CS1110 17 Mar 2009 Arrays. Reading: Secs 8.1, 8.2, 8.3

Listen to the following lectures on loops on your Plive CD. They are only 2-3 minutes long, and each has an insightful message.

1. The 3 lectures on Lesson page 7-6 —read the whole page.
2. The 4 lectures in Lesson page 7-5.

Computational simplicity

Most of us don't write perfect essays in one pass, and coding is the same: writing requires revising; programming requires revising.

If you are writing too much code —it gets longer and longer, with no end in sight: stop and look for a better way.

If your code is getting convoluted and you have trouble understanding it: stop and look for a better way.

Learn to keep things simple, to solve problems in simple ways. This sometimes requires a different way of thinking.

We are trying to teach not just Java but how to think about problem solving.

A key point is to break a problem up into several pieces and do each piece in isolation, without thinking about the rest of them. Our methodology for developing a loop does just that.


```
/* day contains the number of days
   since ORIGINYEAR (1 Jan 1980)
*/
/* Set year to current year and day to
   current day of current year */
year = ORIGINYEAR; /* = 1980 */

while (day > 365) {
  if (IsLeapYear(year)) {
     if (day > 366) {
        day= day – 366;
        year= year + 1;
     }
  } else {
     day= day – 365;
     year= year + 1;
  }
}
```

Does each iteration make progress toward termination? Not if day == 366!!

Apple had problems with daylight savings time

Zune error
http://tinyurl.com/9b4hmy

On 31 Dec 2008, the Zune stopped working. Anger! On 1 Jan 2009 it worked.

Zune clock code keeps time in seconds since beginning of 1980. It calculates current day and year from it.

Example

year	day
1980	738
1981	372
1982	7

Understanding the pieces of a loop

Init

// invariant

while (Cond) {

Repetend

}
// R: (result assertion)

When developing the loop, how do we write the three pieces? When understanding a loop that someone gives us, how do we know the pieces are right?

1. How does the initialization make inv true?
2. Does inv together with !Cond tell us that R is true?
3. Does Repetend make progress?
4. Does the repetend keep inv true

invariant: a definition of the relationship between the variables. Holds before/after each iteration.

Array: object. Can hold a fixed number of values of the same type. Array contains 4 **int** values.

The **type** of the array:

int[]

Variable contains name of the array.

x a0 int[]

Basic form of a declaration:

<type> <variable-name> ;

A declaration of x.

int[] x ;

Does not create array, it only declares x.
x's initial value is **null**.

Elements of array are numbered: 0, 1, 2, …, x.length–1;

	a0
0	5
1	7
2	4
3	-2

Make everything as simple as possible, but no simpler. Einstein

Notes on array length

Array length: an instance field of the array.

This is why we write x.length, not x.length( )

Length field is **final:** cannot be changed.

Length remains the same once the array has been created.

We omit it in the rest of the pictures.

	a0
length	4
0	5
1	7
2	4
3	-2

x a0 int[]

The length is not part of the array type.

The type is **int**[]

An array variable can be assigned arrays of different lengths.

Arrays

int[] x ;

x null int[]

x= **new int**[4];

Create array object of length 4, store its name in x

x a0 int[]

	a0
0	0
1	0
2	0
3	0

x[2]= 5;
x[0]= -4;

Assign 5 to array element 2 and -4 to array element 0

x[2] is a reference to element number 2 of array x

	a0
0	-4
1	0
2	5
3	0

int k= 3;
x[k]= 2* x[0];
x[k-1]= 6;

Assign 2*x[0], i.e. -8, to x[3]
Assign 6 to x[2]

	a0
0	-4
1	0
2	6
3	-8

Difference between Vector and array

	int[] a;	Vector v;
	Elements of a: **int** values	Elements of v: any Objects
Creation:	a= **new int**[n];	v= **new** Vector();
	Array always has n elements	Number of elements can change
Reference:	a[e]	v.get(e)
Change element:	a[e]= e1;	v.set(e, e1);

Array locations a[0], a[1], … in successive locations in memory. Access takes same time no matter which one you reference.

Elements all the same type (a primitive type or class type)

Can't tell how Vectors are stored in memory. Referencing and changing elements done through method calls

Elements of any Object type (but not a primitive type). Casting may be necessary when an element is retrieved.

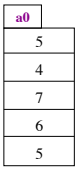
Array initializers

Instead of

```
int[] c= new int[5];
c[0]= 5; c[1]= 4; c[2]= 7; c[3]= 6; c[4]= 5;
```

Use an array initializer:

```
int[] c= new int[ ] {5, 4, 7, 6, 5};
```



No expression between brackets [ ].

(can omit this)

array initializer: gives values to be in the array initially. Values must have the same type, in this case, **int**. Length of the array is the number of values in the list

Computer science has its field called computational complexity; mine is called computational simplicity. Gries

8

Use of an array initializer

```
public class D {
   public static final String[] months= new String[]{"January", "February",
                        "March", "April", "May", "June", "July", "August",
                        "September", "October", "November", "December"};

   /** = the month, given its number m
       Precondition: 1 <= m <= 12 */
   public static String theMonth(int m) {
      return months[m–1];
   }
}
```

Months[m–1] is returned, since

months[0] = "January",
months[1] = "February",
…

Variable months is:

static; object assigned to it will be created only once.

public: can be seen outside class D.

final: it cannot be changed.

9

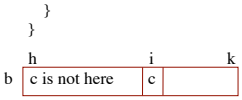
Linear search

```
public class D {
   /** = index of first occurrence of c in b[h..]
       Precondition: c is guaranteed to be in b[h..] */
   public static int  findFirst (int c, int[] b, int h) {
      // Store in i the index of first c in b[h..]
      int i=  h;
      // invariant: c is not in b[h..i-1]
      while ( b[i] != c)  {
         i= i + 1;
      }
      // b[i] = c  and c is not in b[h..i-1]
      return i;
   }
}
```

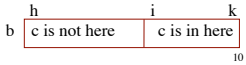
Remember

h..h-1 is the empty range

Loopy questions:
1. initialization?
2. loop condition?
3. Progress?
4. Keep invariant true?



invariant

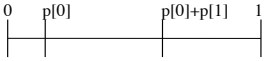


10

```
/** = a random int in 0..p.length-1, assuming p.length > 0.
    The (non-zero) prob of int i is given by p[i].*/
public static int roll(double[] p) {
   double r= Math.random(); // r in [0,1)
   /** Store in i the segment number in which r falls. */
   int i  = 0 ;   double iEnd= p[0];
   // inv: r is not in segments looked at (segments 0..i-1)
   //      and iEnd is the end of (just after) segment i
   while (   r not in segment i  ) {
                 r >= iEnd
      iEnd= iEnd + p[i+1];
      i= i + 1;
   }
   // r is in segment i
   return  i;
}
```

It's a linear search!

1. init
2. condition
3. progress
4. invariant true

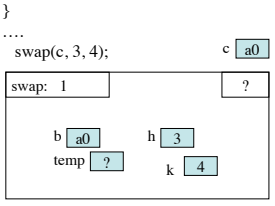


11

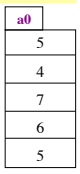
Procedure swap

```
public class D {
   /** = Swap b[h] and b[k] */
   public static void swap (int[] b, int h, int k) {
      int temp= b[h];
      b[h]= b[k];
      b[k]= temp;
   }
}
….
  swap(c, 3, 4);
```

Does swap b[h] and b[k], because parameter b contains name of the array.



frame for call just after frame is created.



12