Specifications & Exceptions II

Spring 2013
Today’s Topics

Review & more on exceptions
Review & more on specifications
Java Exceptions

Unchecked

- Used for unexpected/catastrophic failures
- E.g NullPointerException: “Thrown when an application attempts to use null in a case where an object is required.” [Java Platform Docs]
- Compiler does not check that methods declare these
- Compiler does not check that callers declare or catch unchecked exceptions

Checked

- Used for special results
- Compiler requires method signature state that may throw checked exception
- Caller must either also declare it throws that exception, or handle it
What’s Unchecked vs Checked?

Depends on where in the class hierarchy of Throwable.

Checked

Unchecked exceptions

Reserved for internal JVM errors*
Preconditions & Exceptions

A precondition does NOT require it be checked.

- Often reason for precondition is checking is very expensive
- Example: a find() method using binary search
  - Checking whether data is sorted is more expensive than search
  - Makes sense for precondition to say data is sorted

Users of methods dislike preconditions.

- Burden of ensuring precondition is true
- Not necessarily a predictable way to recover
- This is why Java standard libraries make large effort to check preconditions

Ultimately, an engineering decision.

- Key factors: cost of check, complexity of checking, scope of method’s users
Exceptions: Design Considerations

So far: use checked exceptions for special results, unchecked exceptions for failures.

Problem: Java’s exceptions are costly
- Performance cost: can reduce performance by >10x
- Pain to use
  - Defining your own exception classes
  - Using try/catch or including throws in signature
  - So painful, even some Java library methods use null as special value

Refined rule
- Use an unchecked exception if clients will usually ensure the exception won’t happen, because there is an easy way to avoid the exception, or because the exception is due to catastrophic/unexpected failure
- Otherwise, use a checked exception.
Stack Example
Exception Abuse II

```csharp
public static class NumberHelpers
{
    public static ApplicationException EvenOrOdd(int integer)
    {
        if (integer % 2 == 0) {
            return new ApplicationException("The integer is even.");
        } else {
            return new ApplicationException("The integer is odd.");
        }
    }
}
```

// example usage
void btnTest_Click(object sender, EventArgs e)
{
    try {
        throw NumberHelpers.EvenOrOdd(Convert.ToInt32(txtIntToTest.Text));
    } catch (ApplicationException ex) {
        litResult.Text = ex.Message;
    }
}
```
Exception Commandments

1. Do not design APIs that force exceptions for control flow
2. All unchecked exceptions you use should subclass RuntimeException; all checked exceptions should subclass Exception
3. Avoid unnecessary checked exceptions
   - Instead, provide API method that checks condition before call
   - E.g Iterator<T> hasNext() function
4. Reuse Java’s already-defined exceptions when appropriate
5. Judiciously use exception translation
   - When a higher-level exception is more informative/appropriate
Exception Commandments

6. Exceptions are much more useful if they contain useful information about what caused the failure

- Almost never appropriate to just throw the exception

7. Never allow partial mutations


9. Never ignore exceptions!

Adapted from Bloch, “Effective Java”.
Review: Why Specifications?

Many bugs due to misunderstanding *interfaces between pieces of code*.

For implementer

- Freedom to change implementation (but still follow spec!)
- Easier to determine where the error is

For consumer/client

- Use methods without reading all the code
- Describe how to ensure you get what you expect
Preconditions/Postconditions/Frame conditions

**Preconditions (“requires”)**
- Conditions on state that must be true for the implementation to work
- Assumptions made by implementer
- If not true, then implementation behavior is undefined
  - Infinite loops, exceptions, crashes, weird/unexpected results, etc.

**Postconditions (“effects”)**
- What the implementer of the method promises to do
- Often, what the method returns

**Frame condition (“modifies”)**
- Identifies which objects may be modified
- … and implicitly, all other objects will not be modified
- Often omitted, meaning nothing modified
Preconditions/Postconditions/Frame conditions

**Preconditions ("requires")**
- Obligations as a user of the function
- Describe expectations/assumptions

**Postconditions ("effects")**
- Result of the method call

**Frame condition ("modifies")**
- Special kind of post-condition
- Restrict how the method changes the environment
What about instance variables?

concat

class String {
    public String concat(String str) {
        Concatenates the specified string to the end of this string.
        If the length of the argument string is 0, then this String object is returned. Otherwise, a new String object is created, representing a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

        Parameters:
        str - the String that is concatenated to the end of this String.
        Returns:
        a string that represents the concatenation of this object's characters followed by the string argument's characters.
    }
}

© 6.005 Course Staff 07-13
What about (mutable) instance variables?

append

public StringBuilder append(String str)

    Appends the specified string to this character sequence.

    The characters of the String argument are appended, in order, increasing the length of this sequence by the length of the argument. If str is null, then the four characters "null" are appended.

    Parameters:
        str - a string.

    Returns:
        a reference to this object.
Completeness & Specifications

```java
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
    }
    return a.length;
}

static int findB (int [] a, int val) {
    for (int i = a.length - 1; i >= 0; i--) {
        if (a[i] == val) return i;
    }
    return -1;
}
```

requires: val occurs in a exactly once
returns: index i such that a[i] = val

Underdetermined Specification
Declarative Specifications

```java
static int findA (int [] a, int val) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == val) return i;
    }
    return a.length;
}
```

Operational Specification

requires: val occurs in a
returns: iterates from the beginning of the list until it finds the first instance of val, and returns its index

requires: val occurs in a
returns: the index of the first occurrence of val in a
Specification Ordering

How do you compare specifications? When are methods replaceable?

Specification A is stronger than or equal to specification B if
- A has weaker or equal preconditions
- A has a stronger or equal postcondition than B, for those states corresponding to states that fulfill B’s preconditions

In other words, you can always weaken the precondition or strengthen the postcondition & use the new method wherever the old one was used.
Judging Specifications

What makes a good specification?

- Same as what makes a good method
- Plus, …

Is this a good specification?

```java
static int minFind(int[] a, int[] b, int val)
requires: …
effects: returns smallest index in arrays a and b at which val appears
```

A good specification should be coherent

- Avoid long argument lists, deeply-nested if statements, random boolean flags, doing too many things, unnecessary side effects
Judging Specifications

Is this a good specification?
static V put(Map<K,V> map, K key, V val)
requires: ...
effects: inserts (key,val) into the mapping, overriding any existing mapping for the key, and returns the old value for the key, unless the key was not in the map, in which case it returns null

Return values should be informative
- Ambiguous return values are more (scarily) common than you might think
- Good design considers information users of your method need/want
Judging Specifications

Is this a good specification?

static void addAll(List<T> a, List<T> b)

requires: …
effects: inserts all elements of list b into list a, unless one of the elements is null, in which case a NullPointerException is thrown

Specifications should be strong enough

- Avoid ambiguities in what mutations occur, especially in exceptional cases
- Checked exceptions don’t save you from partial mutations
Judging Specifications

Is this a good specification?
static File open(String fname)
requires: …
effects: opens the file at the path given by fname

Specifications should be weak enough

➢ Do not guarantee things that may not happen
➢ Think ahead to exceptional cases, especially ones that are likely to occur (e.g. the path is wrong or no permission to read/write)
Judging Specifications

Is this a good specification?
static ArrayList&lt;T&gt; reverse(ArrayList&lt;T&gt; original)
requires: …
effects: returns a new list with the same elements as the original list, in reverse order

Specifications should take advantage of abstraction
➢ Use abstract types where possible
➢ Increases utility of the method for users
➢ Increases implementation freedom for implementers
Testing & Specifications

Previously, talked about **glass box vs black box testing**

- Glass box tests written with knowledge of implementation in mind
- Black box tests written assuming nothing about the implementation
- Both are useful!

**Glass box tests must still respect the specification**

- Tests that violate preconditions but “happen” to work
- Tests that hardcode quirks of implementation
  - E.g. only testing that the first element is returned in our findA()/findB() example

**So why use glass box tests then?**

- Use implementation to guide new testcases that exercise different code paths or part of problem space
Specifications: Summary

Specifications help implementers & consumers of methods
- Define interface between parts of a program

Specification design takes practice & consideration

Good specifications will prevent a large class of bugs & simplify implementation

Test methods based on their specifications, not based on implementations