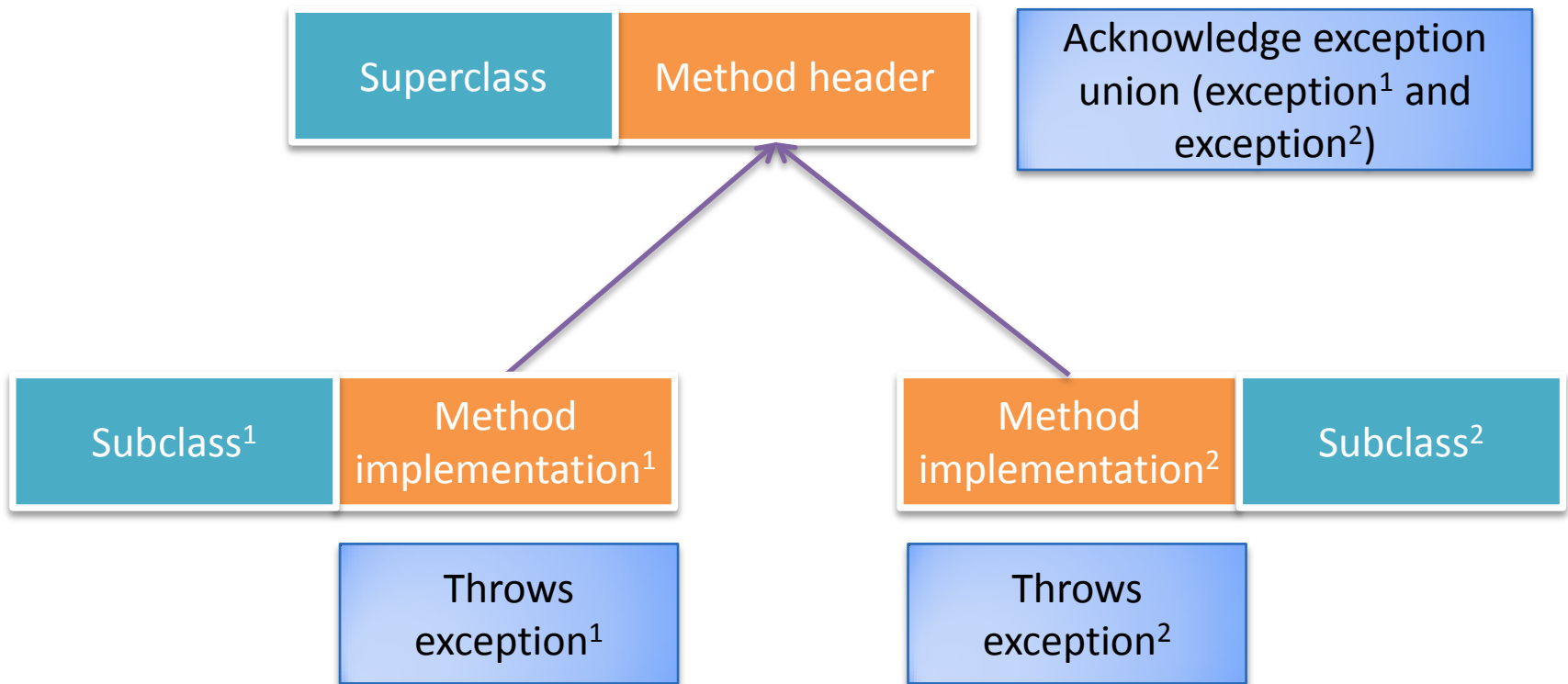
```
public AnInputStreamScannerThrowingIOException ()
    throws IOException {
    nextLine = input.readLine();
}
public boolean hasNext() {
    return nextLine.equals(QUIT_COMMAND);
}
public String next() throws IOException, NoSuchElementException {
    if (!hasNext()) throw new NoSuchElementException();
    String retVal = nextLine;
    try {
        nextLine = inputStream.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return retVal;
}
```



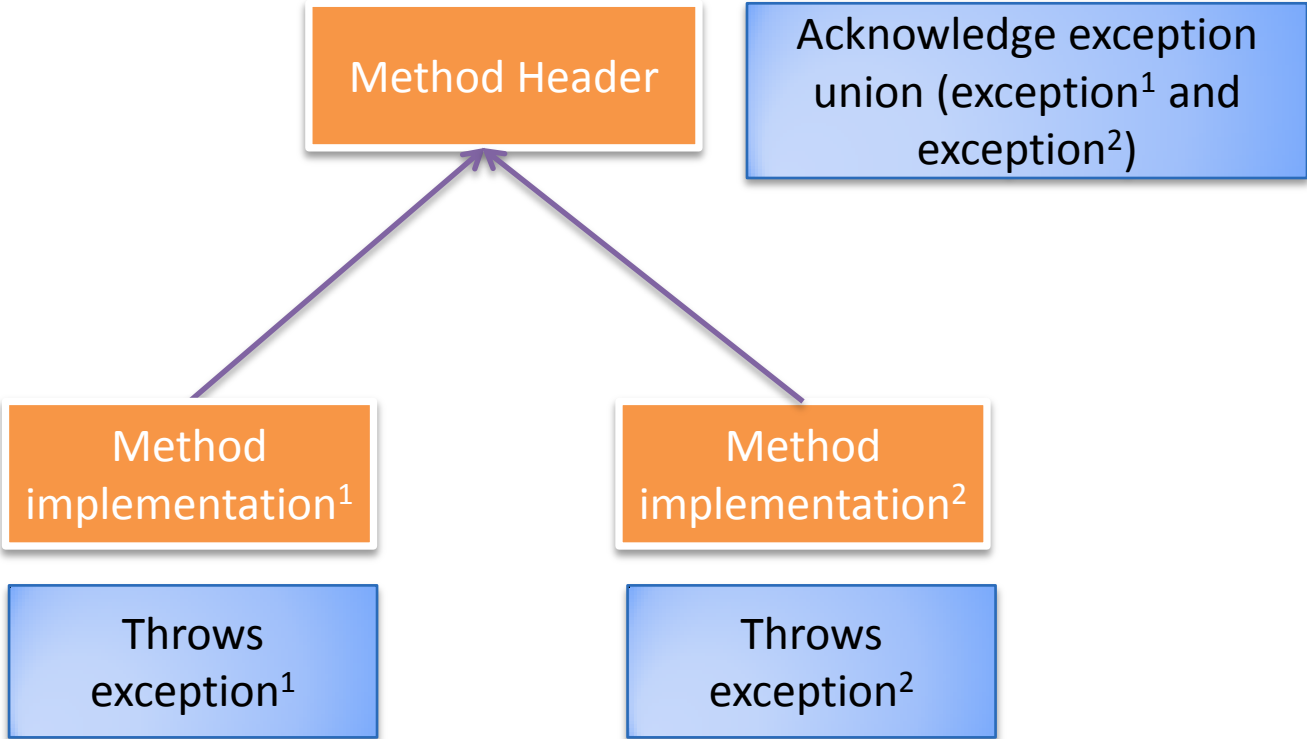
# CHECKED VS. UNCHECKED



# CLASS INSTANCE METHOD VARIATIONS



# INSTANCE METHOD VARIATIONS THROWING DIFFERENT EXCEPTIONS



# 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

Caller does not know what it  
must handle

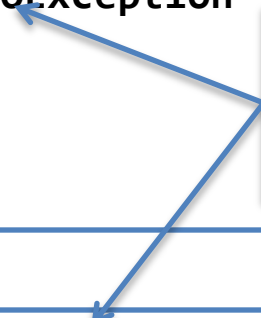
It may not have code to  
handle

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

# WHY NOT

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

ArrayIndexOutOfBoundsException  
not acknowledged



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

No complaints from Java for  
omitting throws clause

# EVEN MORE SEVERE FALSE POSITIVE

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