
CSE 143 Java

Programming by Contract

Reading: Ch. 5

1/27/2005

(c) 2001-5, University of Washington

10-1

Overview

- Topics
 - Kinds of errors
 - Preconditions, postconditions, and invariants
 - Specification as a contract
 - Throwing Exceptions
 - Assertions

1/27/2005

(c) 2001-5, University of Washington

10-2

Example: StringList class

- Here's the interface of a class that implements a simple, fixed-size list data structure. Operations:

```
class StringList {           // a list of strings
    StringList(int capacity); // create new StringList with given capacity
    boolean isEmpty();       // = "this StringList is empty"
    boolean isFull();       // = "this StringList is full"
    int size();             // = # of Strings in this StringList
    boolean add(String str); // add str to this StringList, result true
                          // if success
    boolean contains(String str); // = "this StringList contains str"
    String get(int pos);    // return String at given position
    String remove(int pos); // return String at given position and remove
                          // it from this StringList
}
```

1/27/2005

(c) 2001-5, University of Washington

10-3

StringList Instance Variables

- Representation is an array whose length is fixed when the StringList is created, plus a count of the current number of strings stored in the list

```
class StringList {           // a list of strings
    // instance variables
    private String[] strings; // Strings in this StringList are stored in
    private int size;        // strings[0] through strings[size-1]
    ...
}
```

1/27/2005

(c) 2001-5, University of Washington

10-4

StringList: What Could Go Wrong?

- What kinds of errors could occur in either the implementation or use of StringList
 - This is a *different* question from how would one test for these problems
- For each possible error
 - What could go wrong?
 - How should we deal with it?

1/27/2005

(c) 2001-5, University of Washington

10-5

Error Handling

- Software failures fall into two broad categories
 - Internal programming errors (“bugs”)
 - Failures because of interaction with external resources or users (out of memory, file not found, improper use, etc.)
- Incorrectly formatted data and similar problems also need to be handled, but that is part of normal processing
- For now, focus on software failures
- Principle: If a method detects it is going to fail, it *must* do something appropriate to report the failure; it is *never* acceptable to return to the caller as if nothing happened

1/27/2005

(c) 2001-5, University of Washington

10-6

Preconditions and Postconditions

- Methods typically make assumptions about the state of the world before, during, and after they are executed
 - Typically logical formulas: $0 \leq \text{size} < \text{capacity}$; the array is sorted $a[0] \leq a[1] \leq \dots \leq a[\text{size}-1]$; etc.
- Two key kinds of assumptions
 - *Precondition*: Something that must be true before a method can be called; a requirement
 - *Postcondition*: Something that is guaranteed to be true after a method terminates execution (provided the precondition was true when it was called)

1/27/2005

(c) 2001-5, University of Washington

10-7

Preconditions & Postconditions

- What would be reasonable preconditions for
 - a square root function?
 - a method to insert new item into a list object?
- What would be reasonable postconditions for
 - a sort routine?
 - the constructor for a list object?

1/27/2005

(c) 2001-5, University of Washington

10-8

Class Invariants

- An invariant is a condition that should always be true at a particular place in a program
- Important case: a *class invariant* –an invariant about properties of class instances; often a relationship between instance variables (state)
 - Examples
 - 0 <= size <= capacity
 - The list data is stored in items[0..size-1]
 - Note: a class invariant might be false for a moment while a method is updating related variables, but it must *always* be true by the time a constructor or method terminates

1/27/2005

(c) 2001-5, University of Washington

10-9

Writing Bug-Free Software

- Preconditions, postconditions, and invariants are incredibly useful
- Include all non-trivial ones as comments in the code
 - These are essential parts of the design and a reader must understand them to understand the code
 - 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, check any invariants that refer to it to be sure the invariant still holds
 - May need to update related variables to make this happen

1/27/2005

(c) 2001-5, University of Washington

10-10

Design by Contract

- The preconditions and postconditions of a method can be viewed as a *contract* between the implementer of the method and the client code that uses it
- Clearly specifies the responsibilities of both parties
 - Client must ensure all preconditions are true before calling the method
 - Implementation must guarantee that postconditions are true, provided the preconditions were true when the method was called
 - (assuming that adequate resources are available and other requirements are satisfied – see below)

1/27/2005

(c) 2001-5, University of Washington

10-11

Precondition Failures

- Principle: Crash early!
 - The sooner a precondition failure is detected the better
- Who is responsible for checking?
 - Most logical place is at the beginning of the called method

1/27/2005

(c) 2001-5, University of Washington

10-12

What if a precondition is not true?

- Suppose this method is called with $pos < 0$ or $pos \geq size()$?

```
/** Return list element at given position. Precondition:  $0 \leq pos < size()$  */  
String get (int pos) {  
    ...  
}
```

- What should we do?

1/27/2005

(c) 2001-5, University of Washington

10-13

What if a precondition is not true?

- One solution(?)

```
/** Return list element at given position. Precondition:  $0 \leq pos < size()$   
String get (int pos) {  
    if (pos < 0 || pos  $\geq$  size) {  
        System.out.println("naughty user - pos has bad value in get");  
        return null;  
    } else {  
        return strings[pos];  
    }  
}
```

- Helpful error message, returns something user can check
- Good idea or not?

1/27/2005

(c) 2001-5, University of Washington

10-14

Critique

- Not a good idea for at least two reasons
- Should never have extra output in a method that is not intended to produce output
 - (bad cohesion; also, unexpected output might panic end user)
- Null as an error code (and error codes in general)
 - Can it get confused with a legitimate return value?
 - Will the programmer *always* remember to check?
(What do you think?)

1/27/2005

(c) 2001-5, University of Washington

10-15

Throwing Exceptions

- One good solution: throw an *exception*
- Basic idea: generate a runtime error, exactly as done for things like out-of-bounds array subscripts or null references

```
/** Return list element at given position. Precondition:  $0 \leq pos < size$   
* @throws IndexOutOfBoundsException if pos is invalid */  
String get (int pos) {  
    if (pos < 0 || pos  $\geq$  size) {  
        throw new IndexOutOfBoundsException();  
    }  
    return strings[pos];  
}
```

1/27/2005

(c) 2001-5, University of Washington

10-16

Details

- The statement

```
throw new IndexOutOfBoundsException();
```

creates a new exception object and uses it to signal a particular kind of error

- Simple case: halts execution with a suitable error message
- *Not* the same as a regular return statement – can terminate many active methods at once if nobody catches and recovers from the problem (coming next lecture)
We'll also see how to define new kinds of exceptions (errors)

1/27/2005

(c) 2001-5, University of Washington

10-17

Some common standard Java exceptions

- **IllegalArgumentException**

Parameter value is inappropriate

- **NullPointerException**

Parameter value is null when it should not be

Use this instead of less specific `IllegalArgumentException` if it applies

- **IndexOutOfBoundsException**

Array or list index is out of range

Use this instead of `IllegalArgumentException` if it applies

1/27/2005

(c) 2001-5, University of Washington

10-18

How much checking should we do?

- Can overdo it

- Error checking code can overwhelm normal code
Harder to read, understand, modify
- Checking takes time; can have unacceptable performance penalty

- Distinguish two cases

- Public methods: can't trust the caller
Need to check parameters and signal errors whenever possible
- Non-public methods: programmer controls circumstances under which method is called
Programmer has no one else to blame if something is wrong
Still, worth some sort of check during development to catch bugs

1/27/2005

(c) 2001-5, University of Washington

10-19

Assertions – New in Java 1.4

- Long-time feature of C/C++

- Idea: at any point in the code where some condition should hold, we can write

```
assert <boolean-expression>;
```

- If `<boolean-expression>` is true, execution continues normally
- If false, execution stops with an error, or drops into a debugger
- Variation: can include a message in an assertion

```
assert <boolean-expression> : "error message written if assert fails"
```

1/27/2005

(c) 2001-5, University of Washington

10-20

Enabling Assertions

- Default: asserts are off in Java 1.4 – need to tell the compiler to allow them & tell Java runtime to check them
 - Set option in drjava preferences panel
 - `javac -language 1.4` option for command-line compiler (this is used in the online turnin server for your assignments)
 - “-ea” option in java command line and Eclipse project settings

1/27/2005

(c) 2001-5, University of Washington

10-21

Using Assert

- Class loader options can control whether assertions are checked
- Guideline: use aggressively for consistency checking during debugging
 - Powerful development tool; helps code to crash early
 - Use to check preconditions, but also postconditions, invariants, and other conditions that should be true at particular points in the code;
 - Can be disabled during normal production use if overhead is too high
 - Is this a good idea?

1/27/2005

(c) 2001-5, University of Washington

10-22

Assert vs Exceptions

- Main guideline
 - Use assert to check for programming errors (bugs)
 - Use exceptions to signal failures or problems during execution (network connection fails, all object memory used up, ...)
- What about checking preconditions?
 - These are programming bugs, so use asserts, except that...
 - ... if asserts are disabled these will be missed with disastrous results during execution
 - Best practice: use asserts for internal checking, throw an exception to signal precondition errors due to external client code

1/27/2005

(c) 2001-5, University of Washington

10-23

Summary

- Use assertions and exceptions for disciplined error handling
 - Assert to catch bugs in your code; exceptions for dealing with the outside world
- General principle: it is *much* better to fail early instead of continuing execution in a buggy state
- Coming attraction: exception handling – reacting to and recovering from problems
- Then: on to streams and files

1/27/2005

(c) 2001-5, University of Washington

10-24