Public and Helper Classes

- **Public classes**
  - Declared as public
  - Reside in a file
- **Auxiliary or helper classes**
  - Should not be declared public
  - Implement other classes
  - Can reside in the same file or in a separate file.

Example

```java
public interface List {
    public int size();
    public boolean isEmpty();
    public Object element(int i);
    public Object head();
    public Object last();
    public void insert(Object item, int i);
    public void insertHead(Object item);
    public void insertLast(Object item);
    public Object remove(int i);
    public Object removeHead();
    public Object removeLast();
}
```

Example

```java
public class LinkedList implements List {
    protected Node head, tail;
    protected int count;
    //...
    //implementation of linked list
}
```

- Not accessible to any subclass of LinkedList not in the same package.

Example

```java
public class LinkedList implements List {
    protected Node head, tail;
    protected int count;
    static protected class Node {
        Object element;
        Node next, prev;
    }
    //...
    //implementation of linked list
}
```

- Accessible to subclass of LinkedList.
Class Members

- The order of class members does not affect semantics of the class
- Fields and methods should be ordered according to their accessibility and roles
- A recommend organization of a public class:

  ```java
  public class AClass {
    // public constants
    // public constructors
    // public accessors or selectors
    // public mutators or modifier
    // nonpublic fields
    // nonpublic auxiliary method or nested classes
  }
  ```

Basic Guidelines for Class and Package Design

- Avoid public fields
  - Should be no nonfinal public fields, except when a class is final and the field is unconstrained
  - Example: for a field named `attr`
    - `getAttr()` — access the value of `attr`
    - `setAttr()` — modify the value of `attr`
    - `isAttr()` — access the boolean value of `attr`

Point Class

```java
public class Point {
  public Point() {}
  public Point(final double x, final double y) {
    this.x = x; this.y = y;
  }
  public double getX() {
    return x;
  }
  public double gety() {
    return y;
  }
  public void setX(final double x) {
    this.x = x;
  }
  public double sety(final double y) {
    this.y = y;
  }
  protected double x, y;
}
```

Public Field vs Nonpublic Field

- Fields in `PolarPoint` are constrained by
  \[ x = r \cos(a) \] and \[ y = r \sin(a) \]
- If the fields `r` and `a` are public
  - `PolarPoint p = new PolarPoint();
    p.r = 100.0;
- If the fields `r` and `a` are nonpublic
  - `PolarPoint p = new PolarPoint();
    p.setR(100.0);`
Completeness of the public interface

• The set of public methods defined in the class should provide full and convenient access to the functionality of the class

Separate interface from implementation

• When the functionality supported by a class can be implemented in different ways, it is advisable to separate the interface from the implementation

public class LinkedList implements List {
    ...
}

public class DynamicArray implements List {
    ...
}

Documenting the Source Code

• Order of Tags -- use in documenting the source code in Java
  - @author (classes and interfaces only, required)
  - @version (classes and interfaces only, required)
  - @param (methods and constructors only)
  - @return (methods only)
  - @exception (@throws is a synonym added in Javadoc 1.2)
  - @see
  - @since
  - @deprecated

Examples

• @author Joe Nobody
• @author John Somebody
• @version 1.0beta
• @param count: the number of lines to print for each match
• @return the number of widgets that pass the quality test
• @exception IndexOutOfBoundsException
• @see java.lang.String
• @see String
• @see java.io.InputStream;
• @see String#equals

Contracts and Invariants

• Contract methods
• Invariants of classes
• Assertions
• Design by contracts
Contracts and Invariants

• Contract of a method is a specification of the behavior of the method.
• Potential problems of missing or informally specified contracts:
  - Incompleteness and silence on some aspects of the behavior
  - Ambiguity and multiple interpretations
  - Contradictions with other contracts
• Purpose of formally specified contracts:
  - Precision and unambiguousness
  - Run-time checking to catch violations of contract by implementation as well as misuse of the services to the clients
  - Facilitation of reasoning about the behavior of both the implementation and the clients

Contracts of Methods

• A precondition of a method is a Boolean expression that must hold when the method is invoked.
• A postcondition of a method is a Boolean expression that holds when the method invocation returns.

Contract of a Method

```java
/**
 * @pre precondition(boolean expression)
 * @post postcondition(boolean expression)
 */
Public void aMethod() {
    //...
}
```

Examples of a List

```java
/**
 * Returns the number of elements in the list
 * @pre true
 * @post @nochange
 */
Public int size() {
}

/**
 * Returns true if and only if the list is empty
 * @pre true
 * @post @result <=> size() > 0
 * @post @nochange
 */
Public boolean isEmpty() {
}

/**
 * Returns the i-th element in the list
 * @pre I >= 0 && I < size()
 * @post @nochange
 */
public Object element(int i) {
}
```

head() and last() Methods

```java
/**
 * Returns the first element in the list
 * @pre !isEmpty()
 * @post @result == element(0) > 0
 * @post @nochange
 */
Public Object head() {
}

/**
 * Returns the last element in the list
 * @pre !isEmpty()
 * @post @result == element(size() - 1) > 0
 * @post @nochange
 */
Public Object last() {
}
```

insert() Methods

```java
/**
 * Inserts new element into the list at the i-th position
 * @pre item != null && i >= 0 && i <= size()
 * @post size() == size()+1
 * @post @forall k : [0..size()-1] @
 *              (k < i ==> element(k)@pre == element(k)) &&
 *              (k >= i ==> item@pre == element(k)) &&
 *              (k > i ==> element(k-1)@pre == element(k))
 */
Public void insert(Object item, int i) {
}
```

head() and last() Methods

```java
/**
 * Returns the first element in the list
 * @pre !isEmpty()
 * @post @result == element(0) > 0
 * @post @nochange
 */
Public Object head() {
}

/**
 * Returns the last element in the list
 * @pre !isEmpty()
 * @post @result == element(size() - 1) > 0
 * @post @nochange
 */
Public Object last() {
}
```

insert() Methods

```java
/**
 * Inserts new element into the list at the i-th position
 * @pre item != null && i >= 0 && i <= size()
 * @post size() == size()+1
 * @post @forall k : [0..size()-1] @
 *              (k < i ==> element(k)@pre == element(k)) &&
 *              (k >= i ==> item@pre == element(k)) &&
 *              (k > i ==> element(k-1)@pre == element(k))
 */
Public void insert(Object item, int i) {
}
```
insertHead() and insertLast() Methods

```java
public void insertHead(Object item) {
    // code
}
```

```java
public void insertLast(Object item) {
    // code
}
```

Invariants of Classes

- The state of an object is transient if it is being manipulated.
- The state of an object is stable if it has been initialized and is not being manipulated.
- An invariant of a class is a formally specified condition that always holds on any object of the class whenever it is in a stable state.
- Given the invariants of a class, an object of the class is in a well-formed state if the invariants hold on the state.

Example: LinkedList Class

- If the list is empty, both head and tail should be null.
- If the list is not empty, the head field points to the first node in the list, and the tail field points to the last node in the list.
- The count field should be equal to the number of nodes that are reachable by following the next link from the head of the list, i.e., the node pointed to by the head field.
- For each node that is reachable from the head of the list, the following conditions hold:
  - The prev field points to the preceding node in the list, and the next field points to the succeeding node in the list.
  - The prev field of the first node in the list is null, and the next field of the last node in the list is null.

_wellformed() Method

```java
protected boolean _wellformed() {
    int n = 0;
    for (Node p = head; p != null; p = p.next) {
        n += 1;
        if (p.prev != null) {
            if (p.prev.next != p)
                return false;
        } else {
            if (head != p)
                return false;
        }
        if (p.next != null) {
            if (p.next.prev != p)
                return false;
        } else {
            if (tail != p)
                return false;
        }
    }
    return n == count;
}
```

@invariant Tag

```java
/**
 * @invariant _wellformed()
 */
public class LinkedList implements List {
    // ...
    // implementation of linked list
    protected boolean _wellformed() {
        // ...
        return true;
    }
}
```

Design Guideline: Preserving Invariants

- Establishing invariants by public constructors: For each public constructor of the class, the invariant must be the postcondition of the constructor or implied by its postcondition.
- Preserving invariants by public method: For each public method of the class, the invariant can be assumed to be a precondition of the method and the invariant must be the postcondition of the method or implied by its postcondition.
Assertions

- An assertion is a Boolean condition at a given location of a program that should be true whenever the flow of the execution reaches that location.
- Java supports assertion statements that will be checked at run time.
  - If assertion is true, the statement has no other effects.
  - If assertion is false, an AssertionError exception is thrown.

Example head() implementation

```java
/**
 * Returns the first element in the list.
 * @pre !isEmpty()
 * @post @result == element(0)
 */
public Object head() {
    assert !isEmpty();
    Object result = (head != null ? head.item : null);
    assert result == element(0);
    return result;
}
```

Example insert() implementation

```java
/**
 * Inserts a new element into the list.
 * @pre item != null && i >= 0 && i <= size()
 * @post size() == size() + 1
 */
public void insert(Object item, int i) {
    assert item != null && i >= 0 && i <= size();
    assert _wellformed();
    int size_pre = size();
    if (i <= 0) {
        insertHead(item);
    } else if (i >= count) {
        insertLast(item);
    } else {
        // i > 0 && i < count;
        Node n = head;
        for (int j = 0; n != null && j < i - 1; j++) {
            n = n.next;
        }
        Node node = new Node();
        node.item = item;
        node.next = n.next;
        node.prev = n;
        node.next.prev = node;
        n.next = node;
        count++;
    }
    int size_post = size();
    assert size_post == size_pre + 1;
    assert _wellformed();
}
```

Design Guideline:
Use Assertions Aggressively

- Each method should include assertions on the preconditions and postconditions of the method and invariants of the class.

Design Guideline:
Design by Contract

- Each method in a class must honor the contract of the respective method in its superclass and/or the interfaces implemented by the class.
  Specifically:
  - The precondition of each method in a subclass must not be stronger (be restrictive) than the precondition of the respective method in its superclass and/or interfaces implemented by the class.
  - The postcondition of each method in a subclass must not be weaker (do less) than the postcondition of the respective method in its superclass and/or interfaces implemented by the class.
The Canonical Form of Classes

The canonical form of public classes ensures that instances of the classes will be well behaved when they are manipulated by Java run-time environment and other classes.

Design Guideline: Canonical Form of Public Classes

- **No-arg constructor**: Providing a public no-arg constructor
- **Object equality**: Overriding the `equals()` and `hashCode()` methods.
- **String representation**: Overriding the `toString()` method.
- **Cloning**: Implementing the `Cloneable` interface, and overriding the `clone()` method.
- **Serializiation**: Implementing the `java.io.Serializable` interface and overriding the `readObject()` and `writeObject()` methods, when the instances of the class may need to be saved in files or transferred over the network.

No-Argument Constructor

- Allow instances of the class to be created dynamically by JVM at run time.
- The class will made available at run time.
- Many Java applications and frameworks depend on the ability to load classes and create instances at run time.

Object Equality

- `equals()` method in the `Object` class is used to test the identity of objects
  
  ```java
  object1.equals(object2)
  ```

  Most classes should override this method with the notion of equality based on the contents.

The contract of the `equals()`

- **Reflexivity**: For any object `x`, `x.equals(x)` is always true.
- **Symmetry**: For any objects `x` and `y`, `x.equals(y)` is true if and only if `y.equals(x)` is true.
- **Transitivity**: For any objects `x`, `y` and `z`, if both `x.equals(y)` and `y.equals(z)` are true then `x.equals(z)` is true.
- **Consistency**: For any objects `x` and `y`, `x.equals(y)` should consistently return true or consistently return false, if the state of `x` and `y` are unchanged.
- **Normality**: For any object `x`, `x.equals(null)` should always be false.

Template of a Typical implementation of `equals()`

```java
public boolean equals(Object other) {
    if (this == other) {
        return true;
    }
    if (other instanceof C) {
        C otherObj = (C) other;
    <compare each field, and return false if not equal>
        return true;
    }
    return false;
}
```

- For a field `p` of primitive type
  ```java
  if (p == otherObj.p) {
      return true;
  }
  return false;
  ```

- For a field `r` of reference type
  ```java
  if (r == null && otherObj.r == null) {
      return true;
  }
  return false;
  ```

```java
return false;
```
public boolean equals(Object other) {
    if (this == other) return true;
    if (other instanceof List) {
        List otherList = (List) other;
        if (this.getCount() == otherList.getCount()) { // same length
            for (int i = 0; i < this.getCount(); i++) {
                Object thisElement = this.elementAt(i);
                Object otherElement = otherList.elementAt(i);
                if (thisElement == null) {
                    if (otherElement != null) {
                        return false;
                    } else { // equal element
                        if (!thisElement.equals(otherElement)) {
                            return false;
                        }
                    }
                }
            }
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}

Equals() Method of LinkedList

public boolean equals(Object other) {
    if (other != null &&
        other instanceof LinkedList) {
        LinkedList otherList = (LinkedList) other;
        if (this.size() == otherList.size()) {
            Node thisNode = this.head;
            Node otherNode = otherList.head;
            while (thisNode != null && otherNode != null) {
                if (!thisNode.item.equals(otherNode.item)) {
                    return false;
                }
                thisNode = thisNode.next;
                otherNode = otherNode.next;
            }
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}

Hash Code of Objects

• The `hashCode()` method is used by
collection classes that implement
hash tables such as `HashMap` and
`HashSet`

• Overriding the `equals()` method
requires overriding the `hashCode()`
method also

The Contract of the `hashCode()`

• If two objects are equal according to
the `equals()` method, they must
return the same hash code:

\[ x.equals(y) \Rightarrow x.hashCode() == y.hashCode() \]

Hash Code

• Computing a hash code for each
significant field.

• Combining the hash codes of all
significant fields. The key is to
include all the hash codes while
computing the final hash code.

Cloning Objects

• The `clone()` method returns a clone
of the object itself.
The Contract of the clone()

- The cloned object must not be the same object as the original
  \[ \text{object.clone()} \neq \text{object} \]
- The cloned object and the original object are instances of the same class.
- The cloned object must be equal to the original object
  \[ \text{object.clone().equals(object)} \]

clone() Method of Object Class

- The clone() method in the Object class is protected
- The implementation
  - For a class that implements the marker interface Cloneable, it creates a shallow copy of the original object.
  - For a class that does not implement the marker interface Cloneable, it throws a CloneNotSupportedException.
- A shallow copy means that the value of each field of the original object is copied to the corresponding field of the cloned object.
- For fields of reference types, only the references are copied. The actual objects referenced by the fields are not cloned.
- A deep copy means the actual objects referenced by the fields of reference types are cloned.

clone() of LinkedList for Deep Copy

```java
public class LinkedList implements List, Cloneable {
    / ...
    public Object clone() throws CloneNotSupportedException {
        LinkedList list = (LinkedList) super.clone();
        list.head = list.tail = null;
        list.count = 0;
        for (Node node = head; node != null; node = node.next) {
            if (node.item != null) {
                list.insertLast(node.item);
            }
        }
        return list;
    }
}
```

Shallow Copy and Deep Copy

```
1
2
3
```

Using Clones in Assertions

```java
/**
 * Inserts a new element into the list at the i-th position.
 * @param item new element
 * @param i insertion index
 * @pre item != null && i >= 0 && i <= size()
 * @post size() == size()+1
 * @post @forall k : [0 .. size() - 1] @ (k < i ==> element(k) == element(k)) &&
 *         (k == i ==> item == element(k)) &&
 *         (k > i ==> element(k - 1) == element(k))
 */
public void insert(Object item, int i) {
    // assert the pre-condition
    assert item != null && i >= 0 && i <= size();
    // assert the invariant
    assert _wellformed();
    // the object in pre-state, used in the post-conditions
    int size_pre = size();
    LinkedList this_pre = null;
    try {
        this_pre = (LinkedList) clone();
    } catch (CloneNotSupportedException e) {
    }
    // assert the post-condition
    int new_size = size() + 1;
    int new_count = count() + 1;
    // assert the wellformed
    assert _wellformed();
    // assert the post-condition
    assert size() == new_size && count() == new_count;
    // print the pre-state
    System.out.println("pre:");
    System.out.println(_toString(this_pre, size_pre));
    // print the post-state
    System.out.println("post:");
    System.out.println(_toString(this, count()));
    // check the post-condition
    check(size(), size(), count(), count());
}
```

String Representation of Objects

```java
public String toString() {
    StringBuffer s = new StringBuffer();
    int i = 0;
    for (Node n = head; n != null; n = n.next, i++) {
        s.append("[" + i + "] = " + n.item + 
    }
    return s.toString();
}
```
Serialization

- **Serialization** is the process of transforming an object into a stream of bytes.
- **Deserialization** is the reverse process of serialization.
- A default implementation for serialization and deserialization is provided for all classes that implement the `java.io.Serializable` interface.
- `readObject()` and `writeObject()` can be used to customize the way instances of classes are serialized.

Unit Testing

- **Unit testing**: test each unit or component independently before the units are integrated into the whole system.
- **Integration and system testing**: integrate all the units and test the system as a whole.
- **Acceptance testing**: validate that the system functions and performs as expected.

Testing Coverage Criteria

- Systematically test all aspects of the implementation based on certain established criteria.
- Automatically check the correctness of the results.

Test cases

- **Black box**: based on specification of a component alone without the implementation.
  - Equivalence partitioning of input space.
- **White box**: based on the structure of the code implementing the component.
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Combination condition coverage.

Simple Unit Testing

```java
package test;
import mylist.*;
public class LinkedListTest {
  public static void main(String args[]) throws CloneNotSupportedException {
    LinkedList l = new LinkedList();
    l.insertHead(new Integer(1));
    l.insertHead(new Integer(2));
    l.insertLast(new Integer(3));
    l.insertLast(new Integer(4));
    l.insert(new Integer(5), 3);
    l.insert(new Integer(6), 3);
    l.insert(new Integer(7), 3);
    System.out.println("First pass");
    System.out.println(l);
    l.removeHead();
    l.removeLast();
    l.remove(2);
    System.out.println("Second pass");
    System.out.println(l);
    LinkedList l2 = (LinkedList) l.clone();
    System.out.println("Cloned list");
    System.out.println(l2);
    l2.removeHead();
    System.out.println("Original list");
    System.out.println(l);
    System.out.println("Cloned list");
    System.out.println(l2);
  }
}
```
if (!TestUtil.match(l, TestUtil.toIntegerArray(results[1]))) {
    System.out.println("Result mismatch");
    testPassed = false;
}

LinkedList l2 = (LinkedList) l.clone();
System.out.println("Cloned list");
System.out.println(l2);
if (!TestUtil.match(l2, TestUtil.toIntegerArray(results[2]))) {
    System.out.println("Result mismatch");
    testPassed = false;
}

l2.removeHead();
System.out.println("Original list");
System.out.println(l);
if (!TestUtil.match(l, TestUtil.toIntegerArray(results[3]))) {
    System.out.println("Result mismatch");
    testPassed = false;
}

if (testPassed) {
    System.out.println("Test passed.");
} else {
    System.out.println("Test failed.");
}

TestUtil
package test;
import mylist.*;
public class TestUtil {
    
    public static boolean match(List list, Object[] array) {
        boolean result = false;
        if (list != null && array != null) {
            int n = list.size();
            if (n == array.length) {
                for (int i = 0; i < n; i++) {
                    Object item = list.element(i);
                    if (item != null) {
                        if (!item.equals(array[i])) {
                            return false;
                        }
                    } else {
                        if (array[i] != null) {
                            return false;
                        }
                    }
                }
                result = true;
            } else if (list == null && array == null) {
                result = true;
            }
        }
        return result;
    }

    public static Object[] toIntegerArray(int[] intArray) {
        if (intArray != null) {
            int n = intArray.length;
            Object[] resultArray = new Object[n];
            for (int i = 0; i < n; i++) {
                resultArray[i] = new Integer(intArray[i]);
            }
            return resultArray;
        } else {
            return null;
        }
    }

    ...
}

JUnit -- Unit Testing Tools

JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is used by the developer who implements unit tests in Java.

JUnit is Open Source Software.
http://www.junit.org/

Ant – A Build Tool

Apache Ant is a Java-based build tool.
http://ant.apache.org/