CSE 331
Software Design & Implementation

Dan Grossman
Winter 2014
Exceptions and Assertions
(Based on slides by Mike Ernst, David Notkin, Hal Perkins)
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
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
Defensive programming

Check:
– Precondition
– Postcondition
– Representation invariant
– Other properties that you know to be true

Check \textit{statically} via reasoning and tools

Check \textit{dynamically} via \texttt{assertions}

\begin{verbatim}
assert index >= 0;
assert items != null : "null item list argument"
assert size % 2 == 0 : "Bad size for " + toString();
\end{verbatim}

– 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

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

Don’t perform side effects

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

// Better:

```java
boolean found = list.remove(x);
assert found;
```

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

– Most assertions better left enabled
assert and checkRep()

CSE 331’s checkRep() is another 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
// requires: x \geq 0
// returns: approximation to square root of x
public double sqrt(double x) {
    ...
}

Square root
Square root with assertion

// requires: x \geq 0
// returns: approximation to square root of x
public double sqrt(double x) {
    assert (x \geq 0.0);
    double result;
    ... compute result ...
    assert (Math.abs(result*result - x) < .0001);
    return result;
}

- These two assertions serve very different purposes
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
// 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

```java
public double sqrt(double x) throws IllegalArgumentException {
    ...
}
```

Client code:
```java
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
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
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 callee ensures no other checked exceptions get thrown
  – Static checking 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

```java
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

```java
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
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 → 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

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

Better alternative: catch it and throw it 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(Throwerable cause) { super(cause); }
    NotRealException(String msg, Throwable c) { super(msg, c); }
}
Exceptions as non-local control flow

```java
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 expensive if thrown/caught
– Reserve exceptions for exceptional conditions
Two distinct uses of exceptions

• Failures
  – Unexpected
  – 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 (else many methods have to declare they 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 (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

```java
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

```java
} 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:
- 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
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 is a boolean (int in C) to indicate success or failure

```c
  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
- 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-ish situation
- Clients are likely (?) to remember to check for it

Use an **assertion** for internal consistency checks that should not fail
Use *checked* exceptions most of the time
   – Static checking is helpful

But maybe avoid checked exceptions if possible for many callers to *guarantee* exception cannot occur

Handle exceptions sooner rather than later

Not all exceptions are errors
   – Example: File not found

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