CSE 12:

Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Five
8 Aug 2011

More on interfaces.

Review of `Iterable` and `Iterator` interfaes

- Recall: If a class `X` is to implement the `Iterable` interface, then it must implement a method called `iterator()` which returns an object of type `Iterator`.
- `Iterator` itself is an interface, not a class; hence, `X.iterator()` can return an object of any class that implements the `Iterator` interface.

Iterator interface

- A class can implement the `Iterator` interface if it implements the following three methods:

```
// Returns true if the iteration has more
// elements.
boolean hasNext ();

// Returns the next element in the iteration.
Object next ();

// Removes from the underlying collection the
// last element returned by the iterator
// (optional operation).
void remove ();
```

In the case of the `List12` interface: "The `Iterator` returned by `iterator()` must support the `remove()` method."

`Iterator` interface

- The user of the `Iterator` can call these methods whenever he/she wants, subject to the following constraints (as defined in the `Iterator` interface) on `remove()`:
  - This method can be called only once per call to `next`.
  - `remove()` should throw an `IllegalStateException` if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.

`Iterator` interface

- The `Iterator` interface also specifies that "the behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method."

`Iterator` interface

Unspecified means that you are "absolved of any responsibility" for maintaining correct functionality in the `Iterator` if the user modifies the `DoublyLinkedList12` while he/she is iterating over it.

- The `Iterator` interface also specifies that "the behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method."

Modifications in the case of `DoublyLinkedList12` mean `addToFront()`, `removeFront()`, etc. -- anything that changes the contents of the list.

Interface as a "contract"

- An interface specification serves as a *contract* between user and implementor of the interface.
- The method signatures specify to the user what each method does, and how it is called (i.e., parameters).
- The comments describe to the implementor what each method must do and what values to return.

Interface as a "contract"

- The comments may also prescribe to the user various *constraints* on how the methods are called, e.g., "**`next()`** must be called before **`remove()`**.
- If the user does not adhere to these constraints, then he/she is in *violation of contract.*
- If the user violates the contract, then the implementor may:
  - Throw an exception (e.g., **`InvalidStateException`**).
  - Be "absolved of responsibility" to keep working correctly ("behavior is...unspecified").
    - E.g., calls to **`next()`**/**`remove()`**/**`hasNext()`** may stop working correctly, and this is *no longer the implementor's fault.*

Java interfaces

- Java facilitates the division-of-labor between user and implementor using interfaces.
- An interface contains no method bodies and no instance variables.
- An interface may, however, contain static constants, e.g.,

```
interface List {
  static final int INITIAL_CAPACITY = 100;
  ...
}
```

- Any class that implements an interface automatically can also access the interface's static variables.

Static variables in interfaces

- Example:

In my coding style, I use all-capital variable names to indicate a static constant.

```
final ArrayList list = new ArrayList();
System.out.println(
  "Initial capacity of list is " +
  ArrayList.INITIAL_CAPACITY
);
```

INITIAL_CAPACITY is a class variable, not an instance variable; hence, we specify the class, not an object.

Java interfaces

- In Java, a class may implement *any number* of interfaces as long as it defines method bodies for all methods whose signatures appear in those interfaces, e.g.:

```
interface A {
  // Do something
  void a ();
}
```

```
interface B {
  // Return something
  int b ();
}
```

```
class X implements A, B {
  void a () { System.out.println("a!"); }
  int b () { return 123; }
}
```

- Note that the programmer must explicitly write "implements" -- just implementing the methods themselves is not enough.

Java interfaces

- Example of class implementing multiple interfaces:

```
class String implements Comparable, Serializable {
  ...
}
```

- A **`Comparable`** object is one that can be compared (using **`compareTo`**) to other objects (e.g., **`"str1".compareTo("str2")`**).
  - Useful for *sorting* a list of objects.
- A **`Serializable`** object is one that can be converted into a **`byte[]`** using the **`serialize`** method (recall the Google Earth example from Lecture 1).

Java interfaces

- Implementing multiple interfaces places no constraints on the class structure of the implementing class:
  - E.g., `String` doesn't have to "inherit" from some "`Serializable`" class.
- This gives flexibility to the implementor -- he/she can subclass whatever class he/she wants (if any).

Subinterfaces

- In Java, an interface **`Y`** can “subinterface” another interface **`X`**.
  - This is analogous to a class **`B`** subclassing another class **`A`**.
- An interface **`Y`** that is a subinterface of **`X`** automatically contains all of **`X`**’s method signatures and static variables.

Subinterfaces

- Example:

```
interface X {
  void method ();
}
interface Y extends X {
  void anotherMethod ();
}
```

- Interface **`Y`** implicitly contains **`method`** as well.
  - Hence, if class **`C`** implements **`Y`**, then it must implement both **`method`** and **`anotherMethod`**.

Interfaces as types

- In Java, an interface can serve as the type in a variable declaration, e.g.,

  ```
  List12 list;
  ```

- The `list` variable can be initialized to any class that implements the `List12` interface (e.g., `DoublyLinkedList12`).

- However, one cannot instantiate an interface type -- one can only instantiate a *concrete* (non-abstract) class type:

  ```
  List12 list = new DoublyLinkedList12();  // ok
  List12 list = new List12();  // not ok
  ```

Interfaces as types

- Recall that an object of class **B** can be referenced by a variable declared of class **B** or any parent class of **B**, e.g.:

```
class A { ... }
class B extends A { ... }

B b = new B();
A a = b;  // ok -- A is parent class of B
```

Interfaces as types

- An object of class **B** can also be referenced by a variable declared of any interface type that **B** implements, e.g.:

```
interface X { ... }
interface Y { ... }
interface Z { ... }
class A implements X { ... }
class B extends A implements Y { ... }

B b = new B();
X x = b;  // ok -- B extends A, and A implements X
Y y = b;  // ok -- B implements Y directly
Z z = b;  // not ok -- neither A nor B implements Z
```

Interfaces as types

- Why would you care about being able to refer to a **`DoublyLinkedList12`** as an **`Iterable`**?
  - Because it offers programmers more flexibility, e.g.:

```
void printAllData (Iterable iterable) {
  final Iterator it = iterable.iterator();
  while (it.hasNext()) {
    System.out.println(it.next());
  }
}
```

Interfaces as types

- Why would you care about being able to refer to a `DoublyLinkedList12` as an `Iterable`?
  - Because it offers programmers more flexibility, e.g.:

```
void printAllData (Iterable iterable) {
  final Iterator it = iterable.iterator();
  while (it.hasNext()) {
    System.out.println(it.next());
  }
}
```

The implementor of `printAllData` doesn't care if the argument passed in is a `DoublyLinkedList12`, `ArrayList12`, or even a `List12` -- he/she *only cares that it supports the* `iterator` *method.*

Interfaces versus superclasses

- Some languages, such as C++, offer no support for interfaces -- they only offer classes.
- In C++, if you wanted a type `Iterable` that guaranteed all objects of that type supported an `iterator()` method, then `Iterable` would have to be a *class*.
    - This means that any object `o` passed to `printAllData` would have to be of a class `C` that subclasses `Iterable`.
        - This is less flexible than in Java.
        - But C++ offers multiple inheritance instead. <== complex!

Abstract classes.

Abstract classes

- In addition to interfaces, Java also supports abstract classes.
- In contrast to a "concrete" class, an abstract class does *not* have to implement *all* of its methods.
  - It must simply list their method signatures (similarly to an interface) and define those methods to be **`abstract`**.
- An abstract class can, however, implement *some* of its methods.

Abstract classes

- Example:

A class with at least one abstract method must be declared **`abstract`**.

```
abstract class BasicArrayList {
  final Object[] _underlyingStorage;
  BasicArrayList () {
    _underlyingStorage = new Object[128];
  }
  ...
  public abstract void sort ();
}
```

An *abstract method* contains no body.

Abstract classes

- Example:

An abstract class may contain a constructor, instance variables, as well as "concrete" methods (methods with bodies).

```
abstract class AbstractArrayList {
  final Object[] _underlyingStorage;
  BasicArrayList () {
    _underlyingStorage = new Object[128];
  }
  ...
  public abstract void sort ();
}
```

Abstract class example

- Abstract classes are useful when several subclasses in a class hierarchy have substantial code in common, e.g.:
  - For a drawing program, classes **`Rectangle`**, **`Ellipse`**, and **`Line`** may all inherit from a common **`Shape`** superclass.
  - All classes in the hierarchy should support a **`getColor()`** method.
  - No point in copying+pasting code through all three subclasses -- just implement once in **`Shape`** class.

Abstract class example

- However, a **`Shape`** object cannot draw itself because *it doesn't know what kind of shape it is.*
  - Hence, we make the **`draw()`** method abstract -- we delay implementing this method until we subclass the **`Shape`** class.

Abstract class example

```
abstract class Shape {
  Color _color;
  Color getColor () {
    return _color;
  }
  abstract void draw ();
}
class Rectangle extends Shape {
  void draw () {
    // Actually draw the rectangle
   ...
  }
}
class Ellipse extends Shape {
  void draw () {
    // Actually draw the ellipse
    ...
  }
}
```

Abstract classes as types

- Like interfaces, abstract classes in Java can be used as types, but cannot be instantiated directly:

```
AbstractArrayList list =
  new SomeConcreteArrayList();   // ok
list = new AbstractArrayList();  // not ok
```

Abstract classes

- In order to be useful, abstract classes must be subclassed by "concrete" classes, i.e., classes that implement all the abstract methods, e.g.:

```
class SomeConcreteArrayList extends AbstractArrayList
{
  ...
  public void sort () {  // Concrete implementation
    // Sort the data in _underlyingStorage
    // ...
  }
}
```

Interfaces versus abstract classes

- Interfaces and abstract classes can both contain method signatures without bodies.
- Classes can be subclassed; interfaces can be "subinterfaced".
- An abstract class is allowed to implement some methods; an interface can *never* implement any of them.
- An abstract class can contain instance variables; an interface cannot.

Interfaces versus abstract classes

- Abstract classes and interfaces are both useful when creating multiple implementations of the same "abstract idea" (e.g., a list, a collection, a shape).
- When should one use an interface versus an abstract class?
  - Abstract classes are useful when there is substantial code (i.e., implemented methods) or data (i.e., instance variables) that all subclasses should inherit.
  - Otherwise, interfaces should generally be used because they are more flexible.

Inner classes.

Inner classes

- Java offers the ability to define a class within another class, e.g.:

```
class A {
  int _x;

  ...

  static class B {
    Object _o;
  }
}
```

Static inner class

or

```
class A {
  int _x;

  ...

  class B {
    Object _o;
  }
}
```

Non-static inner class

- Static and non-static inner classes have slightly different *semantics*.

Static inner classes

- A static inner class **`B`** inside class **`A`** is similar to a completely separate class **`B`**, e.g.:

```
class A {
}
class B {
}
```

- However, in contrast to separately defined classes, the instance variables of inner class **`B`** are always accessible to outer class **`A`**, *even if they are private*, e.g.:

```
class A {
  class B {
    private int _x;
  }
  void method () {
    final B b = new B();
    b._x = 7;  // This works!
  }
}
```

Static inner classes

- There are several reasons for using a static inner class:

  1. To provide convenience to class **`A`** to access **`B`**'s private instance variables, but *prevent all other classes from doing so*.
  2. To structure your code to emphasize a tight coupling between **`A`** and **`B`**.
  3. To prevent outside classes from accessing/instantiating class **`B`**. In this case, we can make **`B`** a *private inner* class.

Static inner classes: example

- Consider making the `Node` class a static inner class of `DoublyLinkedList12`:
  - The `Nodes` and the `DoublyLinkedList12` are tightly coupled:
    - Without the `Nodes`, the `DoublyLinkedList12` class cannot be implemented.
    - Without the `DoublyLinkedList12`, the `Nodes` have little relevance.

Static inner classes: example

- It will be convenient for the **`DoublyLinkedList12`** to access the **`Node`**s' instance variables directly.
- We may also wish to make the **`Node`** inner class private.
  - We don't want any external class dealing with **`Node`**s.
  - From the user's perspective, the **`Node`** class is irrelevant; we should hide this detail from the user.

Instantiating objects of static inner classes

- Objects of type **B**, where **B** is a static inner class of **A**, can be instantiated as:

```
class A {
  static class B {
  }
  void method () {
    B b = new B();
  }
}
```

or (from an external class) as:

```
class C {
  void otherMethod () {
    A.B b = new A.B();
  }
}
```

Not possible if B is defined to be *private*.

Non-static inner classes

- Non-static inner classes offer an even "tighter coupling" of instances of the inner class and an instance of the outer class.
- An instance of a *non-static* inner class **B** can access the private instance variables of the *outer* class **A**, e.g.:

```
class A {
  private int _num = 5;
  class B {
    void m () {
      _num = 6;  // Ok! -- Inner classes can access
                 // private variables of outer class.
    }
  }
}
```

Non-static inner classes

```
class A {
  private int _num = 5;
  class B {
    void m () {
      _num = 6;  // Ok! -- Inner classes can access
                 // private variables of outer class.
    }
  }
}
```

- What does this really mean?
- Of *which instance* of **`A`** does method **`m()`** alter the **`_num`** instance variable?

Non-static inner classes

```
class A {
  private int _num = 5;
  class B {
    void m () {
      _num = 6;  // Ok! -- Inner classes can access
                 // private variables of outer class.
    }
  }
}
```

- What does this really mean?
- Of *which instance* of **A** does method **m()** alter the **_num** instance variable?
  - The *enclosing instance*. This is the instance of **A** that the instances of **B** are "linked to" via an *implicit reference*.

Non-static inner classes

- Consider inner class **B** inside of **A**:

```
class A {
  int _num = 5;
  class B {
    String _s;
    B (String s) {
      _s = s;
    }
    void m () {
      _num = 17;
    }
  }

  public void n () {
    final B b1 = new B("inst1");
    final B b2 = new B("inst2");
  }
}
```

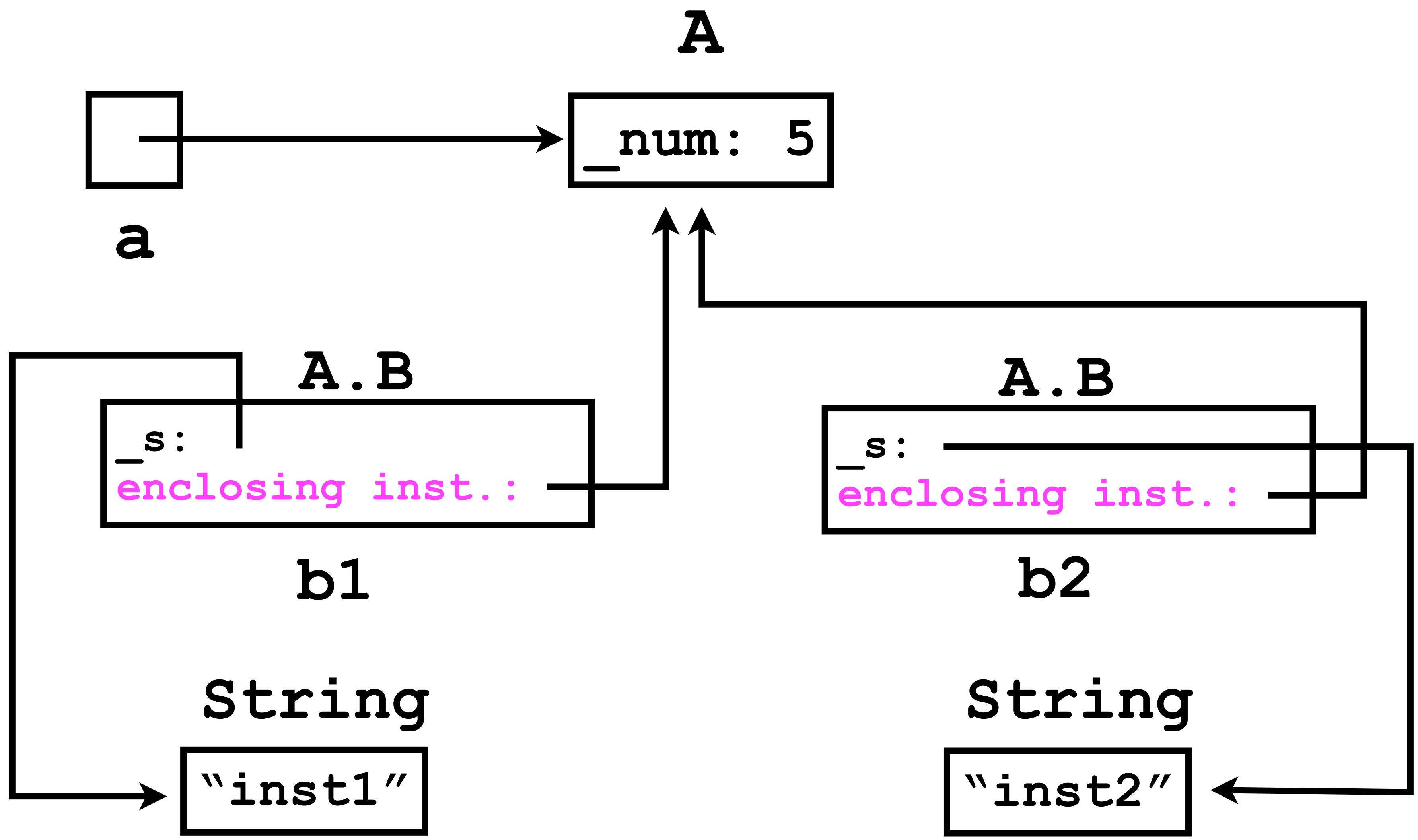
- Now, consider code from some other class **C**:

```
...

final A a = new A();
a.n();

...
```

- How are objects `a`, `b1`, and `b2` related in memory?




```
class A {
  int _num = 5;
  class B {
    String _s;
    B (String s) {
      _s = s;
    }
    void m () {
      _num = 17;
    }
  }

  public void n () {
    final B b1 = new B(“inst1”);
    final B b2 = new B(“inst2”);
  }
}
```

Although not visible in the class declaration, each **B** contains an implicit reference to the enclosing instance of **A**.

```
...

final A a = new A();
a.n();

...
```

Static versus non-static inner classes

- In contrast to non-static inner classes, it makes no sense to try to instantiate an object of the inner class without an enclosing instance of the outer class. For example, from an external class **C**:

```
class C {
  void otherMethod () {
    A.B b = new A.B();  // Will not compile
  }
}
```

- In this context, there is no instance of **A**; hence, an instance of **B** has no "enclosing instance".
  - This code will not compile.

Static versus non-static inner classes

- When to use static versus non-static inner classes?
  - Use *non-static* inner classes if the instances need to reference instance variables of the outer class.
  - Otherwise, use *static* inner classes -- they are faster and take less memory.

Anonymous classes.

Anonymous classes

- There's one more kind of class in Java -- *anonymous classes*.
- Anonymous classes are useful when you intend to instantiate only *one* instance of the class, *ever*.
- When would this situation arise?

Anonymous classes: example

- One particular use of anonymous classes in Java is for callbacks.
  - A *callback* is a method **`m`** that you pass to another method with a request to call **`m`** at some later time.
- Consider the class **`java.util.Timer`**:
  - Timers are useful for scheduling an event in the future, perhaps at a regular interval.
  - For example, in the Confetti simulator, the positions/velocities of all particles are updated every 5 msec.

Anonymous classes: example

- To use a **`Timer`**, one has to create a callback object of type **`TimerTask`**.
- **`TimerTask`** is an abstract class that contains an abstract method **`void run()`**.
- The user of a **`Timer`** will implement a concrete class that subclasses **`TimerTask`**.
  - The user's implementation of **`run()`** will perform whatever task the user wants.

Anonymous classes: example

- Instead of declaring a whole new class -- either in its own file, or as an inner class -- we can be even more "compact" and define an anonymous class:

```
java.util.Timer timer = new java.util.Timer();
timer.schedule(new TimerTask() {
  public void run () {
    // Do whatever you want
  }
}, 0, 5);  // 5 msec
```

This defines an *anonymous class* that extends **`TimerTask`**.

Anonymous classes: example

- Without anonymous classes, we'd have to be more verbose:

```
class MyTimerTask extends TimerTask {
  public void run () {
    public void run () {
      // Do whatever you want
    }
  }
}

java.util.Timer timer = new java.util.Timer();
timer.schedule(new MyTimerTask(), 0, 5);
```

This is the only instance we will ever create.

Implementing an `Iterator`.

Iterator for `ArrayList`

- Given this new "infrastructure" for writing object-oriented Java code, let's implement an `Iterator` for the `ArrayList` we created in previous lectures.

Iterator for **`ArrayList`**

- We need to create a class to implement the Iterator -- let's call it **`ArrayListIterator`**.
- The **`ArrayListIterator`** and the **`ArrayList`** are coupled:
  - The **`ArrayList`** needs to return an instance of **`ArrayListIterator`**.
  - An **`ArrayListIterator`** should never be instantiated outside of **`ArrayList`**.
    - *No other class needs to know it exists.*

Iterator for `ArrayList`

- We need to create a class to implement the Iterator -- let's call it `ArrayListIterator`.
- The `ArrayListIterator` and the `ArrayList` are coupled:
  - The `ArrayList` needs to return an instance of `ArrayListIterator`.
  - An `ArrayListIterator` should never be instantiated outside of `ArrayList`.
    - *No other class needs to know it exists.*
  - Hence, let's make it a private inner class.

Static or non-static?

- The **`ArrayListIterator`** needs to access the individual data stored in **`_underlyingStorage`** of the "enclosing" **`ArrayList`**.
- It also needs to be able to modify the enclosing **`ArrayList`** object.
- Hence, we need a *non-static* inner class.

ArrayListIterator

```
class ArrayList implements List {
  private Object[] _underlyingStorage;
  private int _numElements;

  private class ArrayListIterator implements Iterator {
    ... // What variables do we need?
    public boolean hasNext () {
      ...
    }
    public Object next () {
     ...
    }
    public void remove () {
     ...
    }
  }

  public Iterator iterator () {
    return new ArrayListIterator();
  }
}
```

Unannounced quiz 2

`ArrayListIterator`

- Let's define an int `_currentIndex` instance variable to keep track of the next object we should return in `next()`.
- We initialize `_currentIndex` to `-1` to indicate we haven't returned the first element yet.
- Each call to `next()` both increments `_currentIndex` and returns the object `_underlyingStorage[currentIndex]`.
  - Make sure to increment before fetching the object!
- We also need a variable boolean `_hasNextBeenCalled`.

ArrayListIterator.hasNext()

- To implement **hasNext()**, we simply compare **_currentIndex** to **_numElements**, which is contained in the enclosing instance of **ArrayList**.
  - This is only possible because **ArrayListIterator** is an inner class!
  - Without inner classes, we'd have to either make **_numElements** public (bad idea), or create an *accessor* method (verbose).

ArrayListIterator.remove()

- To remove the element we last returned (in **`next()`**), we need to "shift over" all the elements of **`_underlyingStorage`** to the left by 1.
- We implemented this already in **`ArrayList.remove()`**.
  - Hence, we can just call **`ArrayList.remove()`** in **`ArrayListIterator.remove()`**.

ArrayListIterator code

```
class ArrayListIterator {
  private boolean _hasNextBeenCalled = false;
  private int _currentIndex = -1;

  public Object next () {
    _currentIndex++;
    _hasNextBeenCalled = true;
    return _underlyingStorage[_currentIndex];
  }
  public boolean hasNext () {
    return _currentIndex < _numElements - 1;
  }
  public void remove () {
    if (! _hasNextBeenCalled) {
      throw new InvalidStateException();
    }
    ArrayList.this.remove(_currentIndex);
    _currentIndex--;
    _hasNextBeenCalled = false;
  }
}
```

ArrayListIterator code

```
class ArrayListIterator {
  private boolean _hasNextBeenCalled = false;
  private int _currentIndex = -1;

  public Object next () {
    _currentIndex++;
    _hasNextBeenCalled = true;
    return _underlyingStorage[_currentIndex];
  }
  public boolean hasNext () {
    return _currentIndex < _numElements - 1;
  }
  public void remove () {
    if (! _hasNextBeenCalled) {
      throw new InvalidStateException();
    }
    ArrayList.this.remove(_currentIndex);
    _currentIndex--;
    _hasNextBeenCalled = false;
  }
}
```

Make sure initialization and increment of **_currentIndex** cooperate properly.

Call **remove()** method of *outer class* on the *enclosing instance*.

Make sure next call to **next()** works properly.

Adhering to contract

- What will happen in the following code?

```
final ArrayList arrayList = new ArrayList();
arrayList.add(new Integer(123));
final Iterator iterator = arrayList.iterator();
arrayList.remove(0);
final Object obj = iterator.next();
```

In the particular case of `ArrayList` (given how we implemented it), this call to next() will actually be benign -- it will return the `Integer` object (123) that we initially inserted.

However, in general, calling `next()` after modifying the underlying container class could have unpredictable effects, e.g., a `NullPointerException` or `IndexOutOfBoundsException`. It is best to *guard against these.*

Concurrent modification

- In this case, the user made the mistake of *concurrent modification*.
- According to `Iterator` specification, once an iteration is in progress, only the `Iterator.remove()` method may be used to modify the list.
- Hence, the `IndexOutOfBoundsException` is not the implementor's fault.

Concurrent modification

- However, to be a "more friendly" implementor, we can help the user identify his/her error by *guarding* against this condition.
- We will keep track of any changes the user makes to the **`ArrayList`** while iteration is underway.
  - We can add a variable **`int _numModifications`** to the outer **`ArrayList`**.
    - We increment this counter whenever the user modifies from the **`ArrayList`** (in **`ArrayList.add()`**, **`ArrayList.remove()`**).

Concurrent modification

- We also add `int _expectedNumModifications` to the `ArrayListIterator` itself.
  - Initialize upon construction to `_numModifications` (current value of instance variable of outer `ArrayList` class).
  - In `next()`, we check whether `_numModifications == _expectedNumModifications`.
  - Have to adjust `_expectedNumModifications` in `ArrayListIterator.remove()`!

Concurrent modification

- Here's the punchline:
  - If, in **`ArrayListIterator.next()`**, we find that **`_expectedNumModifications != _numModifications`**, then we throw a **`ConcurrentModificationException`**.
  - *This informs the user explicitly that he/she messed up.*

```
class ArrayList {
  Object[] _underlyingStorage;
  int _numElements;
  int _numModifications;

  ArrayList () {
    _underlyingStorage =
      new Object[128];
    _numElements = 0;
    _numModifications = 0;
  }
  void add (Object o) {
    ...
    _numModifications++;
  }
  void remove (int index) {
    ...
    _numModifications++;
  }

  class ArrayListIterator
    implements Iterator {
    int _currentIdx;
    boolean _hasNextBeenCalled;
    int _expectedNumModifications;

    ArrayListIterator () {
      _currentIdx = 0;
      _hasNextBeenCalled = false;
      _expectedNumModifications =
        _numModifications;
    }
    boolean hasNext () {
      ...
    }
    Object next () {
      if (_numModifications !=
          _expectedNumModifications) {
        throw new CMException();
      }
      ...
    }
    void remove () {
      if (_numModifications !=
          _expectedNumModifications) {
        throw new CMException();
      }
      ...
      ArrayList.this.remove(index);
      _expectedNumModifications++;
    }
  }
}
```

Abbreviation just for slides!

Calling `remove()` is a *valid* way to modify the list during iteration -- we must account for this.

Save a local copy (inside the `Iterator`) of what `_numModifications` *should* be.

`ArrayList` implement `Iterator` directly?

- Instead of implementing `Iterator` inside a non-static inner class of `ArrayList`, we could instead augment `ArrayList` with `hasNext()`, `next()`, and `remove()` methods.
- We could move the `ArrayListIterator` instance variables to `ArrayList` itself.
- What is the disadvantage of this?

`ArrayList` implement **`Iterator`** directly?

- Disadvantage:
  - The "user" could only use one **`Iterator`** at any given time -- i.e., the following would not work properly:

```
final List list = new ArrayList();
list.add("hello");
list.add("goodbye");
final Iterator iterator = list.iterator();
final Iterator iterator2 = list.iterator();
System.out.println(iterator.next());  // hello
System.out.println(iterator2.next()); // goodbye -- wrong!
```

  - Reason: the **`ArrayList`** would only contain *one* **`_currentIdx`** instance variable.

DoublyLinkedList12 Iterator

- Hopefully the **`ArrayListIterator`** example will provide some guidance for finishing the **`Iterator`** in P1.

- Important case to consider:

```
final List list = new DoublyLinkedList12();
list.add("a");
list.add("b");
list.add("c");
final Iterator iterator = list.iterator();
iterator.next();
iterator.remove();
iterator.next();  // What should this return?
```