Exceptions and assertions

CSE 331

Spring 2010

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, many more
3. Unpredictable external problems
   - Out of memory
   - Missing file
   - Memory corruption

How would you categorize these?
- Failure of a subcomponent
- No return value (e.g., list element not found, division by zero)

Avoiding errors

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

Defensive programming

Check

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

Check statically via reasoning

Check dynamically at run time via assertions

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

Write the assertions as you write the code

When *not* to use assertions

Don't clutter the code

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

Don't perform side effects

```
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?

Turn them off in rare circumstances (e.g., production code)

- "**java -ea**" runs Java with assertions enabled
- "**java**" runs Java with assertions disabled (default)
- Most assertions should always be enabled

What to do when something goes wrong

- Something goes wrong: an assertion fails
  - Or if an assertion had been there, it would have failed
- Goal 1: Give information about the problem
  - To the programmer
    - A good error message is key!
  - To the client code
- Goal 2: Prevent harm from occurring
  - 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)
    - External problem: no hope; just be informative
    - Internal problem: if you can fix, you can prevent

Square root without exceptions

```
// 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) {
  double result;
  // compute result
  ...
  assert (Math.abs(result*result – x) < .0001);
  return result;
}
```

Square root with exceptions (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();
  ...
}
```

Client code:

```
try {
  y = sqrt(-1);
} catch (IllegalArgumentException e) {
  e.printStackTrace(); // or take some other action
}
```

Caught by catch associated with nearest dynamically enclosing try

Top-level default handler: stack trace, program terminates

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);
}
```

How can clients know whether a set of arguments to solveQuad is illegal?

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();
  }
}

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

Exception chaining:

```
throw new NotRealException(e);
```

Exceptions as non-local control flow

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

Informing the client of a problem

- Special value
  - **`null`** – Map.get
  - **`-1`** – indexOf
  - **`NaN`** – sqrt of negative number
- Problems with using special value
  - Hard to distinguish from real results
  - Error-prone: what if the programmer forgets to check result?
    - The value should not be legal – should cause a failure later
  - Ugly
  - Less efficient
- A better solution: exceptions

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
- Unpredictable or unpreventable by client

Handling exceptions

- Failures
  - Usually can’t recover
  - If the condition is not checked, the exception propagates up the stack
  - The top-level handler prints the stack trace
- Special results
  - Take special action and continue computing
  - Should always check for this condition
  - Should handle locally

Why catch exceptions locally?

- Failure to catch exceptions violates modularity
  - Call chain: A $\rightarrow$ IntegerSet.insert $\rightarrow$ IntegerList.insert
  - IntegerList.insert throws an exception
    - Implementer of IntegerSet.insert knows how list is being used
    - Implementor of A may not even know that IntegerList exists
- Procedure on the stack may think that it is handling an exception raised by a different call
- Even if exception is better handled up a level
  - May be better to catch it and throw it again ("chaining" or "translation")
  - Makes it clear to reader of code that it was not an omission

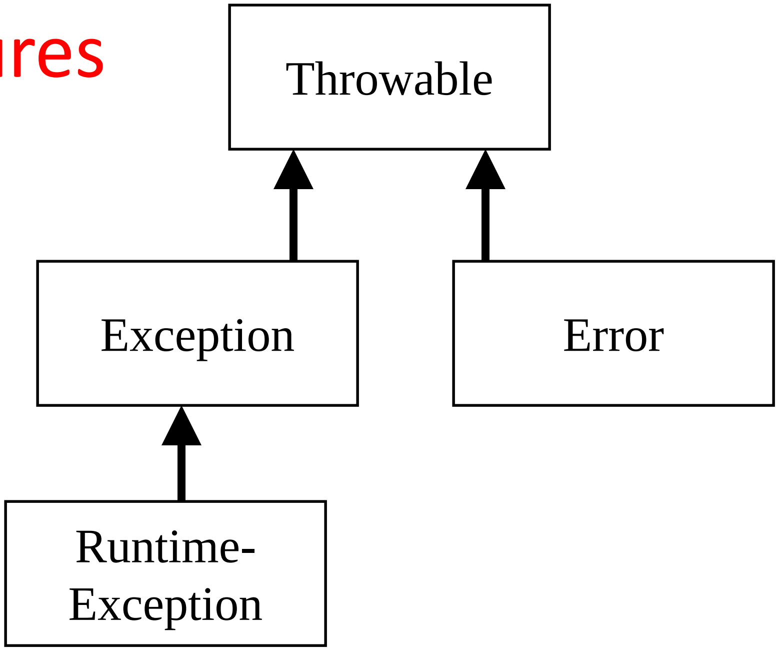
Java exceptions can be used for failures & special cases

Checked exceptions for special cases

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

Unchecked exceptions for failures

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



Avoid proliferation of checked exceptions

- Unchecked exceptions are better if clients will usually write code that ensures the exception will not happen
  - There is a convenient and inexpensive way to avoid it
  - The exception reflects unanticipatable failures
- Otherwise use a checked exception
  - Checked exceptions should be locally caught and handled
  - Checked exceptions that propagate long distances suggests bad design (failure of modularity)
- Java sometimes uses null (or NaN, etc.) as a special value
  - Acceptable if used judiciously, carefully specified
  - Easy to forget to check

Exceptions in 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
- Avoid preconditions because
  - Caller may violate precondition
  - Program can fail in uninformative or dangerous way
  - Want program to fail as early as possible
- How do preconditions and exceptions differ, for the client?

Exceptions in review, continued

- Use checked exceptions most of the time
- Handle exceptions sooner rather than later
- Not all exceptions are errors
  - A program structuring mechanism with non-local jumps
  - Used for exceptional (unpredictable) circumstances
- Also see Bloch's *Effective Java*