CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Exceptions and Assertions

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Announcements

- Midterm graded
  - to be handed out after class
  - solution on the web site after class

- HW6 due Wed
  - you may need to redo parts of HW5 for efficiency
  - (can keep same spec tests)

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - for things you believe will/should never happen
- Exceptions: what, how
  - how to throw, catch, and declare exceptions in Java
  - subtyping of exceptions
  - checked vs. unchecked exceptions
- Exceptions: why *in general*
  - for things you believe are bad and should rarely happen
  - and many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Not all “errors” should be failures

Some error cases:

1. Misuse of your code
   – e.g., precondition violation
   – **should** be a failure (i.e., made visible to the user)

2. Errors in your code
   – e.g., representation invariant fails to hold
   – **should** be a failure

3. Unexpected resource problems
   – e.g., missing file, server offline, …
   – **should not** be a failure (try to recover / hide from user)

What to do when failing

Fail fast and fail friendly

Goal 1: *Give information about the problem*

– a good error message is important for debugging
– failing quickly helps localize the defect

Goal 2: *Prevent harm*

– stop before anything worse happens
– perform cleanup: close open resources etc.

Errors that should be failures

A precondition prohibits misuse of your code

- weakens the spec by throwing out unhandled cases

This ducks the problem of errors-will-happen

- with enough clients, some will use your code incorrectly
- it often makes sense to check for these errors
- even though you don't specify what the behavior will be, it still makes sense to fail fast

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - for things you believe will/should never happen
- Exceptions: what, how
  - how to throw, catch, and declare exceptions *in Java*
  - subtyping of exceptions
  - checked vs. unchecked exceptions
- Exceptions: why *in general*
  - for things you believe are bad and should rarely happen
  - and many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Defensive programming

Assertions about your code:

– precondition, postcondition, representation invariant, etc.

Check these *statically* via reasoning and tools

Check these *dynamically* via assertions

```
assert index >= 0;
assert items != null : "null item list argument"
assert size % 2 == 0 : "Bad size for " +
                                 toString();
```

– write assertions as you write code

– include descriptive messages

Enabling assertions

In Java, assertions can be enabled or disabled at runtime
(no recompile is required)

Command line:

`java -ea` runs code with assertions enabled

`java` runs code with assertions disabled (default)

Eclipse:

Select Run > Run Configurations… then add `-ea` to VM arguments under (x)=arguments tab

Turn them off only in **rare** circumstances
(e.g., production code running on a client machine)

How *not* to use assertions

Don't **clutter** the code with useless assertions

```
x = y + 1;
assert x == y + 1;    // the compiler worked!
```

- Too many assertions can make the code hard to read
- Be judicious about where you include them. Good choices:
  - preconditions & postconditions
  - invariants of non-trivial loops
  - representation invariants after mutations

How *not* to use assertions

Don't perform side effects:

```
assert list.remove(x); // won't happen if disabled

// better:
boolean found = list.remove(x);
assert found;
```

`assert` and `checkRep()`

CSE 331's **`checkRep()`** is another dynamic check

Strategy: use **`assert`** in **`checkRep()`** to test and fail with meaningful message if trouble found

- CSE 331 tests will check that assertions are enabled

Easy to forget to enable them in your own projects

- Google doesn't use them for this reason

Expensive `checkRep()` tests

Detailed checks can be too slow in production

- especially if asymptotically slower than code being checked

But complex tests can be very helpful during testing & debugging (let the computer find problems for you!)

Suggested strategy for **`checkRep`**:

- create a static, global “debug” or “debugLevel” variable
- run expensive tests when this is enabled
- turn it off in graded / production code if tests are too expensive

Square root

```
// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
   ...
}
```

Square root with assertion

```
// requires: x >= 0
// returns: approximation to square root of x
public double sqrt(double x) {
  assert x >= 0.0;
  double result;
  … compute result …
  assert Math.abs(result*result – x) < .0001;
  return result;
}
```

- These two assertions serve different purposes

(Note: the Java library Math.sqrt method returns NaN for x<0. We use different specifications in this lecture as examples.)

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - for things you believe will/should never happen
- Exceptions: what, how
  - how to throw, catch, and declare exceptions *in Java*
  - subtyping of exceptions
  - checked vs. unchecked exceptions
- Exceptions: why *in general*
  - for things you believe are bad and should rarely happen
  - and many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Square root, specified for all inputs

```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
    throws IllegalArgumentException {
  if (x < 0)
    throw new IllegalArgumentException();
  …
}
```

- `throws` is part of a method signature: "it might happen"
  - comma-separated list
- `throw` is a statement that actually causes exception-throw
  - immediate control transfer [like `return` but different]

Using try-catch to handle exceptions

```
public double sqrt(double x)
    throws IllegalArgumentException
  …
```

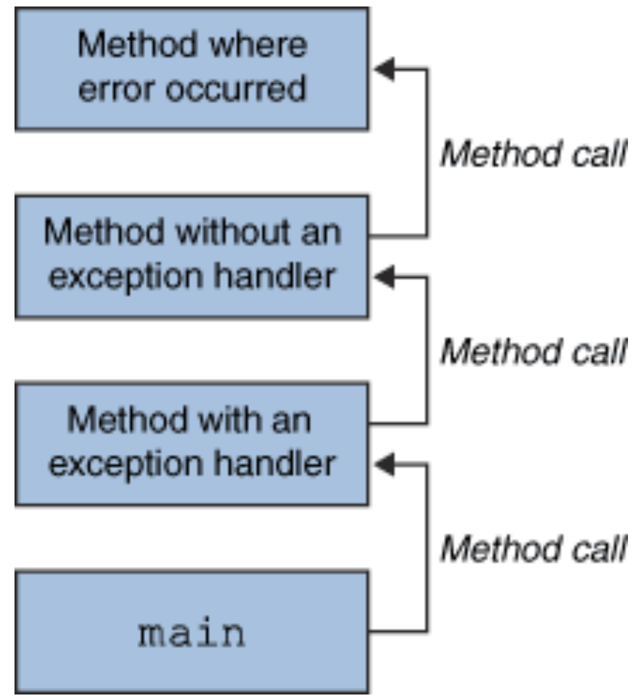
Client code:

```
try {
  y = sqrt(…);
} catch (IllegalArgumentException e) {
  e.printStackTrace(); // or other actions
}
```

- Handled by nearest *dynamically* enclosing **`try/catch`**
  - top-level default handler: print stack trace & crash

Throwing and catching

- Executing program has a stack of currently executing methods
  - dynamic: reflects runtime order of method calls
  - no relation to static nesting of classes, packages, etc.
- When an exception is thrown, control transfers to nearest method with a *matching* catch block
  - if none found, top-level handler used
- Exceptions allow *non-local* error handling
  - a method many levels up the stack can handle a deep error

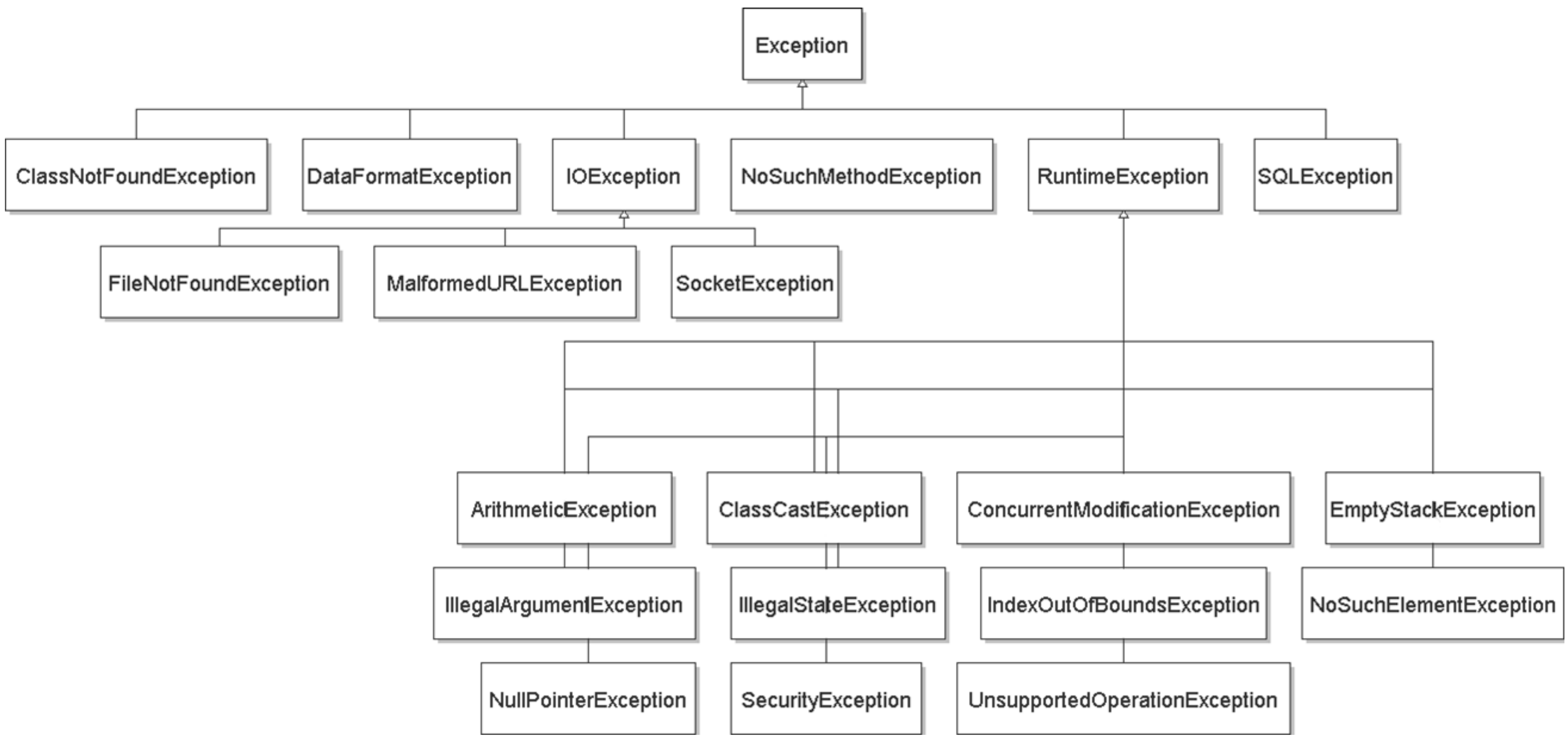


Catching with inheritance

```
try {
   code…
} catch (FileNotFoundException fnfe) {
   code to handle a file not found exception
} catch (IOException ioe) {
   code to handle any other I/O exception
} catch (Exception e) {
   code to handle any other exception
}
```

- A **`SocketException`** would match the second block
- An **`ArithmeticException`** would match the third block
- (Subsequent catch blocks need not be supertypes like this)

(Abridged) Exception Hierarchy



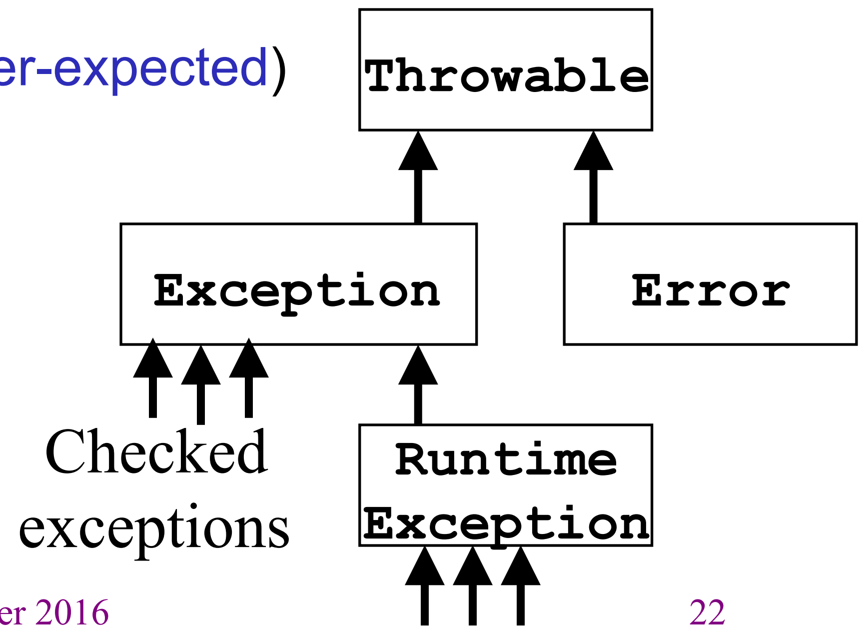
Java's checked/unchecked distinction

Checked exceptions (*style*: for *special cases / abnormal cases*)

- **callee** must declare in signature (else type error)
- **client** must either catch or declare (else type error)
  - even if you can prove it will never happen at run time, the type system does not "believe you"
- guaranteed to be a matching enclosing catch *at runtime*

Unchecked exceptions (*style*: for never-expected)

- **library** has no need to declare
- **client** has no need to catch
- these are subclasses of:
  - **`RuntimeException`**
  - **`Error`** (rarely caught)



Checked vs. unchecked

- No perfect answer to the question “should clients be forced to catch (or declare they throw) this exception?”
  - Java provided both options
- Advantages to checked exceptions:
  - Static checking of callee: only declared exceptions are thrown
  - Static checking of caller: exception is caught or declared
- Disadvantages:
  - impedes implementations and overrides (can’t add exceptions)
  - often in your way when prototyping
  - have to catch or declare even if the exception is not possible

The `finally` block

`finally` block is always executed

- whether an exception is thrown or not

```
try {
  …code…
} catch (Type name) {
  code to handle the exception
} finally {
  code to run after the try or catch finishes
}
```

What `finally` is for

`finally` is used for common “must-always-run” or “clean-up” code

- avoids duplicated code in catch branch[es] and after
- avoids having to catch all exceptions

```
try {
  // ... write to out;  might throw exception
} catch (IOException e) {
  System.out.println("Caught IOException: "
                     + e.getMessage());
} finally {
  out.close();
}
```

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - for things you believe will/should never happen
- Exceptions: what, how
  - how to throw, catch, and declare exceptions in Java
  - subtyping of exceptions
  - checked vs. unchecked exceptions
- Exceptions: why *in general*
  - for things you believe are bad and should rarely happen
  - and many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Propagating an exception

```
// returns: x such that ax^2 + bx + c = 0
// throws: IllegalArgumentException if no real soln exists
double solveQuad(double a, double b, double c)
    throws IllegalArgumentException {
  // No need to catch exception thrown by sqrt
  return (-b + sqrt(b*b - 4*a*c)) / (2*a);
}
```

Aside: should we call it “**illegal**” to give a quadratic with no real soln?

Why catch exceptions locally?

Problems:

1. Failure to catch exceptions often violates modularity
   - call chain: A `->` IntegerSet.insert `->` IntegerList.insert
   - IntegerList.insert throws some exception
     - implementer of IntegerSet.insert knows how list is being used
     - implementer of A may not even know that IntegerList exists
2. Possible that a method on the stack may think that it is handling an exception raised by a different call

Alternative: catch it and throw again

- "chaining" or "translation"
- do this even if the exception is better handled up a level
- makes it clear to reader of code that it was not an omission

Exception translation

```
// returns: x such that ax^2 + bx + c = 0
// throws: NotRealException if no real solution exists
double solveQuad(double a, double b, double c)
                                  throws NotRealException {
  try {
    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
  } catch (IllegalArgumentException e) {
    throw new NotRealException(); // "chaining"
  }
}

class NotRealException extends Exception {
  NotRealException() { super(); }
  NotRealException(String message) { super(message); }
  NotRealException(Throwable cause) { super(cause); }
  NotRealException(String msg, Throwable c) { super(msg, c); }
}
```

Exceptions as non-local control flow

```
void compile() {
  try {
    parse();
    typeCheck();
    optimize();
    generate():
  } catch (RuntimeException e) {
    Logger.log("Failed: " + e.getMessage());
  }
}
```

- Not common – usually bad style, particularly at small scale
- Java/C++, etc. exceptions are (a bit) expensive if thrown/caught
- Reserve exceptions for exceptional conditions

Two distinct uses of exceptions

- Errors that should be failures
  - unexpected (ideally, should not happen at all)
  - should be rare with high quality client and library
  - can be the client's fault or the library's
  - often unrecoverable

- Special results
  - expected, just not the common case
  - possibly unpredictable or unpreventable by client

Handling exceptions

- Errors that should be failures
  - usually can't recover
  - unchecked exceptions the better choice (avoids much work)
  - if condition not checked, exception propagates up the stack
    - top-level handler prints the stack trace

- Special results
  - take special action and continue computing
  - should always check for this condition
  - should handle locally by code that knows how to continue
  - checked exceptions the better choice

Don’t ignore exceptions

Effective Java Tip #65: Don't ignore exceptions

Empty catch block is poor style

```
try {
  readFile(filename);
} catch (IOException e) {}  // silent failure
```

At a minimum, print out the exception so you know it happened

- and exit if that’s appropriate for the application

```
} catch (IOException e) {
  e.printStackTrace();
  System.exit(1);
}
```

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - for things you believe will/should never happen
- Exceptions: what, how *in Java*
  - how to throw, catch, and declare exceptions
  - subtyping of exceptions
  - checked vs. unchecked exceptions
- Exceptions: why *in general*
  - for things you believe are bad and should rarely happen
  - and many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Informing the client of a problem

Special value:

- `null` for `Map.get`
- `-1` for `indexOf`
- `NaN` for `sqrt` of negative number

Advantages:

- can be less verbose than try/catch machinery

Disadvantages:

- error-prone: callers forget to check, forget spec, etc.
- need “extra” result: doesn’t work if every result could be real
    - example: if a map could store `null` keys
- has to be propagated manually one call at a time

General Java style advice: exceptions for exceptional conditions

- up for debate if `indexOf` not-present-value is exceptional
    - Python has two versions, one w/ exception and one w/out

Special values in C/C++/others

- For errors and exceptional conditions in Java, use exceptions!
- But C doesn't have exceptions and some C++ projects avoid them
- Over decades, a common idiom has emerged
  – error-prone but you can get used to it ☹
  – affects how you read code
  – put "results" in "out-parameters"
  – result indicates success or failure

```
type result;
if(!computeSomething(&result)) { … return 1; }
// no "exception", use result
```

- Bad, but less bad than error-code-in-global-variable

Outline

- General concepts about dealing with errors and failures
- Assertions: what, why, how
  - For things you believe will/should never happen
- Exceptions: what, how *in Java*
  - How to throw, catch, and declare exceptions
  - Subtyping of exceptions
  - Checked vs. unchecked exceptions
- Exceptions: why *in general*
  - For things you believe are bad and should rarely happen
  - And many other style issues
- Alternative with trade-offs: Returning special values
- Summary and review

Exceptions: review

Use an exception when
- used in a broad or unpredictable context (client can't predict)
- checking the condition is feasible

Use a precondition when
- checking would be prohibitive
  - e.g., requiring that a list be sorted
- used in a narrow context in which calls can be checked

Use a special value when
- it is a reasonable common case
- clients are likely (?) to remember to check for it

Use an assertion for internal consistency checks that should not fail

Exceptions: review, continued

Use *checked* exceptions most of the time

- static checking is helpful! (**tools**, inspection, & testing)

Avoid checked exceptions if there is probably no way to recover

Handle exceptions sooner rather than later

Not all exceptions are errors (just special cases)

- example: file not found

Good reference: Effective Java, Chapter 9

- a whole chapter? Exception-handling design matters!