Data Structures and Information Systems
Part 1: Data Structures

Ullrich Hustadt

Lecture 6: Arrays (2)

Last time...

- ADT Array
- Java arrays
  - creation
  - access
- Primitive types and reference types
- Internal representation of Java arrays
- Operations on arrays

Topics

- Java arrays: Access implementation
- Arrays: Advantages and disadvantages
- Enlarging arrays
- Arrays of reference types
  - Arrays of string objects
  - Arrays of objects

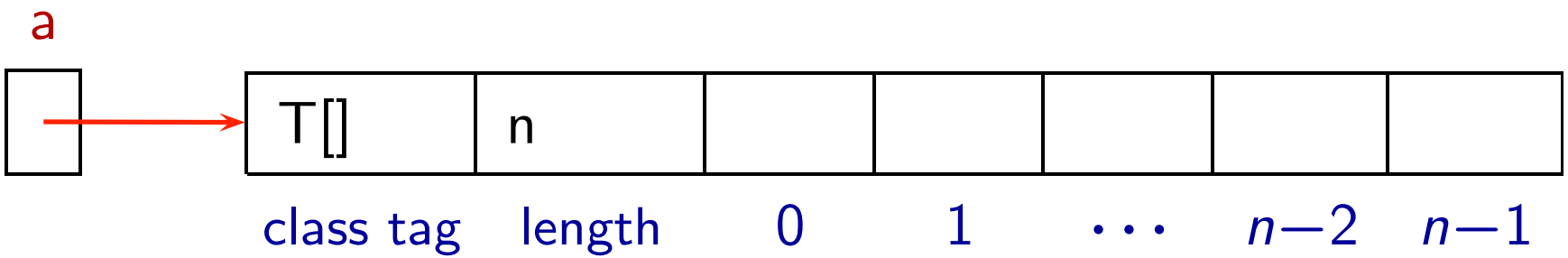
Internal representation of Java arrays

- Java arrays are reference types

  Example:

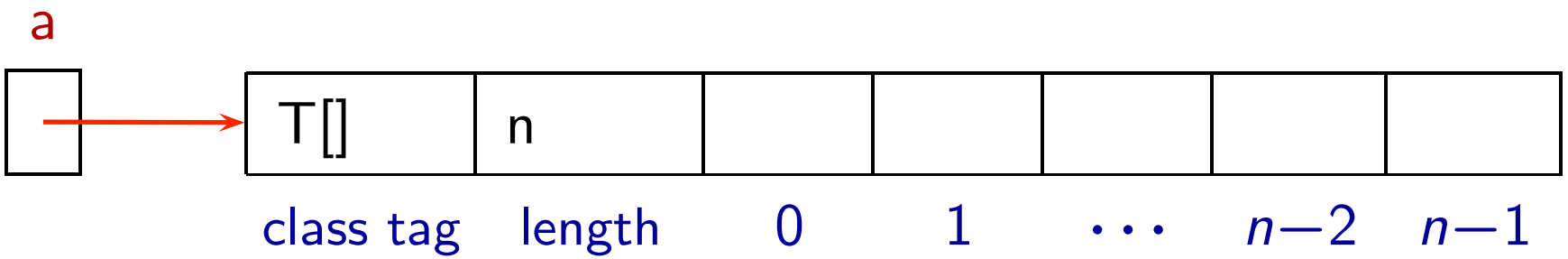
  T[] a = new T[n]

  Defines an array variable with name a whose components are of type T and allocates memory space for n such components



- Given an index i how do we work out where to look for the component a[i]?

Java arrays: Access implementation



- Given an index i how do we work out where to look for the component a[i]?

$$\begin{aligned}\text{memory location of a[i]} = {} & \text{value of variable a} \\ & + \text{size of class tag T[]} \\ & + \text{size of length attribute} \\ & + (\text{i} \times \text{size of component})\end{aligned}$$

- This computation can be done in constant time

Arrays: Advantages and disadvantages

- Arrays, including Java arrays, are static data structures since the number of elements they can store is fixed at the time of creation
- As such they inherit all the advantages and disadvantages of static data structures

Arrays: Advantages

- Easy to specify
  (declaration, allocation of memory space, initialisation can all be done in one line of code)
- There is no run-time overhead to allocate/free memory
  (apart from once at the start and end)
- Random (direct) access to any element via the index
  - e.g. direct access to a student record via student number
    (index can be derived from student number)
- It is usually faster to sequentially access elements due to
  - contiguous storage and
  - constant time computation of the address of a component

Arrays: Disadvantages

- A potential disadvantage of arrays (depending on the application) is the need to know the size at the time of allocation.
- Careful design is required to make sure that the array will be large enough to hold the largest possible group of data but no larger

Can we find a workaround for this problem?

Yes, if more elements are required than an array is able to store, then a new larger array needs to be created and the data transferred across.

Enlarging arrays

Assume that oldArray is an integer array which is too small for the number of elements we want to store in it

The following code replaces oldArray by an array of twice the size of the original array

```
int   oldSize  = oldArray.length;
int   newSize  = 2 * oldArray.length;
int[] newArray = new int[newSize];
for (int i=0; i<oldArray.length; i++)
  newArray[i]=oldArray[i];
oldArray = newArray;

oldArray[oldSize] = 5;
```

The last line works now (instead of throwing an exception)

Arrays as method arguments (1)

Here is an example of how to write a method that takes an array as an argument

It simply encapsulates the algorithm for enlarging an array we have just seen in a method

```
public static int[] doubleSize( int[] oldArray )
{
  int[] newArray = new int[2 * oldArray.length];
  for (int i=0; i<oldArray.length; i++)
    newArray[i] = oldArray[i];
  return(newArray);
}
```

Arrays as method arguments (2)

Now we use this method in a program:

```
import java.io.*;
class TestArray5 {
  public static int[] doubleSize( int[] oldArray ) {
    ...
  }

  public static void main( String args[] ) {
    int[] primes   = new int[]{2,3,5};
    int   oldSize  = primes.length;
    primes = doubleSize(primes);
    primes[oldSize] = 7;
    for (int i=0; i<primes.length; i++)
      System.out.println(primes[i]);
} }
```

Arrays as method arguments (3)

The output of program TestArray5 is:

```
2
3
5
7
0
0
```

Thus, it seems that instead of three integers, our array is now able to store six integers

Doesn't that mean that what we said about the disadvantages of static data structures and Java arrays in particular is wrong?

Static data structures again

- Static data structures, including Java arrays, are fixed in size at the time of creation
- The method doubleSize seems to allow us to enlarge a Java array
- However, looking closer, doubleSize does not enlarge the array it is given as an argument, but it creates a completely new array
- Also, using doubleSize we now incur the same disadvantage that dynamic data structures have: a memory allocation overhead
  As a consequence, this approach only makes sense, if the number of times we have to use doubleSize is significantly smaller than the number of times we have to access an array components (so, that we are still faster than with a dynamic data structure)

Arrays of reference types

- Apart from arrays of primitive data types we can have arrays of reference types (objects)
- These arrays of reference types are created and used in much the same way as the arrays we have already seen
- Note that in an array of reference types not the objects are stored in the array, but references (memory locations) to the objects

  The objects themselves have to be created and memory space has to be allocated for them separately

Arrays of string objects: Creation

- A particualar reference type we already know is String
- So,

```
String[] words = new String[4];
```

  creates an array of strings with four components
- Remember, this declaration does not allocate any memory space for the strings
- Also, all the components are initialised with the null reference, not with an empty string

Arrays of string objects: Storing strings

- Next, let us store strings in our array of strings

```
words[1] = "tim";
words[2] = "tom";
words[3] = "tim";
```

- Remember, `"tim"` and `"tom"` are so-called string literals
- Are words[1] and words[3] equal?

  Both `words[1] == words[3]`

  as well as `words[1].equals(words[3])` return `true`!

  This is due to the compiler which only creates one internal string object for both occurrences of `"tim"` in our code

Arrays of objects: Creation

- The most general kind of array would have components of type Object
- The Object class is located at the top of the Java class hierarchy Every Java class is a descendent, direct or indirect, of the Object class, but not the primitive data types
- We can declare an array with components of type Object as follows:

```
Object[] objArray = new Object[3];
```

Arrays of objects: Storing objects

- In the components of objArray we can now store objects of any subclass of Object

```
objArray[0] = "tim";
objArray[1] = new BasicBox(1,2,3);
```

- Remember that BasicBox is a class that we have defined earlier
  We assume that the definition of the BasicBox class is extended by

```
public String toString() {
      return( "box("+depth+","+height+","+width+")" );
}
```

  which overrides the default method for providing a String representation of an object

Arrays of objects: Storing primitive data

- But, unfortunately, not we cannot directly store an element of a primitive data type in it

  Including the following of code

```
objArray[2] = 5;
```

  results in the following error message by the compiler:

```
TestObjectArray.java:12: incompatible types
found   : int
required: java.lang.Object
    objArray[2] = 5;
                  ^
1 error
```

Arrays of objects: Storing primitive data

- To store an integer value like 5 in objArray we have to make use of a wrapper class
- Each of the primitive data types like char or int has a corresponding wrapper class associated with it, named Character, Integer, etc.

  These classes wrap the primitive data types into a class which in turn is a subclass of Object

```
objArray[2] = new Integer(5);
```

Arrays of objects: Accessing (1)

- To access all the components of an array of reference types we can use the same code that we have used for integer arrays

```
for (int i=0; i<objArray.length; i++)
    System.out.println("objArray["+i+"] = "+objArray[i]);
```

will produce the following output

```
objArray[0] = tim
objArray[1] = box(1,2,3)
objArray[2] = 5
objArray[3] = null
objArray[4] = null
```

Arrays of objects: Accessing (2)

- However, since we have now turned an integer value into an Integer object, we can no longer use it in the same way as before

  Adding the line

```
int n = 3 + objArray[2];
```

  to our program results in the following error message by the compiler:

```
TestObjectArray.java:16: operator + cannot be applied to
                         int,java.lang.Object
    int n = 3 + objArray[2];
                ^
1 error
```

Arrays of objects: Accessing (3)

- Instead, we have to unwrap the Integer object first, to get the integer value back
- Unfortunately, the straightforward approach

```
int n = 3+objArray[2].intValue();
```

produces again an error message by the compiler:

```
TestObjectArray.java:17: cannot resolve symbol
symbol  : method intValue ()
location: class java.lang.Object
    int n = 3 + objArray[2].intValue();
```

The compiler only knows that the components of objArray belong to class Object

It does not know that objArray[2] belongs to Integer

Arrays of objects: Accessing (4)

- One way of solving this problem is to use typecasting: We typecast objArray[2] from the Object class to the subclass Integer

  Once this is done, we can use the intValue method of the Integer class to retrieve the actual integer value

```
int n = 3 + ((Integer)objArray[2]).intValue();
System.out.println("n = " + n);
```

  produces the output

```
n = 8
```