Benjamin Fenster
CS-100 (Damon)
1 March 2003

Notes – Invariants

I. Invarients
  a. How do you know your code works?
    i. The question can't be answered for the program as a whole, so define conditions that are "correct" at some particular point.

II. Object Invariants
  a. Is the representation correct?
  b. If count == 0 array = null, else array.length >= count
  c. Can describe many things.
  d. Two common uses
    i. Range of valid values for a particular field
    ii. Relationships between fields. These are the more interesting ones.
  e. Defining
    i. Define the invariant when choosing the representation.
    ii. Can be formal, using mathematical language.
    iii. Can use common language.
    iv. Can use code or pseudocode.
    v. Same pros/cons apply as for function documentation.
    vi. Anything is fine for CS100, but…
      1. Be precise!
      2. Make sure it's understandable.
      3. Invariants can't mean different things to different people.
  f. Consistency
    i. It *says* it's invariant, but that's not always true.
    ii. In general, the invariant should hold at the end of any method's execution.
      1. That can be problematic if methods call other methods.
      2. Don't necessarily assume the invariant is true at the beginning of any method!
    iii. Eg: Assume two variables must always be equal.
      1. One could be changed, then the other.
      2. Even if both remain equal in the end, there is a point where the invariant is false.
    iv. Invariants ultimately don't apply to any particular point, but to the whole class.
  g. Relationship to Code
    i. Invariants are expressed as comments, usually near where the representation is defined.
    ii. A repok() function could be written to check invariants at runtime. This would be for debugging purposes.

III. Code Invariants
  a. Method entrance
  b. Method exit
  c. Loop
    i. Describe what the state will be on EVERy iteration.
    ii. Can be defined for the beginning or end of the loop, but it's often easier to define at the end.
    iii. Ex: (at the bottom of a summation loop): total = sum(arr[0], arr[I])
    iv. Ex: In max(): 'max' is largest of arr[0]..arr[I]
  d. Proving Code
    i. Beyond the scope of CS100
    ii. Provide the basis to prove that the code is correct.
    iii. Precondition + Code = Postcondition
    iv. Loop invariants provide lemmas that make the proofs easier.
    v. Not usually worth the hassle, except in safety-critical applications.

IV. Testing Invariants

a. Testing should be built-in
b. Use `assert bool-expression;`
c. If bool-expression is false, a runtime exception is thrown.
d. Put each invariant in an assert statement.
e. asserts are only checked if you run with –ea flags from the command line.
    i. This means that the asserts can be left in the code and not cost any execution time for the customer.
    ii. When running through BlueJ, asserts will always be checked.
f. Can also say `assert expression : value;`
    i. Gives extra information in the exception.
    ii. Highly recommended.
    iii. Any object can be used for value; its toString() method will be called.
g. BlueJ, by default, doesn't allow asserts
    i. Set in preferences.
    ii. From command-line, use –source 1.4
    iii. Otherwise asserts won't compile because it's not supported before Java 1.4
h. Eg: `assert i > 0 : "i = " + i;`
i. C/C++
    i. Also has an assert
    ii. Looks like a function, but is really a macro.
    iii. Compile with –DASSERT
    iv. Cannot be changed at runtime.
    v. Eg: assert(I > 0);
    vi. There's no way to add a personalized message since macros can't have a variable number of arguments.
    vii. Found in <assert.h>