

CSE 143 Java

Specifications: Programming by Contract

Reading: Ch. 7, 8-8.2

2/6/2003

(c) University of Washington

10-1

Interfaces

- Clients and implementers 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" has a broad that goes beyond the Java keyword
- "Interface" in the narrow sense 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!

2/6/2003

(c) University of Washington

10-2

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 implementer]
 - what the **return value** must be, in terms of the argument values
[A constraint on the *implementer*, assumed by the client]
 - any **changes in state** that might happen, and when
[A constraint on the *implementer*, assumed by the client]
 - when any **exceptions** might be thrown (more on that later)
[A constraint on the *implementer*, assumed by the client]
- Example:
 - `myAccount.deposit(double);` //allowed args? state changes?
 - `aPerson.getAge();` //return values? state changes?

2/6/2003

(c) University of Washington

10-3

Types vs Values

- Note the emphasis on *values* rather than *types*
 - "any restrictions on the **allowed argument values**"...
 - "what the **return value** must be, in terms of the argument values"...
 - "any **changes in state** that might happen"...
- Compilers are good at checking types
- Compilers are weak at checking values
- Values are an aspect of the **semantics** (meaning) of a program.

2/6/2003

(c) University of Washington

10-4

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 implementer
 - Assumed to be true by the client
- A postcondition is guaranteed only if the preconditions were true when method was called



2/6/2003

(c) University of Washington

10-5

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?

2/6/2003

(c) University of Washington

10-6

Invariants

- An **invariant** is a condition that must be true at a particular point in a program
- Preconditions and postconditions are particular kinds of invariants
 - But there are invariants which are neither pre- nor post-conditions
- There generally are an infinite number of invariants
 - `aPerson.getAge();`
 - "result is ≥ 0 and ≤ 150 " is an invariant (postcondition)
 - Others include "result is > -1 ", "result is > -2 ", "result is > -3 ", "result is < 200 ", "result is < 2000 ", "result is < 20000 ", etc. etc.

2/6/2003

(c) University of Washington

10-7

Loop Invariants

- Definition: something that is true on *each* execution of a **loop body**

```
For (int index=1900; index <= 1999; index++) {  
    /* point A */  
    rainfall += rainRecord[index];  
    /* point B */  
}
```

At points A and B, it is true that $0 \leq \text{index} \ \&\& \ \text{index} \leq 1999$.

Other loop invariants may be harder to state.

*"The value of rainfall is equal to the sum of the values
rainRecord[0]...rainRecord[index]"*

This is true at point B but not at point A!

2/6/2003

(c) University of Washington

10-8

Data and Class Invariants

- **Data invariants** express a relationship between variables, especially instance variables
`son.birthyear > mother.birthyear`
- Data invariants often hold true over an extended portion of the program
 - Compare with preconditions, postconditions, loop invariants: only guaranteed to be true at particular points

2/6/2003

(c) University of Washington

10-9

Class Invariants

- Class invariants express *permanent* 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 temporarily...
 - while an object is being constructed
 - 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 of every method of that class

2/6/2003

(c) University of Washington

10-10

Invariants and Inheritance


- When methods are overridden
 - **preconditions** can be **weakened** in overriding methods
 - **postconditions** can be **strengthened**
- Class invariants...
 - can be strengthened (
since only the class itself can ensure they are respected,
 - can even be 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

2/6/2003

(c) University of Washington

10-11

Writing Bug-Free Software

- Program bugs can often be seen as unforeseen cases of invariants being violated
- 
- Writing down invariants of all kinds is incredibly useful in design and understanding
- In principle:
 - If you could write down all invariants, and have them checked as the program runs, practically all bugs would be found
- In reality:
 1. Writing down all invariants is tedious to impossible
 2. Languages give little direct support for documenting and checking invariants
 3. Test cases may not sufficiently exercise invariant checking

2/6/2003

(c) University of Washington

10-12

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.

2/6/2003

(c) University of Washington

18-13

Suggested Practice

- Include simple invariants as *asserts*
 - Serves as documentation and for checking
- 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!

2/6/2003

(c) University of Washington

18-14

Checking Preconditions: Issues

- 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

2/6/2003

(c) University of Washington

18-15

Handling Violated Preconditions

- Goal: to force immediate termination
 - Reason: the contract has been broken
- Assert the condition (and abort)?
- 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

2/6/2003

(c) University of Washington

18-16

Error Checking and Handling Toolbag

- Status returns
- Assert statements
- Throwing checked exceptions
- Throwing unchecked exceptions
- try/catch blocks
- Messages to console
- Messages to system log

2/6/2003

(c) University of Washington

10-17

When to do What?

- Some goals:
 - Correct operation
 - Efficient operation
 - Uninterrupted operation
 - Clarify of programming and design
 - Detection of bugs
 - Removal of bugs
- These goals are sometimes in conflict!
 - Welcome to the real world...

2/6/2003

(c) University of Washington

10-18

Martin's Stratagem

1. Draw a box around the code *you* are responsible for debugging
2. Inside the box, be aggressively aggressive
 1. *Leviak* use of *assert*
 2. Frequent invariant checking, even if redundant
 3. Explicit subexpressions, single-assignment variables (helpful when using Debugger)
 4. Console messages
3. Outside the box, be conservative
 1. Raise exceptions
 2. Write to system logs
 3. Comments and documentation
 4. Fail-safe recovery

2/6/2003

(c) University of Washington

10-19