CSE 143 Java

Specifications: Programming by Contract

Reading: Ch. 7, 8-8.2

Interfaces

- Clients and implementors of an abstraction (e.g. a method or a class) agree on the **interface**

  - A *contract* between the two parties
  - Gives rights and responsibilities of each, to their mutual benefit
- "Interface" usually refers to the agreed-upon types of arguments and results and the possibly thrown exceptions
  - Compliance enforced by Java's compile-time typechecker
  - But this isn't a complete contract!

Specifications

- A **specification** is a (more) complete contract, that should include
  - any restrictions on the allowed argument values
    [A constraint on the *client*, assumed by the implementor]
  - what the return value must be, in terms of the argument values
    [A constraint on the *implementor*, assumed by the client]
  - any changes in state that might happen, and when
    [A constraint on the *implementor*, assumed by the client]
  - when any exceptions might be thrown (more on that later)
    [A constraint on the *implementor*, assumed by the client]
- Example: a deposit method on a BankAccount object

Preconditions and Postconditions

- Two particularly common types of specifications are **preconditions** and **postconditions**
- Precondition: something that must be true before a method/constructor can be called
  - A constraint on the client (the caller)
  - Assumed true by the method implementation

- Postcondition: something that is guaranteed to be true after the method/constructor terminates execution
  - A constraint on the implementor
  - Assumed to be true by the client

- A postcondition is guaranteed only if the preconditions were true when method was called

Examples

- What would be reasonable **preconditions** for

  a square root function?
  a method to add a new item into a set?
  A method to find the earliest date on a file?

- What would be reasonable **postconditions** for

  a square root function?
  a method to add a new item into a set?
  A method to find the earliest date on a file?

Invariants

- An invariant is a condition that must be true at a particular point in a program
- Preconditions and postconditions are examples of invariants
- But there are invariants which are neither pre- nor post-conditions

Class Invariants

- Special case: a **class invariant** – something that is always true for each *instance* of the class, at least as seen from the outside
- Class invariants express requirements on the **values** or relationships of instance variables
  If employee.jobcode "Programmer", then employee.salary > $50,000
  0 <= this.size <= this.capacity
  The list data is stored in this.elements[0..this.size-1]
- A class invariant might not hold while a method is in the middle of updating related variables, but it must ***always*** be true by the time a constructor or method terminates
- Any class invariant is automatically:
  - A postcondition of every constructor and method of that class
  - A precondition of every method

Writing Bug-Free Software

- Invariants, including pre- and post- conditions, are incredibly useful in design and understanding
- Program bugs can often be seen as unforeseen cases of invariants being violated
- In principle:
  - If you could write down all invariants, and have them checked as the program runs, bugs would practically disappear
- In reality:
  1. Writing down all invariants is tedious to impossible
  2. Java gives little direct support for documenting and checking invariants

  The situation is similar in most common languages

Suggested Practice

- Include all non-trivial invariants as comments in the code (use @param, @return, @throws comments if appropriate)
  - These are ***essential*** parts of the design
  - If you don't write them down, the reader (who may be you) will have to reconstruct them as best he/she can
- Whenever you update a variable, double-check any invariants that mention it to be sure the invariant still holds
  - May need to update related variables to make this happen
  - May need to add preconditions (e.g. no negative deposits) or explicit checks (e.g. for overdraft) to ensure they hold
  - Helps to write the code for you!

Dealing With Precondition Failures

- Should preconditions be checked?
  - In an ideal world, no: if all clients satisfy their preconditions, no implementations would need to check them
  - In a world where programs have bugs: maybe we should
    - Prefer programs that crash right away when a problem happens (controversial!)
- Who is responsible for checking?
  - Most logical place is at the beginning of the called method
- How aggressive should we be about checking?
  - If check all preconditions, can clutter up code
  - Focus on checking preconditions that wouldn't crash already, and that would lead to obscure behavior if they weren't detected

What if a Precondition is not True?

- Goal: to force immediate termination
- Reason: the contract has been broken
- Throw a RuntimeException
  - Since these exceptions shouldn't ever be thrown, and clients shouldn't expect to handle them, they shouldn't be listed in throws clauses
  - Not possible to handle the exception to produce some different output or clean-up operation
- Write error messages to System.out or System.err?
  - Might help you during debug
  - Of marginal help in a production environment
    - Neither you nor the client may have access to the console window

Assertions – New Feature of Java 1.4

- Long-time feature of C/C++
- Idea: at any point in the code where some condition should hold, we can write this type of statement:

  assert <*boolean expression*>;
  - If <*boolean expression*> is true, execution continues normally
  - If false, execution stops with an error, or drops into a debugger, ...
- Asserts can be disabled without removing them from the source code
  - Means there is no performance penalty for production code
- Guideline: use aggressively for consistency checking
  - Powerful development tool; helps code to crash early
  - Use to check all types of invariants, not just preconditions
- Unfortunately, not all invariants can be expressed by simple Boolean conditions.

Invariants and Inheritance

- When methods are overriden
  - **pre**conditions can be **weakened** in overriding methods
  - **post**conditions can be **strengthened**
- Class invariants can be strengthened (since the class itself is ensuring they are respected), or even changed arbitrarily, as long as inherited methods still have proper preconditions met when they're called and inheriting code only assumes inherited postconditions are true