CSE 331

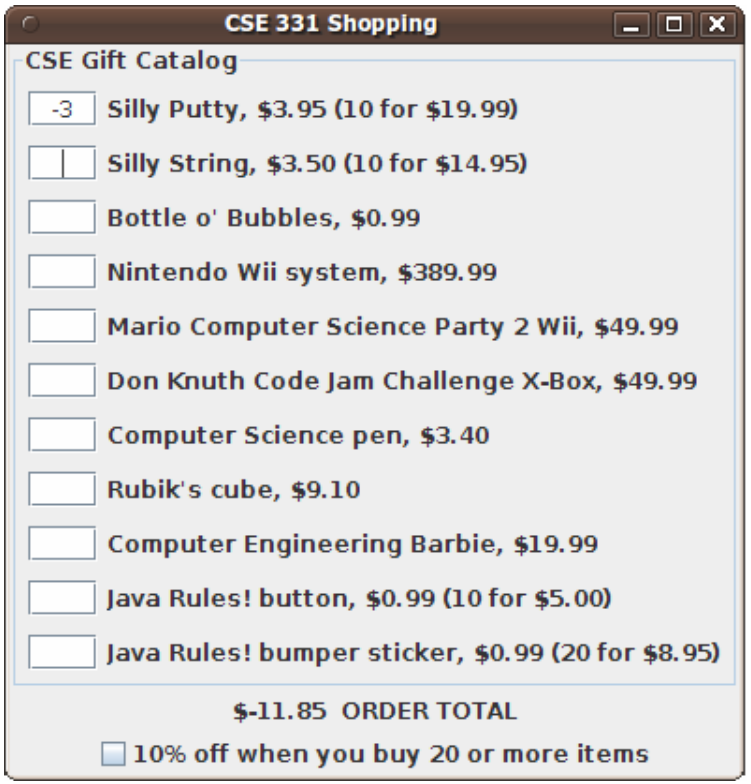
Programming by contract:
pre/post conditions; Javadoc

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

From HW1 spec

- ***"Assume valid parameters**. You may assume that all parameter values passed to all methods and constructors are valid: that prices are always greater than 0, quantities are non-negative, and all objects are non-null."*

- What do you think of this?
  - In real production code, you usually cannot make these kinds of sweeping assumptions or demands on how clients use your code.
  - Well-specified code will be more granular in how it handles misusage.



Effective Java Tip #38

- **Tip #38:** Check parameters for validity.
  - If your method has a notion that some values are "invalid" and knows how to identify those values early in the code.
  - If it's not very expensive to check them.

- But...
  - How does the client know which parameters are / aren't checked?
  - What should you do if they aren't valid?
  - How does the client know what you're going to do if they're invalid?

Programming by contract

- **programming by contract (design by contract)**: Defining formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants.

- Three key questions that the designer must repeatedly ask:
  - What does this code expect?
  - What does it guarantee?
  - What does it maintain?

Preconditions

- **precondition**: Something *assumed to be true* at the start of a call.

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    return elementData[index];
}
```

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	0	0	0	0
size	6									

  - Stating a precondition doesn't "solve" the problem of users passing improper indexes, but it at least documents our decision and warns the client what not to do.

Choosing preconditions

- Examples of poorly chosen preconditions:
  - Stating the obvious:
    - pre: String s is a string! The computer has enough memory to run!
  - Making up for a lazy or poor implementation:
    - for pow: Exponent can't be negative; can only compute positive powers.
    - for list.isSorted: List shouldn't contain any duplicates because our code messes up in that case and returns the wrong answer.
  - Things that clients cannot check, avoid, or ensure:
    - for stack.push: Stack's internal array capacity must be >= stack size.
    - for a download: If it starts, the whole file will arrive successfully.

- **Horstmann OOD Tip 3.6.1**: The client must be able to check the preconditions of a method before calling it.

Precondition violations

- Formally, if a client violates a precondition, (by default) the object does not specify what will happen.
  - It makes *no promise* that the method will work successfully.
    - might do nothing
    - might return an unusual value or "error" value (null, 0, -1, "", etc.)
    - might throw an exception
    - might get stuck in an infinite loop
    - might leave the object in a corrupt state, save the wrong file, etc.
  - *What is the best way to handle a precondition violation?*

Approach 1: return error value

- can handle a precondition violation by returning a special value:

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    if (index < 0 || index >= size) {
        return -1;
    } else {
        return elementData[index];
    }
}
```

  - Is this a good or bad approach?
    - Bad. The -1 returned is indistinguishable from a -1 in the actual data.
    - Bad. The client might not

Approach 2: throw exception

- can handle a precondition violation by throwing an exception:

```
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(index);
    } else {
        return elementData[index];
    }
}
```

  - **fail-fast**: Client learns about the problem immediately and can fix it. Passing a bad value usually indicates a bug in the client, so this is good.

Exceptions in the contract

- from `java.util.Stack : public E pop()`
  - Removes the object on top of this stack and returns it.
  - Returns: The object at the top of this stack.
  - Throws: `EmptyStackException` - if stack is empty.

- Most preconditions are things the stack assumes to be true.
  - (and, as far as the client knows, that are not checked by the stack)
  - If client violates a precondition, stack could do anything.

- In this case the stack documents a predictable behavior (throw) in response to the empty stack condition.
  - So we say that *the exception is part of the contract* .
  - If you change it (say, to return `null`), you have changed the contract.

Preconditions and private

- Private internal methods do not usually test preconditions:

```
// Helper does the real work of removing an item.
private void removeHelper(int index) {
    // should I check 0 <= index < size here?
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] = 0;
    size--;
}
```

- Why not?
  - Since the method can only be called internally, the class author can make sure to call it only when the preconditions hold.
  - If any check at all is made, make it an `assert` statement (see next).

Precondition example

- Binary search on an `int[]` : from Java API

  *"Searches the specified array of `int`s for the specified value using the binary search algorithm. The array **must** be sorted (as by the `sort` method, above) prior to making this call. If it is not sorted, the results are undefined. ..."*

- Why doesn't Sun just check whether the array is sorted? :
  - Idea #1: If it isn't sorted, sort it.
  - Idea #2: If it isn't sorted, throw an exception.

  - Sort is costly (takes O(n log n) or worse; search is O(log n)).
  - Even checking to see whether the array is sorted is costly (O(n)); omitting this check and assuming it to be true makes binary search run much faster.
  - Sort modifies the input array; binarySearch would have a side effect.

    - So how do we catch bugs where the client violates this precondition? ...

Checking preconditions

- **assertion**: A logical statement that can be made about a program at a particular point in time and is expected to be true.
  - "At this point in the code, it should be the case that x > 0."

- Java and other languages supply an `assert` statement.
  - Assert statements can be enabled/disabled; they are off by default.
  - Assertions should be used to check your basic assumptions that should never fail; they uncover things that should not have happened!
    - For example, verify preconditions when testing/debugging.
  - When an assertion fails, this is considered an error on the part of the developer and should be fixed immediately.
    - Exceptions in the contract are more common.

Assertions in Java

```
assert condition ;
assert condition : message;
```

- enabling assertions
  - `java -enableassertions ClassName`

    (or tell your editor/IDE to enable them)
    - Assertion code is zero-cost when disabled; very important!

  - In C/C++, `assert` is a compile-time thing.
  - In Java, you can selectively en/disable assertions at runtime.

Assert statement example

```
// Returns index of n in a, or -1 if not found.
// precondition: a is in sorted order.
public static int binarySearch(int[] a, int n) {
    assert isSorted(a) : "Array must be sorted";
    ...
}

// Returns true if the given array is sorted.
public static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            return false;
        }
    }
    return true;
}
```

Postconditions

- **postcondition**: Something your method *promises will be true* at the end of its execution, if all preconditions were true at the start.

```
// Makes sure that this list's internal array is large
// enough to store the given number of elements.
// Precondition: capacity >= 0
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    while (capacity > elementData.length) {
        elementData = Arrays.copyOf(elementData,
                                    2 * elementData.length);
    }
}
```

  - If your method states a postcondition, clients should be able to rely on that statement being true after they call the method.

Javadoc comments

```
/**
 *  description of class/method/field/etc.
 *
 *  @tag  attributes
 *  @tag  attributes
 *  ...
 *  @tag  attributes
 */
```

- **Javadoc comments**: Special comment syntax for describing detailed specifications of Java classes and methods.
  - Put on all class headers, public methods, constructors, public fields, ...
    - *Main benefit:* Tools can turn Javadoc comments into HTML spec pages.
  - Eclipse and other editors have useful built-in Javadoc support.
    - *Main drawback:* Comments can become bulky and harder to read.

Javadoc tags

- on a method or constructor:

tag	description
`@param` ***name description***	describes a parameter
`@return` ***description***	describes what value will be returned
`@throws` ***ExceptionType reason***	describes an exception that may be thrown (and what would cause it to be thrown)
`{@code` ***sourcecode*** `}`	for showing Java code in the comments
`{@inheritDoc}`	allows a subclass method to copy Javadoc comments from the superclass version

- on a class header:

tag	description
`@author` ***name***	author of a class
`@version` ***number***	class's version number, in any format

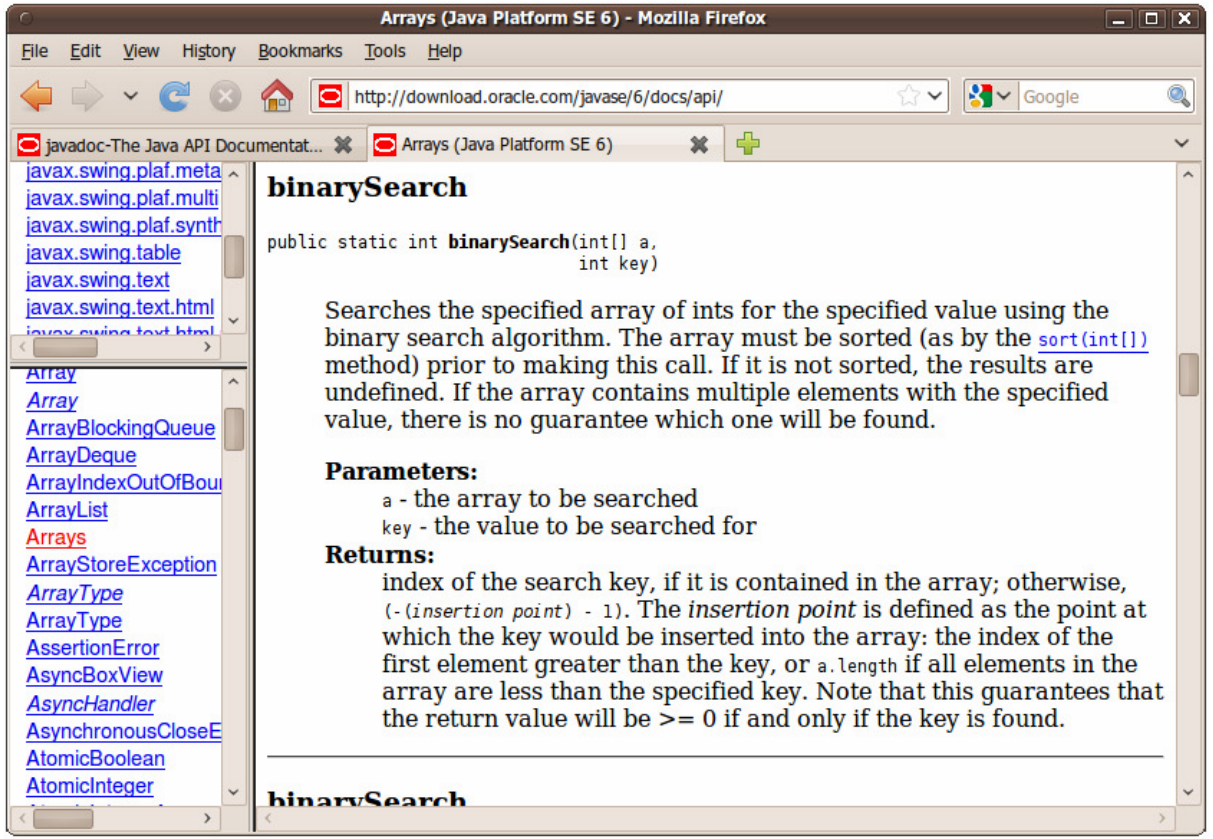
Javadoc example

```
/**
 * Each BankAccount object models the account information for
 * a single user of Fells Wargo bank.
 * @author James T. Kirk
 * @version 1.4 (Aug 9 2008)
 */
public class BankAccount {
    /** The standard interest rate on all accounts. */
    public static final double INTEREST_RATE = 0.03;
    ...

    /**
     * Deducts the given amount of money from this account's
     * balance, if possible, and returns whether the money was
     * deducted successfully (true if so, false if not).
     * If the account does not contain sufficient funds to
     * make this withdrawal, no funds are withdrawn.
     *
     * @param amount the amount of money to be withdrawn
     * @return true if amount was withdrawn, else false
     * @throws IllegalArgumentException if amount is negative
     */
    public boolean withdraw(double amount) {
        ...
    }
}
```

Javadoc output as HTML

- Java includes tools to convert Javadoc comments into web pages
  - from Terminal: `javadoc -d doc/ *.java`
  - Eclipse has this built in: Project → Generate Javadoc...
  - The actual Java API spec web pages are generated from Sun's Javadoc comments on their own source code:



Javadoc HTML example

- from `java.util.List` interface source code:

```
/**
 * Returns the element at the specified position
 * in this list.
 * <p>This method is <em>not</em> guaranteed to run
 * in constant time. In some implementations it may
 * run in time proportional to the element position.
 *
 * @param index index of element to return; must be
 *          non-negative and less than size of this list
 * @return the element at the specified position
 * @throws IndexOutOfBoundsException if the index is
 *          out of range
 *          ({@code index < 0 || index >= this.size()})
 */
public E get(int index);
```

  - Notice that HTML tags may be embedded inside the comments.

Javadoc enums, constants

- Each class constant or enumeration value can be commented:

```
/**
 * An instrument section of a symphony orchestra.
 * @author John Williams
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as trumpet. */
    BRASS,

    /** Percussion instruments, such as cymbals. */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

What goes in @param/return

- Don't repeat yourself or write vacuous comments.

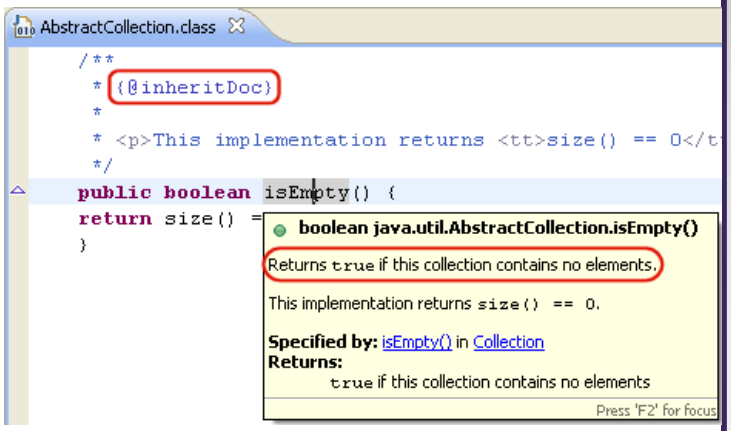
```
/** Takes an index and element and adds the element there.
 * @param index index to use
 * @param element element to add
 */
public boolean add(int index, E element) { ...
```

- better:

```
/** Inserts the specified element at the specified
 * position in this list.  Shifts the element currently at
 * that position (if any) and any subsequent elements to
 * the right (adds one to their indices).  Returns whether
 * the add was successful.
 * @param index index at which the element is to be inserted
 * @param element element to be inserted at the given index
 * @return true if added successfully; false if not
 * @throws IndexOutOfBoundsException if index out of range
 *         ({@code index < 0 || index > size()})
 */
public boolean add(int index, E element) { ...
```

Your Javadoc is your spec

- **Whenever you write a class to be used by clients, you should write full Javadoc comments for all of its public behavior.**
  - This constitutes your specification to all clients for your class.
  - You can post the generated HTML files publicly for clients to view.
  - Common distribution of a library of classes:
    - **binaries** (.class files, often packaged into an archive)
    - **specification** (Javadoc .html files, or a public URL to view them)
  - Eclipse uses Javadoc for auto-completion.



- **Effective Java Tip #44:**
  Write Javadoc comments for all exposed API elements.

  (anything that is non-private)

Javadoc and private

- Private internal methods do not need Javadoc comments:

```
/** ... a Javadoc comment ... */
public void remove(int index) { ... }

// Helper does the real work of removing
// the item at the given index.
private void removeHelper(int index) {
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] = 0;
    size--;
}
```

  - Private members do not appear in the generated HTML pages.

Custom Javadoc tags

- Javadoc doesn't have tags for pre/post, but you can add them:

tag	description
`@pre` ***`condition`*** *`(or @precondition)`*	notes a precondition in API documentation; describes a condition that must be true for the method to perform its functionality
`@post` ***`condition`*** *`(or @postcondition)`*	notes a postcondition in API documentation; describes a condition that is guaranteed to be true at the *end* of the method's functionality, so long as all preconditions were true at the *start* of the method

  - By default, these tags won't show up in the generated HTML. But...

Applying custom Javadoc tags

- from Terminal: `javadoc -d doc/`
  `-tag pre:cm:"Precondition:"`
  `-tag post:cm:"Postcondition:" *.java`
- in Eclipse: Project → Generate Javadoc... → Next → Next →
  in the "Extra Javadoc options" box, type:

```
-tag pre:cm:"Precondition:" -tag post:cm:"Postcondition:"
```

- The generated Java API web pages will now be able to display `pre` and `post` tags properly!