# *Inner/Nested Classes*
# *Collection Implementation*

*Thanks to Nick Parlante for much of this handout*

## Inner and Nested Classes

- Suppose you have some "outer" object

- Inner and nested classes allow us to create a small, subsidiary object of the outer object. The inner or nested object is closely integrated with the outer object.

- Appropriate if need the inner object as a temporary, separate feature of the outer object (e.g. iterator)

- Appropriate if we need many little inner/nested objects to go with the one outer object (e.g. binary tree nodes)

## What Does a Function Pointer Accomplish?

- Suppose we have objects `Alice` and `Bob`

- Goal: `Alice` has code. `Alice` gives something to `Bob` that `Bob` can hold, and later on, it allows `Bob` to call `Alice's` code in some way.

- OOP: `Alice` gives `Bob` an object. The object responds to messages `foo()` and `bar()`. `Bob` can hold the object, and later on, call `foo()` and `bar()` on it.

- So A implements the code, gives something to B, where B can invoke the code later.

- Inner classes are widely used to solve this sort of function-pointer problem in Java.

- In Java 7, there are proposals for a "closure" feature, making a simpler syntax for this sort of thing.

## Inner Class

- An "inner" class is a class defined inside some other "outer" class. The inner class may or may not be exposed for use by clients. (See the iterator example below)

- Use an inner class when you need one or more separate objects (like an iterator object) where it makes sense for that object to be closely related to some outer object.

- e.g. A `BinaryTree` class might have an inner `Node` class that it uses internally to build the tree. The `Node` class is probably not exposed to clients -- it's just for internal use.

- The inner class operates like a sub-part of the outer class

- The inner class can have ivars, a ctor, etc. just like a regular class.

- With classes named `Outer` and `Inner`, the full name of the `Inner` class is `Outer.Inner`.

- Access style
    - The outer and inner classes can access each other's state, even if it is **private**. Stylistically, they are basically one implementation code base, so mixing access is ok, but we prefer receiver-relative coding for both the outer and inner classes where it makes sense.

- Inner class usually created in the context of an "owning" outer object. Normally, the "new" call to make the inner is done from the context of a method of the outer object.

    - Calling "new" just anywhere will not work to create an inner object, because of the need for an outer object. The very obscure syntax `outer.new Inner()` can create object of class `Inner` owned by an `Outer` object. I mention this for completeness only. The standard way is to call `new Inner()` inside an outer class method to create new inner objects.

- The inner object automatically has a pointer to its outer object -- can access ivars of outer object automatically

- In the inner class code, `Outer.this` refers to the `this` pointer of the `Outer` object (`Outer` being the name of the outer class)

- Use an inner class if there is a natural need to access the ivars of the outer object, otherwise use a nested class (below)

```
public class Outer {
    private int ivar;

    private class Inner {  // inner class
        private int num; // (could have an Inner() ctor)

        private void foo() {
            num++;            // can access our regular inner ivar
            ivar = 13;        // we can "see" our outer class automatically
            Outer.this.ivar = 13;  // same as above
        }

        public String toString() {
            return "Beat Cal";
        }
    }

    public Object test() {
        ivar = 10;
        Inner in = new Inner();
        in.foo(); // can see things, even if private
        in.toString();  // call an Object method

        return in;
        // Return pointer to inner to our caller as Object.
        // They can call toString() on it.
    }
}
```

## Inner Like Function Pointer

- The above inner class has a simple `toString()` implementation. We can pass a pointer to `Inner` out to some client code as type `Object` (keeping the `Inner` type `private`). The client can call `toString()` on it. In this way, we are using `Inner` like a little function pointer, passed out to the client.

## Nested Class (static)

- Like an inner class, but does not have a pointer to the outer object and so does not automatically access the ivars of the outer object.

- Uses the "**static**" keyword -- that's what distinguishes and inner from a nested class

```
public class Outer {
    private int ivar;

    private static class Nested {
        private int num;
        private void foo() {
            num++; // ok
            // no automatic access to outer ivars
        }
    }

    public void test() {
        Nested nested = new Nested();
        nested.foo();
        ...
    }
}
```

## Inner/Nested Example

```java
// Outer.java
/*
 Demonstrates inner/outer classes.
 Outer has an ivar 'a'.
 Inner has an ivar 'b'.

 Main points:

 -Each inner object is created in the context of a
 single, "owning", outer object. At runtime, the inner
 object has a pointer to its outer object which allows
 access to the outer object.

 -Each inner object can access the ivars/methods
 of its outer object. Can refer to the outer object
 using its classname as "Outer.this".

 -The inner/outer classes can access each other's ivars
 and methods, even if they are "private". Stylistically,
 the inner/outer classes operate as a single class
 that is superficially divided into two.

*/

public class Outer {
    private int a;

    private void increment() {
        a++;
    }

    private class Inner extends Object {
        private int b;
        private Inner(int initB) {
            b = initB;
        }

        private void demo() {
            // access our own ivar
            System.out.println("b: " + b);

            // access the ivar of our outer object
            System.out.println("a: " + a);

            // message send can also go to the outer object
            increment();

            /*
             Outer.this refers to the outer object, so could say
             Outer.this.a or Outer.this.increment()
            */
        }
    }


    // Nested class is like an inner class, but
    // wihout a pointer to the outer object.
    // (uses the keyword "static")
    private static class Nested {
        private int c;

        void demo() {
            c = 11; // this works
            // a = 13; // no does not compile --
            // nested object does not have pointer to outer object
        }
    }
```

```
public void test() {
    a = 10;
    Inner i1 = new Inner(1);
    Inner i2 = new Inner(2);

    i1.demo();   // output: b 1, a 10
    i2.demo();   //         b 2, a 11

    // Obscure syntax to create an inner object
    // not from within an outer method. Consider never
    // doing this.
    Inner i3 = outer.new Inner(6);


    Nested n = new Nested();
    n.demo();

}
```

# Collection Implementation

## AbstractCollection

- A utility class in the java library that implements the convenience methods in the `Collection` interface except the foundation methods: `add()`, `size()`, and `iterator()`, which must be filled in by a real implementing class.

- The easiest way to implement a `Collection` class is to implement the foundation methods, and subclass off of `AbstractCollection` to inherit all the others methods for free.

- Some `AbstractCollection` methods…

- `String toString()`
    - Uses the `Iterator` to print out the elements between square brackets [ ..]

- `boolean contains(Object)`
    - Iterates of the collection to find the given element (`.equals()`)

- `boolean remove(Object)`
    - Iterates of the collection to remove given element if present (`.equals()`). Uses `it.remove()` to remove during iteration.

- `boolean containsAll(Collection)`
    - True if all of the given elements are in the collection (`.equals()`)

- There is also an `AbstractList` class which adds the `List` ideas of a numeric index for `get/set` operations.

## Collection Implementation Strategy

- Subclass off `AbstractCollection<E>`
    - `public class MyCollection<E> extends AbstractCollection<E> {`
    - In generic collections, the conventional name `E` is used to mean "element" instead of the standard `T`. I'm not sure if this convention is a good idea -- it might be simpler to just use "T" for everything.

- Implement the fundamentals: ctor, `size()`, `add()`, and `iterator()`

- The `Iterator<E>` is the most work, since it requires a separate object that responds to `hasNext()`, `next()`, and (optional) `remove()`

- Implement the iterator with an inner class

- The generic "E" or whatever is just a placeholder used when the client compiles. When the collection runs, "E" is essentially just `Object`.

- Implementation hint: Pick a precise meaning for the ivars in the iterator. The pre-advanced convention (below) is probably the best. In any case, it is critical that `size()`, `hasNext()`, `next()`, and `remove()` treat the iterator ivars in a consistent way.

## LameCollection Example

```java
// LameCollection.java
/*
 Demonstrates implementing a simple generic Collection<E>,
 using an inner class for the iterator.
 Stores the elements using a simple E[] array.

 We subclass off AbstractCollection, so we can inherit
 toString() and other convenience methods that are implemented
 in terms of basic Collection methods.

 The collection is "lame" since it crashes with more than 100 elements!
*/
import java.util.*;

public class LameCollection<E> extends AbstractCollection<E> {
    public final static int SIZE = 100;
    private E[] array;
    private int length;         // current logical length

    // Creates an empty collection.
    // @SuppressWarnings("unchecked")  // how to get rid of warning
    public LameCollection() {
        array = (E[]) new Object[SIZE];
        // The above line is an impossible case for the generics.
        // Cannot say "new E[SIZE];" since, E does not exist at runtime.
        // Best compromise new Object[SIZE] with (E[]) cast. The cast is
        // actually meaningless, since E is erased to just Object at runtime.

        length = 0;
    }

    // Adds an elements to the end.
    public boolean add(E x) {
        array[length] = x;
        length++;
        return(true);
    }

    // Returns the current number of elements.
    public int size() {
        return(length);
    }

    // Returns a new iterator at the beginning
    // of the collection.
    public Iterator<E> iterator() {
        return(new LameIterator());
    }

    /*
     Iterator implemented as a private inner class, so it
     has an implicit pointer to the outer LameCollection
     object. So it can just refer to "length" and "array"
     to get the ivars of the outer LameCollection= -- neato!

     Our internal state is "pre-advanced" -- after
     the call to next(), our state is already
     advanced to point to the next element.
     In this way, hasNext() and next() are simplest.

     Tricky: class here is LameIterator not LameIterator<E>,
     since we are already in a scope where E is bound to something.
    */
    private class LameIterator implements Iterator<E> {
        private int index = 0; // this is the next elem to return
                               // (not the one we just returned)

        public boolean hasNext() {
            return(index < length);   // cute: index is ours, length is the outer class'
```

```
        }

        public E next() {
            E result = array[index];
            index++;
            return result;
        }

        public void remove() {
            // this is an optional operation for iterators

            // This is classic tricky array/boundary/off-by-one code -- make a drawing

            // Move the array elements to the left one slot, starting at index
            // arraycopy args: source, source index, dest, dest index, length.
            System.arraycopy(array, index, array, index-1, length-index);

            // Null out the last pointer to help the GC (optional)
            array[length-1] = null;

            // Move back both the length AND the iteration index
            length--;
            index--;
        }
    }
```

## AbstractCollection -- Classic OOP Pop-down at work

- AbstractCollection implements the convenience methods for you
    - All of the above convenience methods are implemented by AbstractCollection depending
        only the existence of the fundamental size(), add() and iterator().

- Neat example of inheritance
    - Subclass off AbstractCollection
    - You implement the fundamentals
    - Now contains(), toString(), remove(), etc. all work -- they run up in
        AbstractCollection and pop down to your, size(), iterator() code as needed.

- Code re-use: ArrayList, ChunkList, LinkedList, ... they all use the one copy of the code for
    contains() up in AbstractCollection.

## Contains() JDK Source Code Example

```
/**
     * Returns <tt>true</tt> if this collection contains the specified
     * element.  More formally, returns <tt>true</tt> if and only if this
     * collection contains at least one element <tt>e</tt> such that
     * <tt>(o==null ? e==null : o.equals(e))</tt>.<p>
     *
     * This implementation iterates over the elements in the collection,
     * checking each element in turn for equality with the specified element.
     *
     * @param o object to be checked for containment in this collection.
     * @return <tt>true</tt> if this collection contains the specified element.
     */public boolean contains(Object o) {
        Iterator e = iterator();
        if (o==null) {
            while (e.hasNext())
                if (e.next()==null)
                    return true;
        } else {
            while (e.hasNext())
                if (o.equals(e.next()))
                    return true;
        }
        return false;
    }
```