COMP 401 Fall 2017

Recitation 10: Assertions

Assertions

- Executable, runtime checks of program state
- Semantics: *“Assert (Condition): <Message>”*
  - Evaluate Condition → boolean
  - If !Condition, generate an Error (halt the program)
    - If Message is set, include Message in the error thusly generated
  - **NOT part of program logic:**
    - If an assertion fails, the program halts (logically, no recovery)
    - Does not obviate the need for other validation logic – specifically, the assertions, among other things, help us verify that such validation is correct.
- Motivations:
  - Ensuring program correctness, safety
  - Debugging/validation
  - Self-documentation

Requisite and Desirable Properties

- **Program correctness must not depend on assertions being enabled!**
- The predicate asserted (Condition) must be executable
- The predicate is a statement that evaluates to a boolean value, e.g.,

```
0 <= j && j < array.length
null != arg1 || this.list.size() > 0
withdrawlAmount > 0 &&
      balance-withdrawlAmount >= MIN_BALANCE
```

- Compact syntax – clear and brief, minimal clutter
- Conditionally-evaluated, e.g., based on runtime settings

Assertions in Java

- Added to the language in Java 1.4
- Enabled by JVM option "`-ea`" or "`-ea <package>`"; disabled with "`-da`" or "`-da <package>`"
- Syntax:

```
assert condition;
assert condition : message;
```

- Where:
  - `condition` is a `boolean` expression
  - `message` is a `String` expression
- Java semantics: if assertions are enabled and an assertion is executed with a predicate that evaluates to false, an `AssertionError` is thrown. If message is set, the exception's message includes `message`.

AssertionDemo

class `BankAccount`

Brief introduction to Pre/Postconditions

- Preconditions and postconditions are predicates (boolean expressions)
- As defined by Tony Hoare in the 1970's, for program state *S*, and code *C*, and pre/postconditions *Pre(S)* and *Post(S)* consider:

  Pre(S)
  Evaluate(C)
  Post(S)

- We can say (loosely) that *C* is "correct" if and only if
  Pre(S), Evaluate(C) → Post(S)
- I.e, if *Pre(S)* is true and we run the code *C*, then *Post(S)* will be true after running *C*
- E.g. (pseudocode); assume some number *x*

  y ← 0

  Pre(x) : x >= 0

  //Code
  y ← sqrt(x)

  *Post(x) : y >= 0 && (y * y == x)*

Pre/Postconditions continued

- Pre/Postconditions are often used to define the behavior of functions/methods
- These are often stated in the program documentation (e.g., JavaDoc) and, in practice, may or may not be checked at runtime
- We can interpret these as a contract whereby if the precondition is true when the method is called, then the postcondition will be true when the method returns
  - If the precondition is not true, the method's behavior may or may not be well-defined and/or the method may generate an error or throw an exception.

Invariants

- An invariant is a predicate that is always true before and after the execution of some code.
- A "class invariant" is a predicate that is always true before and after each method invocation on a class
  - The invariant may be false (transiently) during method invocation, e.g., as the internal state of the class changes
  - Ideally, the invariant will remain true even in the face of error conditions
- We can think of a class invariant as an implicit precondition and postcondition of all methods exposed by the class.
- There is also a notion of a "loop invariant" which is not explicitly covered in this course but which I strongly encourage you to read about.

Implementation of Predicates

- The full predicates of pre/postconditions and invariants may or may not be "implemented" as executable code
  - Often, important parts are executed as part of the program logic, e.g., to validate inputs
  - Many predicates would be computationally-expensive to verify at runtime (think about checking universal or existential quantification over a large collection)
- It may be useful to implement parts or all of some predicates as assertions to help with:
  - Documenting formal correctness of code
  - Debugging and/or testing
- Whether or not such predicates are implemented, it is helpful and recommended that they be written down in non-trivial situations (e.g., as comments or JavaDoc)
  - Writing down the predicates makes you think more carefully about what your code is trying to do
  - Future users and maintainers of your code, including you, will benefit from having the documentation.

Terminology and thinking about predicates

- Consider predicates *P(x)* and *Q(x)* where *x* is in the domain *X* of possible inputs to *P* and *Q*.
- Consider the sets
  $X_P = \{ x \in X : P(x) \}$
  $X_Q = \{ x \in X : Q(x) \}$
- We say that a predicate *P* is *stronger* than *Q* if and only if $X_P \subset X_Q$, i.e., the set of conditions under which *P* holds is strictly a subset of those where *Q* holds. This could also be stated as, "there exists some element $x' \in X_Q$ *s.t.* *P(x')=false*
- If *P* is stronger than *Q*, then *Q* is weaker than *P*.
- What is the strongest possible predicate?
- What is the weakest possible predicate?
- What if $X_P \not\subset X_Q$ and $X_Q \not\subset X_P$ (consider
  P(x) = (x % 2 == 0) and Q(x) = (x % 2 == 1))

Thinking about predicates

- Disjunction (OR)
  - Consider predicates *P(x) = A(x) and Q(x) = A(x) || B(x)*
  - In general, what can we say about the strength of P and Q?
- Conjunction (AND)
  - Consider predicates *P(x) = A(x) and R(x) = A(x) && B(x)*
  - In general, what can we say about the strength of *P* and *R*?

Strength of Pre and Postconditions

- Do we want strong or weak preconditions?
- Do we want strong or weak postconditions?

- In general, we would like to write the *weakest* precondition that implies the *strongest* postcondition

- If we think about invariants as being implicit pre and postconditions, then we want the weakest possible invariant that implies the strongest possible invariant… this leads to a predicate that is a *necessary and sufficient condition*

Threads

A very brief introduction

Very High-Level

- A thread is a context (stack and program counter) within a program executing some sequential code
- A program may comprise multiple threads that run asynchronously with respect to one another – multiple “threads” of execution within the program
- Threads logically run concurrently with their instructions interleaved arbitrarily
- In Java, every program has at least one thread
- Almost every application you use on your computer or mobile device is multi-threaded.

High Level – Think-abouts

- Why would we want multiple, asynchronous, concurrent execution contexts within our program?

- What sorts of design practices and patterns might this enable or enhance?

- Could multi-threading (potentially) lead to hazards?

YES

- Will we be discussing all of these hazards? **NO**. *Caveat emptor* (and take the Operating Systems class!)

Threads in Java

- `java.lang.Thread` is the base implementation of all threads in Java
- Among other idioms, you can execute your code under a separate thread by:
  - Extending `Thread` (and generally overriding run())
  - Or creating a (regular or anonymous) class that implements `Runnable`
- Implementing `Runnable` is typically preferred. (Why?)
- Given a `Runnable r`, you create and run a thread by:

```
Thread t = new Thread(r);
//Later on…
t.start();
```

PrinterThread/ThreadDemo