**CS61B, Fall 2002** **Discussion #2** **Amir Kamil**
**UC Berkeley** **9/5/02**

**Topics: Running a Program, Scope, `this`, Pointers, Parameter Passing, Control Structures, Arrays, Lists**

## 1 Runnning a Program

- Java programs are run from the command line. First you must compile your code (e.g. `javac -g Foo.java`) in order to get its .class file. Then you run the program by typing `java <classname>` (e.g. `java Foo`) at the command line. There are a few requirements in order to run a program:

  1. You must have a method defined with the following prototype:
     `public static void main(String[] args)`
     The accessor does not necessarily have to be public, however, and the parameter name can be anything.
  2. The class that contains the above method must be defined in a file with the same name as the class. For example, you cannot run `Foo.main()` if you defined `Foo` in `Bar.java`.

  When you run the program from the command line, the Java machine runs the code in the above method. Command line arguments are placed in the `String[]` parameter.

- Example:

```
public class Computer {
  ... // variables and methods
  public static void main(String[] args) {
    Computer comp = new Computer("Unix");
    comp.turnOn();
    comp.boot();
    System.out.println("state = " + comp.state);
    return;
  }
}
```

## 2 Scope

- The scope of a field is the entire class definition.
- The scope of a method parameter is the entire method body.
- The scope of a variable declared within a method is from where it is declared to the end of the block in which it was declared. A block is delimited by braces, `{` and `}`.
- Variable names refer to the variable of that name in the inner-most scope:

```
public class Computer {
  String os;
  public Computer(String os) {
  /* Here, "os" would refer to the object passed in as the
   * argument
   */
  }
}
```

- Inner variables need not be of the same type as outer variables of the same name:

```
public class Computer {
  int os;
  public Computer(String os) { // this works fine
    ...
  }
}
```

## 3 this

- The keyword `this` refers to the current object. It can be used to refer to instance variables when their names are obscured by scoping rules. In the above example, `this.os` in the body of the constructor would refer to the instance variable `os`.
- You can also call constructors using `this(...)`. However, you can only do so from another constructor, and only as the first statement in the body of the constructor:

```
  /* This constructor is legal */
  public Computer(String os, int clockspeed) {
    this(os); // works
    this.clockspeed = clockspeed;
  }

  /* This constructor is illegal */
  public Computer(String os, int clockspeed) {
    this.clockspeed = clockspeed;
    this(os); // doesn't work
  }
```

## 4 Pointers

- Variables that correspond to object types don't actually hold objects, but they hold `references` or `pointers` to those objects.

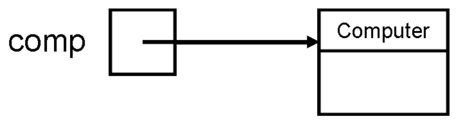

Figure 1: The value of a `Computer` variable.

- A pointer to an object is actually the location in memory at which the object is stored, but Java abstracts away this detail.
- A pointer that doesn't point to anything is a `null pointer` and is symbolized in Java by the keyword `null`. It is illegal to access fields or call methods of `null`.

## 5 Parameter Passing

- Parameter passing is ALWAYS pass-by-value. Remember that containers for primitive types contain values of their respective types, while containers for objects contain values that are references to objects. The value of this container is what is passed when a method is called.

  ```
  static void foo() {
    String s  = "hello world";
    bar(s);
    stdout.format("%s\n").put(s);
  }
  static void bar(String s) {
    s = "goodbye world";
  }
  ```

  What is printed to the screen when `foo()` is called? Figures 2 through 5 show the environment diagrams during its execution. As you can see, `bar()` only modifies its local copy of the variable `s`, which has no effect on `foo()`'s copy. So `hello world` is printed out to the screen.

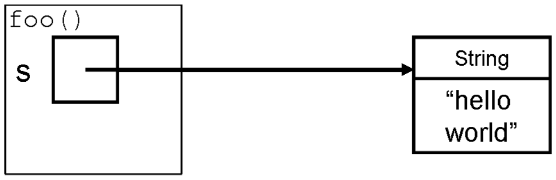

Figure 2: The environment of `foo()` prior to the call to `bar()`.

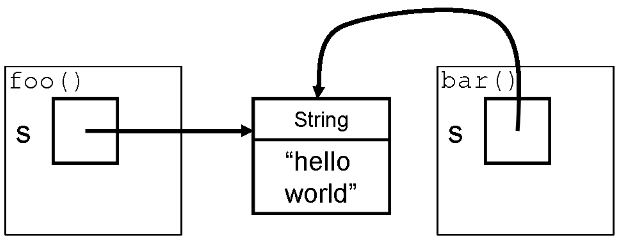

Figure 3: The environment of `foo()` and `bar()` immediately after the call to `bar()`.

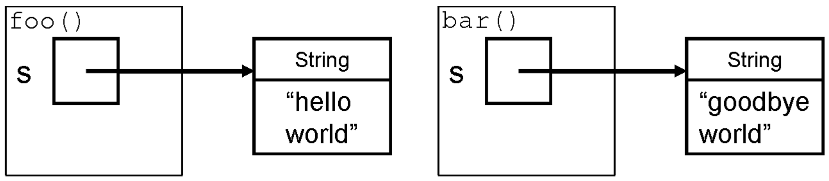

Figure 4: The environment of `foo()` and `bar()` during the call to `bar()`.

## 6 Control Structures

- if statement:

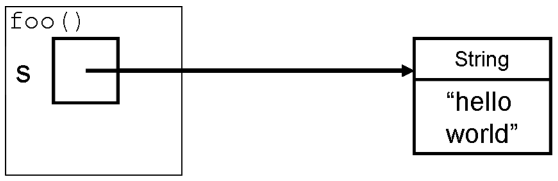

Figure 5: The environment of `foo()` after the call to `bar()`.

```
if (<boolean condition>) {
  <true clause>
}
```

==OR==

```
if (<boolean condition>) {
  <true clause>
} else {
  <false clause>
}
```

If a clause is only one statement, braces surrounding it can be omitted.

Cascaded if:

```
if (...) {
  ...
} else if (...) {
  ...
} else {
  ...
}
```

- while loop:

  ```
  while (<boolean condition>) {
    <body>
  }
  ```

  Execution starts by checking the condition. The body is evaluated if the condition is true, and the process is repeated. This continues until the condition evaluates to false, in which case execution proceeds to the next statement after the loop.

- do while loop:

  ```
  do {
    <body>
  } while (<boolean condition>);
  ```

  This is similar to the while loop, except the condition is checked at the end of each iteration. As a result, the body will execute at least once.

  Equivalent:

```
<body>
while (<boolean condition>) {
  <body>
}
```

- for loop:

  ```
  for (<init>; <condition>; <update>) {
    <body>
  }
  ```

  The init statement is run first, then the condition is checked. If it is true, the body is executed, the update statement is executed, and the condition is rechecked. This continues until the condition is false. Note that the init statement is only run once.

  Equivalent:

  ```
  <init>
  while (<condition>) {
    <body>
    <update>
  }
  ```

## 7 Arrays

- The first data structure you have learned.
- Declaration:

  ```
    <type>[] <name>;
  ```

  Examples:

  ```
    int[] numbers;
    String[] sentences;
  ```

- Creation:

  ```
    new <type>[<size>]
  ```

  Examples:

  ```
    int[] numbers = new int[4];
    String[] sentences = new String[18];
  ```

  Another way:

  ```
    int[] numbers = {1, 2, 3};
  ```

  This creates an int array of size three with the above elements. Note that this form can only be used when declaring the array.

- Access:
  To access the nth element in an array:

  ```
  <name>[n]
  ```

  Example:

  ```
  numbers[2] = 4;
  ```

- As shown above, array elements can be assigned to.
- Array indexing starts at 0, i.e. the first element of `numbers` is `numbers[0]`.
- Legal array access is limited to n between 0 and one less than the size of the array, inclusive. To get the size of `numbers`, use `numbers.length`.
- Arrays are objects and therefore have methods and fields. They inherit all methods and fields from the `Object` class.
- Primitive arrays (e.g. `int[]`) store values of their respective types, while object arrays (e.g. `Computer[]`) store references to objects of their respective types. Positions in object arrays that don't point to an object hold `null` pointers.

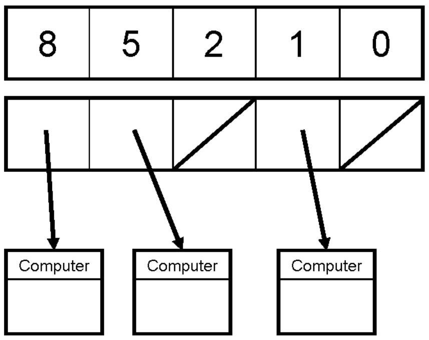

Figure 6: An `int` and a `Computer` array of size 5.

## 8 Lists

- A list consists of multiple `nodes` that each contain a single object and references to other nodes in the list.
- Lists can be singly or doubly-linked. In singly-linked lists, each node only points to the next node in the list. In doubly-linked lists, each node points to both the previous and next nodes in the list.
- A node in a singly-linked list:

```
public class ListNode {
  public ListNode next; // the next node in the list
  public int data;
  public ListNode(int data, List next) {
    this.data = data;
    this.next = next;
```

```
  }
}
```

- Some lists abstract away their implementation details and only provide a container class that has access to individual nodes, but doesn't provide the user access to those nodes. Example:

```
public class SList {

  private ListNode head;

  public SList() {
    head = null;
  }

  // insert a value at the front of the list
  public void insertFront(int value) {
    head = new ListNode(value, head);
  }

  // remove the front of the list, return its value
  public int removeFront() {
    int value = head.value;
    head = head.next;
    return value;
  }

  // other methods

}
```

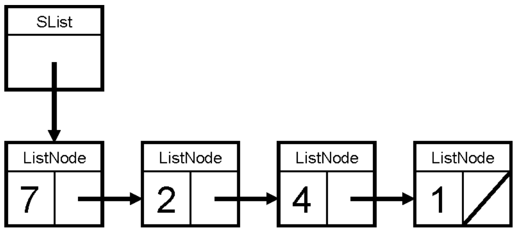

Figure 7: An `SList` of size 4.