CSPP 511-01:
Introduction to Object-Oriented Programming

Harri Hakula

Ryerson 256, tel. 773-702-8584
hhakula@cs.uchicago.edu

July 24, 2000
Outline

• Reference Types
• Inner Classes
• this
• Polymorphism
• Sorting
• Searching
Reference Types

A variable of class type does not store an object of that class. It stores the memory address instead.

Q: What is the memory storage needed by

```java
public class Link {
    ... methods ...
    int value_;
    private Link next_;
}
```

A: An int plus a single memory address, that is, a reference type.
Inner Classes

Our list-implementation was not an ADT. Here is the outline for one, where the Link-class has been extended with a void connectTo(Link next)-method allowing us to modify the links in the list:

```java
public class IntList {
    private class Link {
        public Link(int value, Link next) ... 
        public Link(int value) ... 
        public int value() { return value_; } 
        public Link next() { return next_; } 
        public void connectTo(Link next) 
            { next_ = next; } 
    }
}
```
private int value_;  
private Link next_;  
}  

IntList() ...  
int size() ...  
void insert(int value) ...  
void remove() ...  
int topValue() ...  
}
Inner Classes

Our implementation of `list` requires us to maintain certain complex relationships for which classes are the natural representation. It is a natural extension of the type system to allow for classes within classes – inner classes.

In practice, one should define complex classes on their own. Inner classes should simple to understand and use. (This is a question of style, of course.)

By declaring the inner class to be private, we can be sure that the state of the objects of that type cannot be modified from outside the scope of the owner class.
There are many ways to count the number of objects in a list. A simple solution is to add a counter to the class and increment and decrement it as needed. The `int size()`-method can be implemented like this also:

```java
int count = 0;
for(Link tmp = root; tmp != null;
    tmp = tmp.next())
    ++count;
```

But when we refer to `int size()` from within the class, we must tie it to an instance. What is that instance? Well, it is `this` instance, of course.
int mySize = this.size();

However, this is handled automatically by the compiler so that this. can usually be omitted.

If the class definition wants to call some other classes, the current instance is stored in this.

ClassInstanceBroker.add(this);
Polymorphism

Polymorphism means that a class can have multiple methods with the same name, but with different signatures. The compiler and the virtual machine can detect which one of the definitions is the correct one.

Why is this useful? In our list-example, we could have two insert-methods: one for adding a single element and one for inserting the contents of another list.

- void insert(int value)
- void insert(List other)
Sorting

Sorting is the prerequisite for efficient searching. It is one of the most studied and best understood parts of classical computer science.

Selection of the right algorithm depends on the choice of underlying data structures. This is to be expected since the accessing of different elements happens at different speeds.

Sorting lists is easy, but relatively slow. Sorting of arrays is much faster, but the even the standard algorithms tend to be complex.
Sorted Lists

Since we can only access elements of the list by traversing from the root link, it is obvious that the best way to sort a list is to ensure that as the elements are inserted, the list remains sorted.

Special cases are:

1. Minimum: \( \text{root} = \text{new Link(value, root)} \)

2. Maximum:
   \( \text{last.next} = \text{new Link(value, null)} \)

3. In the middle:
   \( \text{link.next} = \text{new Link(value, link.next)} \)
Sorted Arrays

Let us consider the selection sort algorithm:

```java
for(int i = 0; i < array.length-1; ++i) {
    target = index of the smallest item of array in the interval [i+1, length-1]
    swap the items at i and target
}
```

This algorithm is correct, because after every iteration of the loop, the items at indices less than the loop variable are sorted.
Searching

For non-ordered lists and arrays, searching can only be done by testing every item.

If the list is sorted, it is possible to stop before every item has been examined. The limitations of lists are valid here also: It is more expensive to access the last item than the first one.

For sorted arrays, we can employ the binary search algorithm. The idea is simple: test the value against the item at the middle of the array. There are three possibilities:

1. Value is found.
2. Value is less than that at the middle, examine the lower half.
3. Value is more than that at the middle, examine the upper half.
Binary Search

```
public class BinarySearch {
    public static int find(int[] a, int value) {
        int lo = 0;
        int hi = a.length - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (a[mid] == value) return mid;
            else if (a[mid] < value) lo = mid + 1;
            else hi = mid - 1;
        }
        return -1; // Not found
    }
}
```