CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2022
Exceptions and Assertions
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 vs reasoning
   - e.g., representation invariant fails to hold
   - **should** be a failure

3. Unexpected resource problems
   - e.g., missing file, server offline, …
   - not an error in the sense above (... these are not bugs)
   - **should not** be a failure (i.e., do try to recover)
What to do when failing

Fail fast and fail friendly

Goal 1: Prevent harm
- stop before anything worse happens
- (do still need to perform cleanup: close open resources etc.)

Goal 2: Give information about the problem
- failing quickly helps localize the defect
- a good error message is important for debugging
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, someone will use your code incorrectly

Practice *defensive programming*:
   – usually 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();
  ```
  – throws AssertionError if condition is false
  – includes 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

\[
x = y + 1;
\]
\[
\text{assert } x == y + 1; \quad \text{// the compiler worked!}
\]

• 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:

```java
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 didn’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 on during unit tests
    • can use JUnit’s `@Before` for this
Square root

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

CSE 331 Spring 2022
Square root with assertion

```java
// 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.)