Exceptions and assertions

CSE 331
University of Washington

Michael Ernst
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
   Memory corruption

Using the above categorization, how would you categorize these?
   – Failure of a subcomponent
   – No return value (e.g., list element not found, division by zero)
What to do when something goes wrong

Fail early, fail friendly

Goal 1: Give information about the problem
  To the programmer
  To the client code and/or human user

Goal 2: Prevent harm from occurring
  Abort: halt/crash the program
    Prevent computation (continuing could be bad or good)
    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)
    External problem: no hope; just be informative
    Internal problem: if you can fix, you can prevent
Avoiding blame for failures

A precondition prohibits misuse of your code
   Adding a precondition weakens the spec

This ducks the problem
   Does not address errors in your own code
   Does not help others who are misusing your code

Removing the precondition requires specifying the behavior
   Strengthens the spec
   Example: specify that an exception is thrown
      “Partial spec” vs. “complete spec” (neither is better)
Defensive programming: prevent or detect errors

Check

- precondition
- postcondition
- representation invariant
- other properties that you know to be true

Check **statically** via reasoning and tools

Check **dynamically** at run time via assertions

```java
assert index >= 0;
assert size % 2 == 0 : "Odd size for " + toString();
```

Write the assertions as you write the code

Descriptive message is optional
Outline

⇒ Assertions

• Exceptions

• Designing with exceptions
When *not* to use assertions

Don’t clutter the code

```java
x = y + 1;
assert x == y + 1; // useless, distracting
```

Don’t perform side effects

```java
assert list.remove(x); // modifies behavior if disabled
// Better:
boolean found = list.remove(x);
assert found;
```

How can you test at run time whether assertions are enabled?

Why would you want to do this?
Disabling assertions

Most assertions are better left enabled
– Prevents downstream problems
– Early indication of trouble eases debugging
– The cost is worth it during testing and debugging!

“What would we think of a sailor who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea?”

The user controls whether Java assertions run

```
java  -ea  runs Java with assertions enabled
java  runs Java with assertions disabled (default 😞)
```

A reason to use an assertion library

Turn off expensive assertions in CPU-limited production runs
– Common approach: guard expensive assertions (maybe including `checkRep()`) by static variable `debug`
– Set `debug` to false in production / graded code
// requires: x ≥ 0
// returns: approximation to square root of x
public double sqrt(double x) {
    ...
}
}
What is the purpose of each assertion?
Outline

• Assertions

⇒ Exceptions

• Designing with exceptions
Square root, specified for all inputs

```java
// 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();
    ...
}
```

Throwing an exception causes immediate control transfer
Like `return` but different

True subtyping for `throws` clauses:
Subclass method throws fewer more specific exceptions
Compiler does not enforce true subtyping
Using try-catch to handle exceptions

public double sqrt(double x) throws IllegalArgumentException

A thrown exception is handled by the catch associated with the nearest *dynamically enclosing* try

**Client code:**
```java
try {
    field = sqrt(-1);
} catch (IllegalArgumentException e) {
    ... take some action ...
}
```

**Client code:**
```java
try {
    field = sqrt(-1);
    foo();
} catch (IllegalArgumentException e) {
    ... take some action ...
}

void foo() {
    field = sqrt(-1);
}
```

Top-level default handler around `main()`: stack trace, program terminates
Throwing and catching

- At run time, Java maintains a call stack of methods that are currently executing
  - Dynamic from method calls during execution
  - Has no relation to static nesting of classes, packages, etc.
- When an exception is thrown, control transfers to the nearest method with a matching (= supertype) catch block
  - If none is found, top-level handler
    - Print stack trace, terminate program
- Exceptions allow non-local error handling
  - A method many levels up the stack can handle a deep error
The first matching `catch` clause executes

```java
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
}
```

- e.g., `SocketException`
- e.g., `ArithmeticException`
The finally block

finally body is always executed

Whether an exception is thrown or not

If an exception was thrown, the exception continues being thrown after the finally block executes

Useful for “clean-up” code, re-establishing invariants, ...

FileWriter out = null;
try {
    out = new FileWriter(...);
    ... write to out; may throw IOException
} finally {
    if (out != null) {
        out.close();
    }
}
Calling a method that might throw an exception

```java
public double sqrt(double x) throws IllegalArgumentException;

// returns: x such that ax^2 + bx + c = 0

double solveQuad(double a, double b, double c) {
    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
}
```

The compiler rejects this code. How can we fix it?
Declaring an exception

public double sqrt(double x) throws IllegalArgumentException;

// 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);
}

Uninformative to clients:

   solveQuad(1,0,1) ⇒ “-4 is less than zero”
Why handle exceptions locally?

Failure to catch exceptions may violate modularity

Call chain: \[ A \rightarrow \text{IntegerSet.insert} \rightarrow \text{IntegerList.insert} \]

\text{IntegerList.insert} throws an exception

Implementer of \text{IntegerSet.insert} knows how list is being used
Implementer of \( A \) may not even know that \text{IntegerList} exists

Procedure 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”

Maybe 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
public double sqrt(double x) throws IllegalArgumentException;

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

Note: clients don’t know whether a set of arguments to solveQuad is legal or illegal
public double sqrt(double x) throws IllegalArgumentException;

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

Useful mostly for debugging
Note: clients don’t know whether a set of arguments to solveQuad is legal or illegal
Exceptions as non-local control

Execute \texttt{procElt} on \((x, y)\) pairs, until \texttt{procElt} returns true

... boolean \texttt{finished} = false; ... try {
  for (int \texttt{x} : \texttt{xIter}) {
    for (int \texttt{y} : \texttt{xIter}) {
      if (\texttt{procElt}(\texttt{x}, \texttt{y})) {
        \texttt{finished} = true;
        break; // \texttt{y} loop
      }
    }
    if (\texttt{finished}) {
      break; // \texttt{x} loop
    }
  }
  \texttt{if} (\texttt{finished}) {
    break; // \texttt{x} loop
  }
} ... rest of method

... try {
  for (int \texttt{x} : \texttt{xIter}) {
    for (int \texttt{y} : \texttt{yIter}) {
      if (\texttt{procElt}(\texttt{x}, \texttt{y})) {
        \texttt{throw new Finished()};
      }
    }
  }
  \texttt{catch} (\texttt{Finished} \texttt{f}) { 
    // nothing to do
  }
  ... rest of method
Exceptions as non-local control

Execute `procElt` on `(x, y)` pairs, until `procElt` returns true

```
void procMatrix() {
    for (int x : xIter) {
        for (int y : xIter) {
            if (procElt(x, y)) {
                return;
            }
        }
    }
}
```

Procedural abstraction can improve code structure. Also gives a name to logical chunks of code.

... 

boolean finished = false;
xloop:
for (int x : xIter) {
    for (int y : xIter) {
        if (procElt(x, y)) {
            break xloop;
        }
    }
}
... rest of method

Reserve exceptions for exceptional conditions
Outline

• Assertions
• Exceptions

⇒ Designing with exceptions
Informing the client of a problem

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

Problems with using a special value
  No special value may be available
  Error-prone: the programmer may forget to check result
    Causes wrong computation and more obscure failure later
  Verbose – handle at each call, up the stack
    A positive: Clients can omit handling if they prove the special value is impossible
    Less efficient

A better solution: exceptions
Types of exceptional outcomes.  
Is it expected? What can the client do?

Errors

Unexpected
  Can be the client’s fault or the library’s
  Should be rare with well-written client and library

Usually unrecoverable

Special cases

Expected – client knows it is a possibility

  Unpredictable or unpreventable by client
    If client knows the result, no need to make the call
  Not easy to prevent/ignore with a precondition

Client can and *should* do something about it
Handling exceptions

Errors
Client usually can’t recover
Exception propagates to callees

Special cases
Take special action and continue computing
Client should always check for this condition
Client should handle locally
Java exceptions for errors and for special cases

Unchecked exceptions for errors

Library: no need to declare
Client: no need to catch
*RuntimeException*, *Error*, and their subclasses

Checked exceptions for special cases

Library: must declare in signature (compiler-enforced)
Client: must either catch or declare (compiler-enforced)
Even if you can prove it will never happen at run time
There is guaranteed to be a dynamically enclosing catch

Diagram:
- `Throwable`
- `Exception`
- `Error`
- `RuntimeException`
Checked vs. unchecked exceptions

Unchecked exceptions for errors
Use if (some) clients can ensure the exception will not happen
It would be verbose & irritating if clients had to write a catch block nonetheless

Checked exception for special cases
Static (compiler) checking ensures the caller handles it – can’t forget
Prevents program defects
Annoying while prototyping
Can’t omit handling even if you know it cannot happen
Checked exceptions have a lot of haters

If a library may throw a checked exception, the client *must* have a catch or throws clause

– Prevents program defects
– Can’t omit handling even if you know it cannot happen
– Annoying while prototyping

My take: Good idea, poor implementation

• Weird class hierarchy
• Unintuitive name “checked”
• Some classes are in wrong category

![Diagram of exception classes]

- Throwable
- Exception
- RuntimeException
- Error
- checked exceptions
- unchecked exceptions
Don’t ignore exceptions

• An empty catch block is poor style
  – often done to hide an error or get code to compile

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

• At minimum, print the exception so you know it happened

} catch (IOException e) {
    e.printStackTrace(); // be informative
    System.exit(1); // exit if appropriate
}
Exceptions and specifications

Use an **exception** (complete specification) when
- Used in a broad or unpredictable context
- Checking the condition in the library is feasible

Use a **precondition** (partial specification) when
- Checking in the library would be prohibitive
  - E.g., requiring that a list be sorted
- Used in a narrow context in which calls can be checked

Avoid preconditions in public APIs because
- Caller may violate precondition
- Program can fail in an uninformative or dangerous way
Exceptions in review

Use checked exceptions most of the time
  Static checking is useful
Use unchecked exceptions if
  – callers can guarantee the exception cannot occur, or
  – callers can’t do anything about it
Not all exceptions are due to program defects
  Example: File not found
  A program structuring mechanism with non-local jumps
  Used for exceptional (unpredictable) circumstances

Make implementation fail as early as possible
Handle exceptions sooner rather than later
Also see Bloch’s *Effective Java*, chapter 9