CSE1720

Week 01, Lecture 02

Winter 2014 ◆ Thursday, Jan 8, 2014

YORK UNIVERSITÉ UNIVERSITY

Objectives for this class meeting

1. Conduct review of core concepts concerning contracts and pre/post conditions
2. Complete and discuss questions about Exceptions

YORK UNIVERSITÉ UNIVERSITY

Recap about the Utility and Non-Utility Classes

utility classes:

- cannot be instantiated
- all methods and/or fields are static
- e.g., the class `Toolbox`

non-utility classes:

- can be instantiated
- may include both non-static and static methods and/or fields
- e.g., the class `Integer`. `Integer.MAX_INT` is a static field and `Integer.parseInt` is a static method. The class has no non-static fields and `toString()` is a non-static method

3

YORK UNIVERSITÉ UNIVERSITY

Recap about the Client View

- implementers: offers services in the form of classes (utility and non-utility classes)
- clients: make use of the services offered by implementers, subject to the Pre and Post conditions

4

YORK UNIVERSITÉ UNIVERSITY

Recap about the PRE and POST Conditions

- PRE – condition(s) that the client must satisfy
- POST – condition(s) that the implementer must satisfy
- PRE and POST conditions are used to establish correctness.
  - Do the services function according to their specification
  - This is done during development, before services are released and go into production
- PRE and POST conditions eliminate redundancy.
  - Is the CLIENT or IMPLEMENTER responsible for checking the value of the parameters being passed to methods?
  - In the absence of information, both might do this. This could cause an app to be inefficient.

5

YORK UNIVERSITÉ UNIVERSITY

PRE and POST: the Client View

- the Pre and Post conditions are described in the API
- the Pre and Post conditions are sometimes formulated in terms of a boolean expression that must evaluate to `true`

6

YORK UNIVERSITÉ UNIVERSITY

PRE Example

suppose we have the method

```
public String fraxas(int num)
```

- PRE written version :
  `num` must be strictly greater than `0`
  - for the PRE to be met, the written description must hold
- boolean expression version:
  `num > 0`
- for the PRE to be met, `num` must be such that
  `num > 0 == true`

YORK UNIVERSITÉ UNIVERSITY

PRE Example

see sec 2.3.3 for further review

suppose we have the method

```
public String saxa(int num)
```

- PRE written version :
  `num` can be any int
- for the PRE to be met, the written description must hold

…but it will always be trivially true, since the compiler will do the type checking

- boolean expression version:
  `true`
- for the PRE to be met, `num` must be s.t.
  `true == true` (or just `true`)

*…but **any** value of* `num` *will satisfy this, since the boolean expression does not even depend on* `num`

YORK UNIVERSITÉ UNIVERSITY

PRE and POST: the Client View

- in Java, most often the PRE is `true`
  - this means the client needs simply to provide the parameter values (and nothing further in terms of conditions on those values)

- in Java, it quite often happens that the POST consists of both:
  1. a specification of the return, and
  2. a specification of the condition under which exceptions are thrown

YORK UNIVERSITÉ UNIVERSITY

Example

substring

`public String substring(int beginIndex)`

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

```
"unhappy".substring(2) returns "happy"
"Harbison".substring(3) returns "bison"
"emptiness".substring(9) returns "" (an empty string)
```

no precondition is specified
PRE is `true`

Parameters:
`beginIndex` - the beginning index, inclusive.

Returns:
the specified substring.

Throws:
`IndexOutOfBoundsException` - if `beginIndex` is negative or larger than the length of this `String` object.

"returns" and "throws" are parts of the post condition

YORK UNIVERSITÉ UNIVERSITY

Multiple Choice Quiz

- complete the questions to the best of your abilities
- you can discuss with your neighbours
- we will discuss the answers, pending time

Q1 - Example

```
int denom = 0;

int result = 7 / denom;
```

The compiler finds the two statements above to obey all syntax and semantic rules, yet the resulting bytecode contains invalid instructions.

Q3

Exceptions may be thrown due to invalid instructions.

Exceptions may be thrown due to bytecode instruction!

- For example: the method `parseInt` in `Integer` may through an exception. The implementer of the class `Integer` accomplished this by design (by using a statement that makes use of the reserved word `throw`)

Q4 - Example

- Suppose the specification of App01 states that the app prompts the user for two numbers and will output the product. The specification also states the user does not enter two numbers, then the app prints a friendly message and terminates.
- When we run App01, it never crashes. However, when the user enters two numbers, it always outputs the **sum** of the two numbers.
- The bytecode for App01 does not contain any invalid instructions, and yet it is not correct.

Q5

- The running environment consists of the VM, the operating system, the hardware, and the network, if any. (p.429)

YORK UNIVERSITÉ UNIVERSITY

Q7 – counter example

- Here is an app that throws an exception, yet never crashes:

```
1  import type.lib.ToolBox;
2
3  public class L02_Ex1 {
4
5      public static void main(String[] args) {
6
7          try {
8              ToolBox.crash(true, "An exception is thrown!");
9              // the crash method throws an RuntimeException if the first
10             // parameter is true
11         } catch (RuntimeException e) {
12             System.exit(0);
13         }
14     }
15 }
```


YORK UNIVERSITÉ UNIVERSITY

Q8

- This question doesn't make any sense!!!
- The definition of a crash is that the VM terminates the program with a non-clean exit.
- A non-clean exit, by definition, is an exit **without** the VM conducting an orderly shutdown

YORK UNIVERSITÉ UNIVERSITY

Q9

- The VM certainly does have a shutdown routine!!
  - see the slides from L01

YORK UNIVERSITÉ UNIVERSITY

Q10

- the API for a service may stipulate that an exception may be thrown under certain conditions.
- The potential for an exception is part of the post-condition.
- If an exception is thrown, it is not punishment. The method may be merely following its contract.
- if the client violates the pre-condition, the "punishment" is that the implementer is no longer obligated to follow their post condition
  - this might even mean that an exception is NOT thrown, when the client may be expecting this!

YORK UNIVERSITÉ UNIVERSITY

Q11

- exceptions are not intrinsically bad – they are often a features of services
- if an app does not handle exceptions that may potentially arise, this is usually a sign of a not-so-great implementation

YORK UNIVERSITÉ UNIVERSITY

Q12

- The API should specify the PRE and POST. It should not specify consequences.
- Knowing the consequences for pre- and post-condition violation is part of the knowledge base that software developers should have. It is not included in the API documentation.

YORK UNIVERSITÉ UNIVERSITY

Q13

- if a service potentially throws an exception, this will be documented in the post condition.
- It doesn't make any sense for it to be elsewhere in the API
- It violates the principle of contracts if the API omits this piece of information.

YORK UNIVERSITÉ UNIVERSITY

Q14

- The delegation model is built into the VM, it doesn't change depending on the app that the VM is asked to invoke.

YORK UNIVERSITÉ UNIVERSITY

Q15

- If class A uses the services of class B, then A is the client of B!
- The "is-a-client-of" relationship is transitive

- E.g., A is a client of B, B is a client of C, C is a client of D

By transitivity, A is an *indirect* client of D

The "chain" of client relationships, during method invocations, defines the invocation trail.

The trail starts with the main method and ends with the client that does not delegate to any other method.

YORK UNIVERSITÉ UNIVERSITY

Q16

- We start searching for a handler in the **currently executing method**
- the search is transferred to the **caller** method
- repeat up the trail until we reach the main method (all calls ultimately originate in the main method)
- the trail of the search can be seen in the stack trace!

```
1  public class L02_Ex2 {
2
3      public static void main(String[] args) {
4
5          System.out.println("Hello".substring(9));
6      }
7  }
```

```
<terminated> L02_Ex2 [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Jan 9, 2014 12:58:11 AM)
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -4
        at java.lang.String.substring(String.java:1937)
        at java.lang.String.substring(String.java:1904)
        at L02_Ex2.main(L02_Ex2.java:7)
```

25

YORK UNIVERSITÉ UNIVERSITY

Q17

- searching “down” the invocation trail would mean starting in the main method, and then going **down** the stack trace
- This might be technically possible to do, but wouldn’t be a great idea
  - it would violate the principle of delegation

26

YORK UNIVERSITÉ UNIVERSITY

Q18

- the designers of Java actually have the **converse** of this policy, they don't want software developers to explicitly handle or delegate **all** exceptions
  - we know this because some exceptions are *checked* and others are *unchecked* (sec 11.3.3)
  - if a client makes use of a services that potentially throws a checked exception, the compiler will **enforce** exception handling
  - this means that the client must explicitly handle the exception
    - either with a try-catch or a throws clause
    - if no explicit handling, compiler error!!
- but a big subset of the exceptions are unchecked!!

YORK UNIVERSITÉ UNIVERSITY