CSE 331
Software Design and Implementation

Lecture 12
Assertions & Exceptions

Leah Perlmutter / Summer 2018

Announcements

Announcements

- Happy Friday!

- Reading 5 and Quiz 5 to be released today, due Thursday
- HW6 due Thursday
- Midterm to be graded this weekend

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

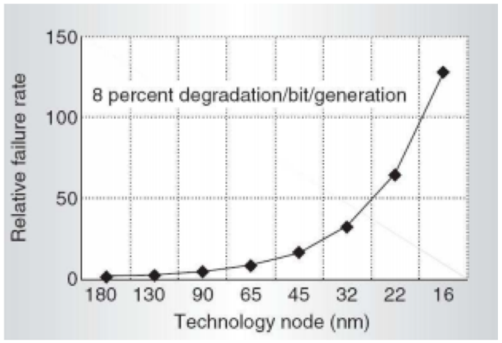
Motivation

Failure happens!

In 2000 there were reports that transient faults caused crashes at a number of Sun's major customer sites, including America Online and eBay. Later, Hewlett Packard admitted multiple problems in the Los Alamos Labs supercomputers due to transient faults. Finally, Cypress Semiconductor has confirmed "The wake-up call came in the end of 2001 with a major customer reporting havoc at a large telephone company. Technically, it was found that a single soft fail. . . was causing an interleaved system farm to crash".

Fault-tolerant Typed Assembly Language

-- Walker et al.



Failure causes

Partial failure is inevitable

- Goal: prevent complete failure
- Structure your code to be reliable and understandable

Some failure causes:

1. Misuse of your code
   - Precondition violation
2. Errors in your code
   - Bugs, representation exposure, …
3. Unpredictable external problems
   - Out of memory, missing file, …

What to do when something goes wrong

Fail early, fail friendly

Goal 1: *Give information about the problem*

- To the programmer – a good error message is key!
- To the client code: via exception or return-value or …

Goal 2: *Prevent harm*

Abort: inform a human
  - Perform cleanup actions, log the error, etc.

Re-try:
  - Problem might be transient

Skip a subcomputation:
  - Permit rest of program to continue

Fix the problem?
  - *Usually* infeasible to repair from an unexpected state

Avoiding errors

A precondition prohibits misuse of your code

- Adding a precondition weakens the spec

This ducks the problem of errors-will-happen

- Mistakes in your own code
- Misuse of your code by others

Removing a precondition requires specifying more behavior

- Often a good thing, but there are tradeoffs
- Strengthens the spec
- Example: specify that an exception is thrown

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

Assertions

Defensive programming

Check:

- Precondition
- Postcondition
- Representation invariant
- Other properties that you know to be true

Check *statically* via reasoning and tools

Check *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 without recompiling

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

(These tool details were covered in section already)

When *not* to use assertions

Don’t clutter the code with useless, distracting repetition

```
x = y + 1;
assert x == y + 1;
```

Don’t perform side effects

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

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

Turn them off in rare circumstances (production code(?))

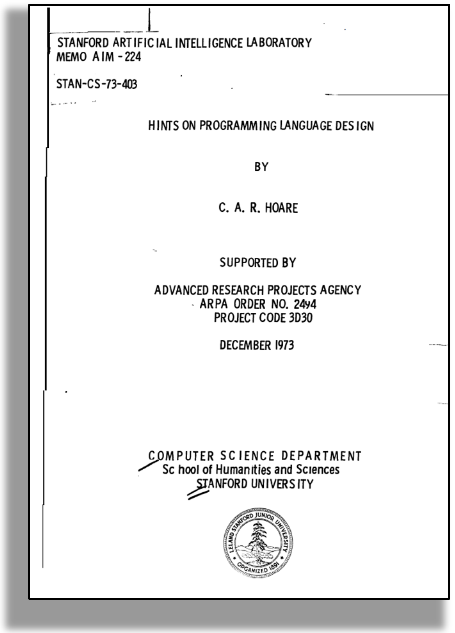
– Most assertions better left enabled

Don’t go to sea without your lifejacket!

Finally, it is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea?

Hints on Programming Language Design

-- C.A.R. Hoare



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-224

STAN-CS-73-403

HINTS ON PROGRAMMING LANGUAGE DESIGN

BY

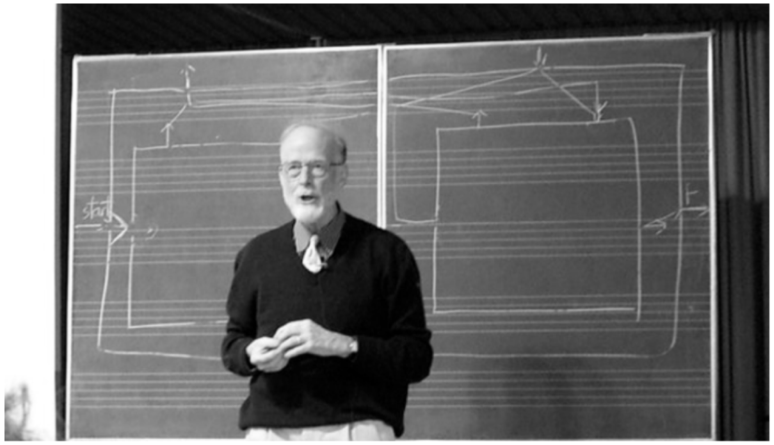
C. A. R. HOARE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494
PROJECT CODE 3D30

DECEMBER 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



`assert` and `checkRep()`

CSE 331's **`checkRep()`** is an example of a dynamic check

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

- Be sure to enable asserts when you do this!

Asserts should be enabled always for CSE 331 projects

- We will enable them for grading

Expensive `checkRep()` tests

Detailed checks can be too slow in production

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

No perfect answers; 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

Often helpful: put expensive / complex tests in separate methods and call as needed

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 very different purposes

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

Side note: The other kind of assertion

Junit assertions (for unit testing) are different from Java assertions

- Junit assertions were covered in section, not discussed here

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

Exceptions: what, how

Square root, specified for all inputs

```
/**   [...]
 * @throws IllegalArgumentException if x < 0
 * @return approximation to square root of x   */
public double sqrt(double x)
      throws IllegalArgumentException  {
  if (x < 0)
    String msg = "sqrt needs positive x, got " + x;
    throw new IllegalArgumentException(msg);
  ...
}
```

- **`throws`** keyword used in method signature: “it might happen”
  - Declares comma-separated list of exception types
- **`throw`** is a statement that actually causes exception-throw
  - Immediate control transfer [like **`return`** but different]
- **`@throws`** is a tag in the Javadoc that documents the circumstances when each declared exception could be thrown

Using try-catch to handle exceptions

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

Client code:

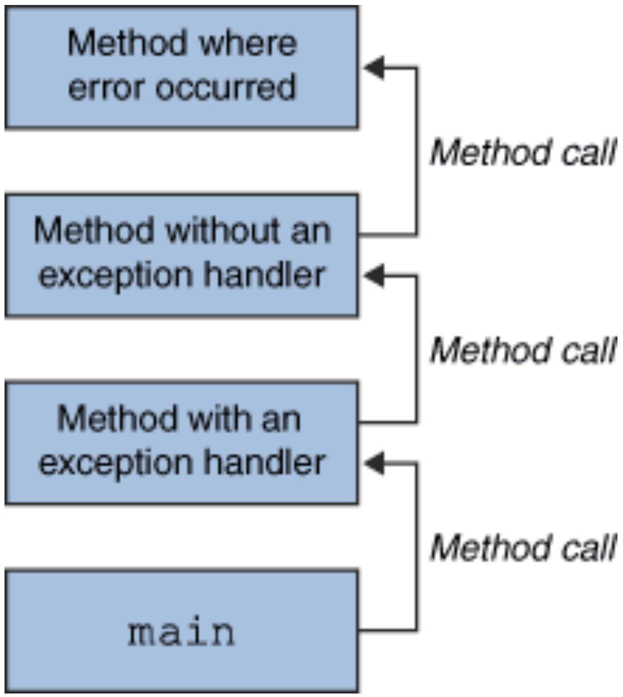
```
try {
  y = sqrt(…);
} catch (IllegalArgumentException e) {
  e.printStackTrace(); //and/or take other actions
}
```

Handled by nearest *dynamically* enclosing **`try/catch`**

– Top-level default handler: stack trace, program terminates

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 prints stack trace and terminates
- Exceptions allow *non-local* error handling
  - A method many levels up the stack can handle a deep error


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Exception-handling mechanisms in modern programming languages help software developers build robust applications by separating the normal control flow of a program from the control flow of the program under exceptional situations. ... One consequence is that developers wishing to improve the robustness of a program must figure out which exceptions, if any, can flow to a point in the program. ***Unfortunately, in large programs, this exceptional control flow can be difficult, if not impossible, to determine.***

Gail C Murphy

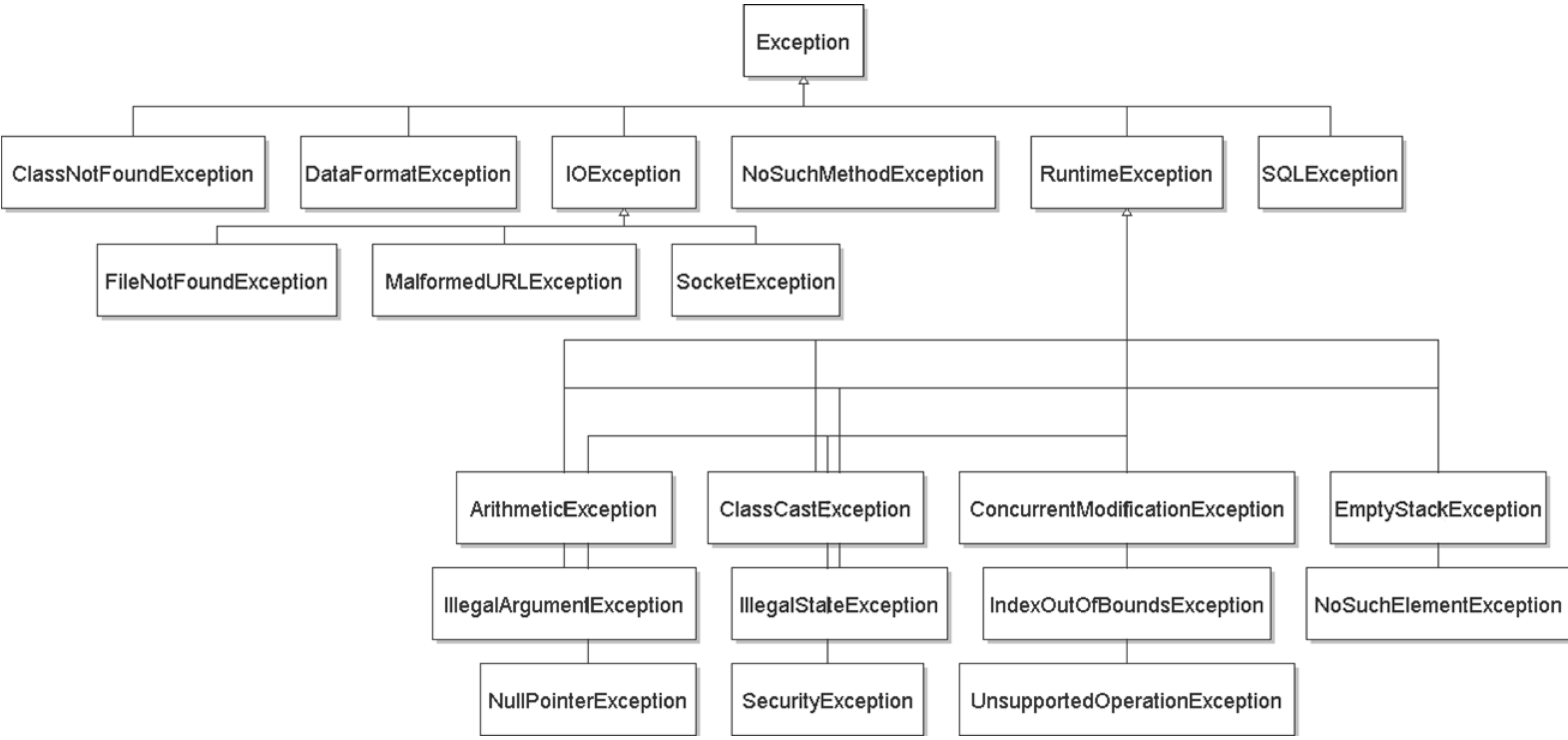
Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems

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

Exception Hierarchy



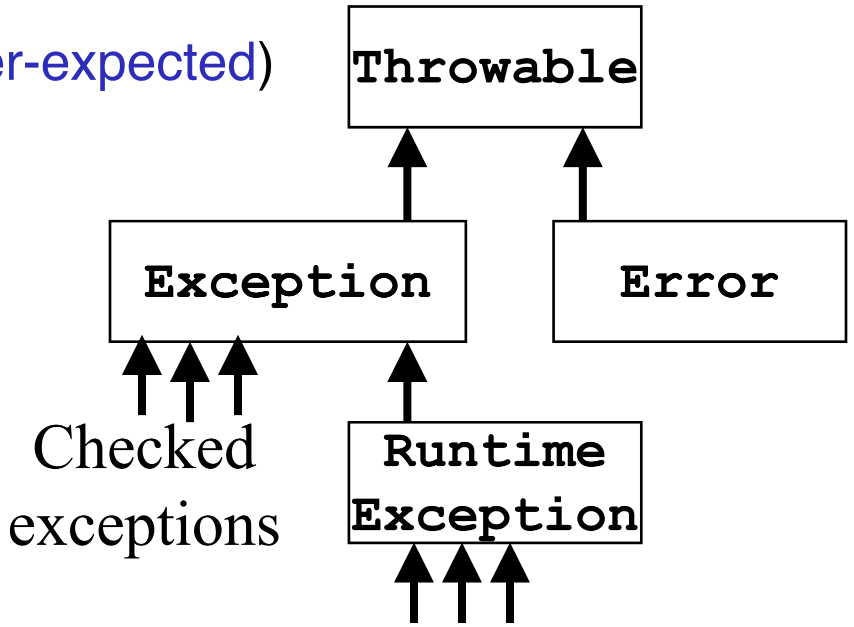
Java's checked/unchecked distinction

Checked exceptions (*style*: for *special 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"
- There is guaranteed to be a dynamically enclosing catch

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

- Library: No need to declare
- Client: No *need* to catch
- Subclasses of **`RuntimeException`** and **`Error`**



Checked vs. unchecked

- No perfect answer to “should possible exceptions thrown” be part of a method signature
  - So Java provided both
- Advantages to checked exceptions:
  - Static checking of callee ensures no other checked exceptions get thrown
  - Static checking of caller ensures caller does not forget to check
- Disadvantages:
  - Impedes implementations and overrides
  - Often in your way when prototyping
  - Have to catch or declare even in clients where 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/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 *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: when, why

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: How can clients know if a set of arguments
to **`solveQuad`** is illegal?

Why catch exceptions locally?

Failure to catch exceptions usually violates modularity

- Call chain: A $\rightarrow$ IntegerSet.insert $\rightarrow$ 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

Method on the stack may think that it is handling an exception raised by a different call

Better alternative: catch it and throw again

- “chaining” or “translation”
- Restate in a level of abstraction that the client can understand
- 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 {
    double discriminant = b*b - 4*a*c;
    return (-b + sqrt(discriminant)) / (2*a);
  } catch (IllegalArgumentException e) {
    String msg = "need positive b^2-4ac, got " + discriminant;
    throw new NotRealException(msg, e); // chaining
  }
} ...
class NotRealException extends Exception {
  // implement constructor alanogous to each of super's
  NotRealException() { super(); }
  NotRealException(String message) { super(message); }
  NotRealException(Throwable cause) { super(cause); }
  NotRealException(String msg, Throwable c) { super(msg, c); }
}
```

~~Not a Real Exception~~ Exception thrown when not a real number

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


public static void main(String[] args) {
  ...
  try {
    double root = solveQuad(a, b, c);
  } catch (NotRealException nre) {
    nre.printStackTrace(); // figure out what happened
    // doing a deep dive
    Throwable cause = nre.getCause();
    cause.printStackTrace();
  }
  ...
}
```

Two distinct uses of exceptions

- Failures
  - Unexpected (this will never happen)
  - Should be rare with well-written client and library
  - Can be the client's fault or the library's
  - Usually unrecoverable

- Special results
  - Expected but not the common case
  - Unpredictable or unpreventable by client

Handling exceptions

- Failures
  - Usually can’t recover
  - If condition not checked, exception propagates up the stack
  - The top-level handler prints the stack trace
  - Unchecked exceptions the better choice for failures (else many methods have to declare they could throw it)

- 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 for special results (encourages local handling)

Don’t ignore exceptions

Effective Java Tip #65: Don't ignore exceptions

Empty catch block is (common) poor style – often done to get code to compile despite checked exceptions

- Worse reason: to silently hide an error

```
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

Alternative: Sentinel Values

Informing the client of a problem

Special value:

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

Advantages:

- For a normal-ish, common case, it "is" the result
- Less verbose clients 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

Aside: Special values in C/C++ (Don’t do this in Java!)

- 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
  - Put “results” in “out-parameters”
  - Return value indicates success or failure

```
double result; // different from return value
if(!compute(&result)) { // return 0 (false) on success
...
} else {  // failure (nonzero return value)
...
}
```

- 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

Some Tips

- Use *checked* exceptions most of the time
  - Static checking is helpful

- Handle exceptions sooner rather than later

- Good reference: Effective Java, Chapter 9
  - A whole chapter? Exception-handling design matters!

Exceptions and Assertions: Summary

Unchecked exception

- throw in situations we never expect to happen
- e.g. programming errors, violated preconditions
- does not have to be caught or declared in method header

Checked exception

- throw in edge cases we expect might happen
- must be caught or declared in method header

Sentinel value

- sometimes used in place of checked exception
- caveat: clients might forget to check for it

Assertion

- use for internal consistency checks that should not fail

Announcements

Announcements

- Happy Friday!

- Reading 5 and Quiz 5 to be released today, due Thursday
- HW6 due Thursday
- Midterm to be graded this weekend

- Thanks for coming to class and enjoy your weekend!