Section 6:
Interfaces
Midterm Practice

Slides by Vinod Rathnam

with material from Alex Mariakakis
Kellen Donohue, David Mailhot, and Hal Perkins

AGENDA

- Interface
- Midterm
  - Wednesday, February 18th
  - 50 minutes, 10:30am-11:20am
  - **Review Session**: Next Tuesday, February 17th 4:30pm
  - Study!

CLASSES, INTERFACES, TYPES

- The fundamental unit of programming in Java is a class
  - everything is defined in some class
- But Java also provides interfaces…
- Classes can extend other classes and implement interfaces…
- Interfaces can extend other interfaces…
- Some classes are abstract…

RELATIONSHIPS BETWEEN CLASSES

- How do we express relationships between classes?
- Inheritance captures what we want if one class "is-a" specialization of another
  - `class Cat extends Mammal { … }`
- But that's not right for unrelated classes that share a behavior or concept:
  - e.g., Strings, Sets, and Dates are "Comparable" but there are no "is-a" relationships between them
- And what if we want a class with multiple properties?
  - Can't extend multiple classes, even if that would do what we want…

JAVA INTERFACES

- Pure type declaration. Example (without generics):

```
public interface Comparable {
    int compareTo(Object other);
}
```

- Can contain:
  - Method specifications (*no* implementations)
    - implicitly **`public abstract`**
  - Named constants
    - implicitly **`public final static`**

- Cannot create instances of interfaces – they're abstract
  - e.g., can't do **`Comparable c = new Comparable();`**

IMPLEMENTING INTERFACES

- A class can implement one or more interfaces:
  - `class Gadget implements Comparable{ … }`
- Semantics:
  - The implementing class and its instances have the interface type(s) as well as the class type(s)
  - The class must provide or inherit an implementation of all methods defined in the interface(s)

USING INTERFACE TYPES

- An interface defines a type, so we can declare variables and parameters of that type
- A variable with an interface type can refer to an object of *any* class implementing that type
- Examples:

```
List<String> x = new ArrayList<String>();
List<String> y = new LinkedList<String>();
```

PROGRAMMING WITH INTERFACE TYPES

For example

```
void mangle(List victim) { … }
```

Method argument can be anything that has type **`List`** (like an **`ArrayList`** or **`LinkedList`**)

GUIDELINES FOR INTERFACES

- Provide interfaces for significant types / abstractions
- Write code using interface types like **`Map`** wherever possible; only use specific classes like **`HashMap`** or **`TreeMap`** when you need to
  - Allows code to work with different implementations later
- Consider providing classes with complete or partial interface implementation for direct use or subclassing
- Both interfaces and classes are appropriate in various circumstances

MIDTERM PRACTICE

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_____________________}
z = x + y;
{_____________________}
y = z - 3;
{x > y}
```

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_____________________}
z = x + y;
{x > z - 3}
y = z - 3;
{x > y}
```

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{x > x + y – 3 => y < 3}
z = x + y;
{x > z – 3}
y = z – 3;
{x > y}
```

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_____________________}
p = a + b;
{_____________________}
q = a - b;
{p + q = 42}
```

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{_____________________}
p = a + b;
{p + a - b = 42}
q = a - b;
{p + q = 42}
```

WINTER 2013 Q1

Using backwards reasoning, find the weakest precondition for each sequence of statements and postcondition below. Insert appropriate assertions in each blank line. You should simplify your answers if possible.

```
{a + b + a - b = 42 => a = 21}
p = a + b;
{p + a - b = 42}
q = a - b;
{p + q = 42}
```

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

I.
```
void withdraw(int amount) {
    balance -= amount;
}
```

A	B	C

Another way to ask the question:

If the client does not know the implementation, will the method do what he/she expects it to do based on the specification?

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

I.
```
void withdraw(int amount) {
    balance -= amount;
}
```

A	B	C

X if specification is met
O if it is not

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

I.
```
void withdraw(int amount) {
    balance -= amount;
}
```

A	B	C
X		

a. The method does exactly what the spec says

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

I.
```
void withdraw(int amount) {
    balance -= amount;
}
```

A	B	C
X	X	

a. **The method does exactly what the spec says**
b. **If the client follows the `@requires` precondition, the code will execute as expected**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

I.
```
void withdraw(int amount) {
    balance -= amount;
}
```

A	B	C
X	X	O

a. **The method does exactly what the spec says**
b. **If the client follows the `@requires` precondition, the code will execute as expected**
c. **The method does not throw an exception**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

II.
```
void withdraw(int amount) {
    if (balance >= amount) balance -= amount;
}
```

A	B	C

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

II.
```
void withdraw(int amount) {
    if (balance >= amount) balance -= amount;
}
```

A	B	C
O		

a. **The `balance` will not always decrease**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

II.
```
void withdraw(int amount) {
    if (balance >= amount) balance -= amount;
}
```

A	B	C
O	X	

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

II.
```
void withdraw(int amount) {
    if (balance >= amount) balance -= amount;
}
```

A	B	C
O	X	O

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**
c. **The method does not throw an exception**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

III.
```
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```

A	B	C

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

III.
```
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```

A	B	C
O		

a. **The `balance` will not always decrease**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

III.
```
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```

A	B	C
O	X	

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

III.
```
void withdraw(int amount) {
    if (amount < 0) throw new IllegalArgumentException();
    balance -= amount;
}
```

A	B	C
O	X	O

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**
c. **The method throws the wrong kind of exception and for the wrong reason**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

IV.
```
void withdraw(int amount) throws InsufficientFundsException{
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```

A	B	C

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

IV.
```
void withdraw(int amount) throws InsufficientFundsException{
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```

A	B	C
O		

a. **The `balance` will not always decrease**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

IV.
```
void withdraw(int amount) throws InsufficientFundsException{
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```

A	B	C
O	X	

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**

WINTER 2013 Q2

Suppose we have a `BankAccount` *class with instance variable* `balance`*. Consider the following specifications:*

A.
```
@effects decreases balance by amount
```

B.
```
@requires amount >= 0 and amount <= balance
@effects decreases balance by amount
```

C.
```
@throws InsufficientFundsException if balance < amount
@effects decreases balance by amount
```

Which specifications does this implementation meet?

IV.
```
void withdraw(int amount) throws InsufficientFundsException{
    if (balance < amount) throw new InsufficientFundsException();
    balance -= amount;
}
```

A	B	C
O	X	X

a. **The `balance` will not always decrease**
b. **If the client follows the `@requires` precondition, the code will execute as expected**
c. **The method does exactly what the spec says**

WINTER 2013 Q3

```
/**
* An IntPoly is an immutable, integer-valued polynomial
* with integer coefficients. A typical IntPoly value
* is a_0 + a_1*x + a_2*x^2 + ... + a_n*x_n. An IntPoly
* with degree n has coefficent a_n != 0, except that the
* zero polynomial is represented as a polynomial of
* degree 0 and a_0 = 0 in that case.
*/

public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.
}
```

WINTER 2013 Q3

```
public class IntPoly {
    /**
    * Return a new IntPoly that is the sum of this
    and other
    * @requires
    * @modifies
    * @effects
    * @return
    * @throws
    */
    public IntPoly add(IntPoly other);
}
```

WINTER 2013 Q3

```
public class IntPoly {
    /**
    * Return a new IntPoly that is the sum of this
    and other
    * @requires other != null
    * @modifies none
    * @effects none
    * @return a new IntPoly that is the sum of this
    *          and the other
    * @throws none
    */
    public IntPoly add(IntPoly other);
}
```

WINTER 2013 Q3

```
public class IntPoly {
    /**
    * Return a new IntPoly that is the sum of this
    and other
    * @requires other != null
    * @modifies none
    * @effects none
    * @return a new IntPoly that is the sum of this
    *          and the other
    * @throws none
    */
    public IntPoly add(IntPoly other);
}
```

Note: if you have an instance variable in `@modifies`, it better appear in `@effects` as well

WINTER 2013 Q3

```
public class IntPoly {
    /**
    * Return a new IntPoly that is the sum of this
    and other
    * @requires other != null
    * @modifies none
    * @effects none
    * @return a new IntPoly that is the sum of this
    *          and the other
    * @throws none
    */
    public IntPoly add(IntPoly other);
}
```

Note: if you have an instance variable in `@modifies`, it better appear in `@effects` as well

Note2: this is not the only answer, you could specify an exception in `@throws` or specify the output in `@return`

WINTER 2013 Q4

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

// Return the coefficients of this IntPoly
public int[] getCoeffs() {
        return a;
}
```

One of your colleagues is worried that this creates a potential representation exposure problem. Another colleague says there's no problem since an `IntPoly` *is immutable. Is there a problem? Give a brief justification for your answer.*

WINTER 2013 Q4

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

// Return the coefficients of this IntPoly
public int[] getCoeffs() {
        return a;
}
```

One of your colleagues is worried that this creates a potential representation exposure problem. Another colleague says there's no problem since an `IntPoly` *is immutable. Is there a problem? Give a brief justification for your answer.*

Yes there is a problem. The return value is a reference to the same coefficient array stored in the `IntPoly` and the client code could alter those coefficients.

WINTER 2013 Q4

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

// Return the coefficients of this IntPoly
public int[] getCoeffs() {
        return a;
}
```

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs()` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

WINTER 2013 Q4

```
public class IntPoly {
    int a[];
    // AF(this) = a has n+1 entries, and for each entry,
    // a[i] = coefficient a_i of the polynomial.

// Return the coefficients of this IntPoly
public int[] getCoeffs() {
        return a;
}
```

If there is a representation exposure problem, give a new or repaired implementation of `getCoeffs()` *that fixes the problem but still returns the coefficients of the* `IntPoly` *to the client. If it saves time you can give a precise description of the changes needed instead of writing the detailed Java code.*

Create a new array the same length as `a`, copy the contents of `a` to it, and return the new array.

WINTER 2013 Q5

We would like to add a method to this class that evaluates the `IntPoly` *at a particular value x. In other words, given a value x, the method* `valueAt(x)` *should return* $a_0 + a_1x + a_2x^2 + ... + a_nx^n$*, where* a_0 *through an are the coefficients of this* `IntPoly`*.*

For this problem, develop an implementation of this method and prove that your implementation is correct.

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {_____}
        while (k != n) {
                {_____}
                xk = xk * x;
                {_____}
                val = val + a[k+1]*xk;
                {_____}
                k = k +1;
                {_____}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {_____}
                xk = xk * x;
                {_____}
                val = val + a[k+1]*xk;
                {_____}
                k = k +1;
                {_____}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {inv && k != n}
                xk = xk * x;
                {_____}
                val = val + a[k+1]*xk;
                {_____}
                k = k +1;
                {_____}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {inv && k != n}
                xk = xk * x;
                {xk = x^(k+1) && val = a[0] + a[1]*x + … + a[k]*x^k}
                val = val + a[k+1]*xk;
                {_____}
                k = k +1;
                {_____}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {inv && k != n}
                xk = xk * x;
                {xk = x^(k+1) && val = a[0] + a[1]*x + … + a[k]*x^k}
                val = val + a[k+1]*xk;
                {xk = x^(k+1) && val = a[0] + a[1]*x + … + a[k+1]*x^(k+1)}
                k = k +1;
                {_____}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {inv && k != n}
                xk = xk * x;
                {xk = x^(k+1) && val = a[0] + a[1]*x + … + a[k]*x^k}
                val = val + a[k+1]*xk;
                {xk = x^(k+1) && a[0] + a[1]*x + … + a[k+1]*x^(k+1)}
                k = k +1;
                {inv}
        }
        {_____}
        return val;
}
```

WINTER 2013 Q5

```
/** Return the value of this IntPoly at point x */
public int valueAt(int x) {
        int val = a[0];
        int xk = 1;
        int k = 0;
        int n = a.length-1; // degree of this, n >=0
        {inv: xk = x^k && val = a[0] + a[1]*x + … + a[k]*x^k}
        while (k != n) {
                {inv && k != n}
                xk = xk * x;
                {xk = x^(k+1) && val = a[0] + a[1]*x + … + a[k]*x^k}
                val = val + a[k+1]*xk;
                {xk = x^(k+1) && a[0] + a[1]*x + … + a[k+1]*x^(k+1)}
                k = k +1;
                {inv}
        }
        {inv && k = n => val = a[0] + a[1]*x + … + a[n]*x^n}
        return val;
}
```

WINTER 2013 Q6

Suppose we are defining a class to represent items stocked by an online grocery store. Here is the start of the class definition, including the class name and instance variables:

```
public class StockItem {
        String name;
        String size;
        String description;
        int quantity;

        /* Construct a new StockItem */
        public StockItem(…);
}
```

WINTER 2013 Q6

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
        return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

WINTER 2013 Q6

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
        return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

Object s1 = new StockItem("thing", 1, "stuff", 1);
Object s2 = new StockItem("thing", 1, "stuff", 1);
System.out.println(s1.equals(s2));

WINTER 2013 Q6

A summer intern was asked to implement an `equals` *function for this class that treats two* `StockItem` *objects as equal if their* `name` *and* `size` *fields match. Here's the result:*

```
/** return true if the name and size fields match */
public boolean equals(StockItem other) {
        return name.equals(other.name) && size.equals(other.size);
}
```

This `equals` *method seems to work sometimes but not always. Give an example showing a situation when it fails.*

Object s1 = new StockItem("thing", 1, "stuff", 1);
Object s2 = new StockItem("thing", 1, "stuff", 1);
System.out.println(s1.equals(s2));

The equals method was overloaded, rather than overwritten

WINTER 2013 Q6

Show how you would fix the `equals` *method so it works properly (*`StockItems` *are equal if their* `names` *and* `sizes` *are equal)*

```
/** return true if the name and size fields match */
```

WINTER 2013 Q6

Show how you would fix the `equals` *method so it works properly (*`StockItems` *are equal if their* `names` *and* `sizes` *are equal)*

```
/** return true if the name and size fields match */
@ Override
public boolean equals(Object o) {
        if (!(o instanceof StockItem))
                return false;
        StockItem other = (StockItem) o;
        return name.equals(other.name) && size.equals(other.size);
}
```

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

legal	wrong

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

legal	wrong
X	

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

legal	wrong
X	
X	

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

legal	wrong
X	
X	
	X

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

legal	wrong
X	
X	
	X
	X

WINTER 2013 Q6

Which of the following implementations of `hashCode()` *are legal:*

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + quantity;
}

public int hashCode() {
        return quantity;
}
```

The `equals` method does not care about `quantity`

legal	wrong
X	
X	
	X
	X

WINTER 2013 Q6

Which implementation do you prefer?

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}
```

WINTER 2013 Q6

Which implementation do you prefer?

```
public int hashCode() {
        return name.hashCode();
}

public int hashCode() {
        return name.hashCode()*17 + size.hashCode();
}
```

(ii) will likely do the best job since it takes into account both the size and name fields. (i) is also legal but it gives the same `hashCode` for `StockItems` that have different sizes as long as they have the same name, so it doesn't differentiate between different `StockItems` as well as (ii).

WINTER 2013 Q7

Suppose we are specifying a method and we have a choice between either requiring a precondition (e.g., `@requires: n > 0`*) or specifying that the method throws an exception under some circumstances (e.g.,* `@throws IllegalArgumentException if n <= 0`*).*

Assuming that neither version will be significantly more expensive to implement than the other and that we do not expect the precondition to be violated or the exception to be thrown in normal use, is there any reason to prefer one of these to the other, and, if so, which one?

WINTER 2013 Q7

Suppose we are specifying a method and we have a choice between either requiring a precondition (e.g., `@requires: n > 0`*) or specifying that the method throws an exception under some circumstances (e.g.,* `@throws IllegalArgumentException if n <= 0`*).*

Assuming that neither version will be significantly more expensive to implement than the other and that we do not expect the precondition to be violated or the exception to be thrown in normal use, is there any reason to prefer one of these to the other, and, if so, which one?

It would be better to specify the exception. That reduces the domain of inputs for which the behavior of the method is unspecified. It also will cause the method to fail fast for incorrect input, which should make the software more robust – or at least less likely to continue execution with erroneous data.

Note: You could just as easily argue the other way. It may be better to specify the precondition because once the exception is in the specification, it has to stay there because the client may expect it.

WINTER 2013 Q8

Suppose we are trying to choose between two possible specifications for a method. One of the specifications S is stronger than the other specification W, but both include the behavior needed by clients. In practice, should we always pick the stronger specification S, always pick the weaker one W, or is it possible that either one might be the suitable choice? Give a brief justification of your answer, including a brief list of the main criteria to be used in making the decision.

WINTER 2013 Q8

Suppose we are trying to choose between two possible specifications for a method. One of the specifications S is stronger than the other specification W, but both include the behavior needed by clients. In practice, should we always pick the stronger specification S, always pick the weaker one W, or is it possible that either one might be the suitable choice? Give a brief justification of your answer, including a brief list of the main criteria to be used in making the decision.

Neither is necessarily better. What is important is picking a specification that is simple, promotes modularity and reuse, and can be implemented efficiently.

(Many answers focused narrowly on which would be easier to implement. While that is important – we don't want a specification that is impossible to build – it isn't the main thing that determines whether a system design is good or bad.)

MIDTERM TOPICS

- Reasoning about code
- Specifications vs. Implementation
- ADT’s
- Interfaces & classes
- Testing
- Exceptions & assertions
- Identity & equality
- Subtypes & subclasses