



# Full Java, Arrays, Mutation

---

Corky Cartwright

Vivek Sarkar

Department of Computer Science

Rice University



# What are Language Levels Hiding?

---

- In principle, nothing...  
Java could have supported a notion of *immutable* (single assignment) classes, and DrJava Intermediate Level could just have been a subset of Java.
- But Java is what it is ...
- Transforming DrJava IL code to full Java code:
  - Explicit constructors
  - Explicit accessors
  - Explicit overriding of toString()
  - Explicit overriding of equals
  - Explicit overriding of hashCode()

} Discussed in today's class



# Explicit Constructors

---

- A constructor definition has the form:

```
<ClassName>(arg1, ..., argn) {  
    <optional super call on superclass constructor>  
    <code body that initializes instance fields of class>  
}
```

- All fields not initialized in explicit constructors are set to the default value for their respective type: **0** for all primitive number/char types, **false** for **boolean** and **null** for all object (reference) types.
- Multiple constructors are permissible (static overloading with different signatures and in different subtypes).
- If no explicit constructors are provided, Java automatically generates a default 0-ary constructor with an empty body.



# Explicit Accessors

---

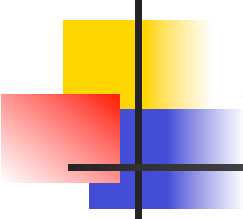
- An accessor definition is an ordinary instance method definition of the form:  
`<accessorName>() { return <fieldName>; }`
- The choice of `<accessorName>` is arbitrary. We recommend using the corresponding `<fieldName>`. Another common convention is `get<fieldName>`.
- Instance fields should never be `public`.



# Explicit Overriding of `toString`

---

- The default definition of `toString`, which has signature `public String toString();` is awful: `<className>@<hashCode>`.
- Why is `toString` important? This representation is used anytime that an object is printed, e.g. in many testing contexts.



# Recap of Scheme Vectors (Lecture 16)

---

- `(vector V-0 ... V-n)` creates a vector with  $n+1$  elements,  $V-0$  through  $V-n$
- `(vector-length V)` returns the number of items in vector  $V$ 
  - Results in an error if  $V$  is not a vector
- `(vector-ref V i)` returns the  $i^{\text{th}}$  item in vector  $V$ 
  - Results in an error if  $V$  is not a vector or  $i$  is not a number or  $i < 0$  or  $i \geq (\text{vector-length } V)$



# Java Arrays

---

- An *array* of  $T$  is a contiguous sequence of data values of type  $T$ . The length of the array is fixed when the array is created.
- The Java type for an array of  $T$  is written as  $T[ ]$ . Arrays are a special form of object. The array class cannot be extended, but arrays can be used anywhere arbitrary objects can be used. When an array is allocated, the size of the array is inserted between the square brackets: `new int[17]`. By default, the elements of the array are initialized to the default value for the element type (typically zero or null).
- An alternate syntax for the `new` operation on arrays `new int[] {0,1,2}` explicitly lists the initial contents of the array and leaves the length implicit.
- NOTE: Arrays are not supported at the Intermediate Level of DrJava



# Java Arrays (contd)

---

- To extract an element from an array *a*, we simply use the subscript operation `[index]` where *index* is an integer expression with a value in the range 0...n-1, where n is the length of the array. For example,

```
new int[] {0,1,2} [0] => 0
```

```
new int[] {0,1,2} [1] => 1
```

```
new int[] {0,1,2} [3] => ArrayIndexOutOfBoundsException
```

- To update an array, you can use an array subscript in the left-hand side of a conventional assignment statement:

```
int[] a = new int[2];
```

```
a[0] = 0;
```

```
a[1] = 1;
```

- The meanings of `equals` and `toString` in array classes are the standard `Object` defaults.
  - Beware. `equals` means object identity and `toString` prints `L<eltType>@address`, where `eltType` is a Java abbreviation for the element type and `address` is the hashCode of the object expressed in hexadecimal.





# Java Arrays (contd)

---

- The only interesting member of array classes is the field **length**
  - NOTE: collection classes typically use a **size()** method instead.
- Since the array class cannot be extended and the default members of the class provide little functionality beyond array in C, many Java programmers use **ArrayList** (preferred) or **Vector** (archaic) instead.
- To explain how to write clean code for processing arrays, we identify them with lists and use some cleverness in representing the tails of arrays.
- It is straightforward to write tail-recursive code to perform array computations from the perspective that arrays are restricted lists. But to produce good array code in Java we must go one step farther and convert that tail-recursive code to loop code.



# Recipes for Processing Arrays

---

- Assume that we want to sum the elements of an `int[]` array. We can express the naive solution using structural recursion as follows:

```
class ArrayUtil {
    public static int sum(int[] a) { return sumHelp(a, 0); }
    /** sumHelp(a, i) sums a[i],...,a[a.length - 1] */
    public static int sumHelp(int[] a, int i) {
        if (i >= a.length) then return 0;
        else return a[i] + sumHelp(a, i+1);
    }
}
```

We introduced a help function because auxiliary arguments are required to describe list tails.

- We can improve this naïve program by converting it to tail-recursive form:

```
class ArrayUtil {
    public static int sum(int[] a) { return sumHelp(a, 0, 0); }
    /** Returns the sum of A[0],...,A[a.length-1] given 0 <= i < a.length
        and accum = A[0],...,A[i-1] */
    public static int sumHelp(int[] a, int i, int accum) {
        if (i >= a.length) return accum;
        else return sumHelp(a, i+1, a[i] + accum);
    }
}
```



# Tail Recursion Is Not Enough!

---

- Java does not generally optimize tail calls (it is implementation dependent and unsupported in Sun JVMs which means you cannot rely on tail recursion.) In my opinion, this was a grievous error in the definition of Java (an opinion shared by Guy Steele who wrote the original edition of the JLS).
- Implication: must use Java loops instead of tail recursive helper functions to get good performance and memory utilization.



# Connection between Loops and Tail Recursion

---

- What is a while loop? The code  
`while (test) update`  
evaluates the boolean expression `test` and falls through to the next statement if `test` is false. Otherwise, it evaluates `update` and executes the loop again.
- If we model mutations in the code containing the while loop as changes to fields of the enclosing object, then the while loop is equivalent to calling a method  

```
void whileFun() {  
  if (! test) return;  
  else {  
    update;  
    whileFun();  
  }  
}
```
- Note that this function template is simply a restricted form of tail recursion. Let's convert our `sum` function to while loop form. The state will be the pair of variables `(i, accum)` formulated as parameters in the tail recursive code.



# Loops and Tail Recursion

---

- Given

```
class ArrayUtil {
    public static int sum(int[] a) { return sumHelp(a, 0, 0); }
    /** Returns the sum of A[0],...,A[a.length-1] given 0 <= i < a.length
        and accum = A[0],...,A[i-1] */
    public static int sumHelp(int[] a, int i, int accum) {
        if (i >= a.length) return accum;
        else return sumHelp(a, i+1, a[i] + accum); /* update i, accum */
    }
}
```

- The same code in loop form (no recursion):

```
class ArrayUtil {
    public static int sum(int[] a) {
        /** Returns the sum of A[0],...,A[a.length-1] */
        accum = 0; i = 0; /* bind parms i, accum */
        /* Invariant: accum = a[0] + ... + a[i-1] */
        while (i < a.length) {
            accum = accum + a[i]; i++; /* update i, accum */
        }
        return accum;
    }
}
```



# Using a **for** loop instead of **while**

---

- In Java/C

```
for ( init; test; incr ) body
```

- abbreviates

```
init;  
while (test) {  
    body;  
    incr;  
}
```

- Hence, we can rewrite our loop:

```
class ArrayUtil {  
    public static int sum(int[] a) {  
        /** Returns the sum of A[0],...,A[a.length-1] */  
        int accum = 0;  
        /* Invariant: accum = a[0] + ... + a[I-1] */  
        for (int i = 0; i < a.length; i++) {  
            accum = accum + a[i];  
        }  
        return accum;  
    }  
}
```



# Mutation: Succumbing to the Dark Side?

---

- Four common problems:
  1. Assume that we are repeatedly evaluating a method/function  $m$  often evaluating  $m$  on the same list of arguments. How can we avoid performing the same computation more than once?
  2. Assume we want to compute the number of a nodes in a tree data structure where nodes can be shared (the standard situation in functional programming or OO programming with immutable data). How can we efficiently perform this computation.
  3. Perhaps simplest data structure from the perspective of machine implementation is the array: a fixed-size list of elements  $T$  that is allocated in contiguous machine memory where each element  $T$  is represented by a fixed size chunk of memory. The array was the *only* data structure in the original Fortran language. How can we create such structures using simple machine operations? How can we efficiently compute new ones?
  4. How can I represent cyclic linked structures (general graphs rather trees)?
- The best solutions to these four problems all rely on *mutation*