Lecture 2

Designing new Data Types: Composition

Creating New Types (Classes)

- Java already has many types (classes) – each designed to perform a specific task
- See API

https://docs.oracle.com/en/java/javase/12/docs/api/java.base/module-summary.html

- However for each new task we might need a new class
- Rarely we build the entire new class from scratch: we reuse other classes
- There are two primary techniques for doing this
  - Composition (Aggregation)
  - Inheritance

Composition

- Instance variables can be of any type: they can also be reference variables which refer to an object of a custom type (class)
- This way we can construct complex objects which contain simpler objects inside them
- The method of constructing a program by incorporating smaller objects inside a larger one is called **composition**
- This is the most useful and widely used approach in Object-Oriented Programming

- New class has no special access to the instance variables/methods of included objects
- Methods in new class are often implemented by utilizing methods from the instance variable objects

Example: *Hospital* class

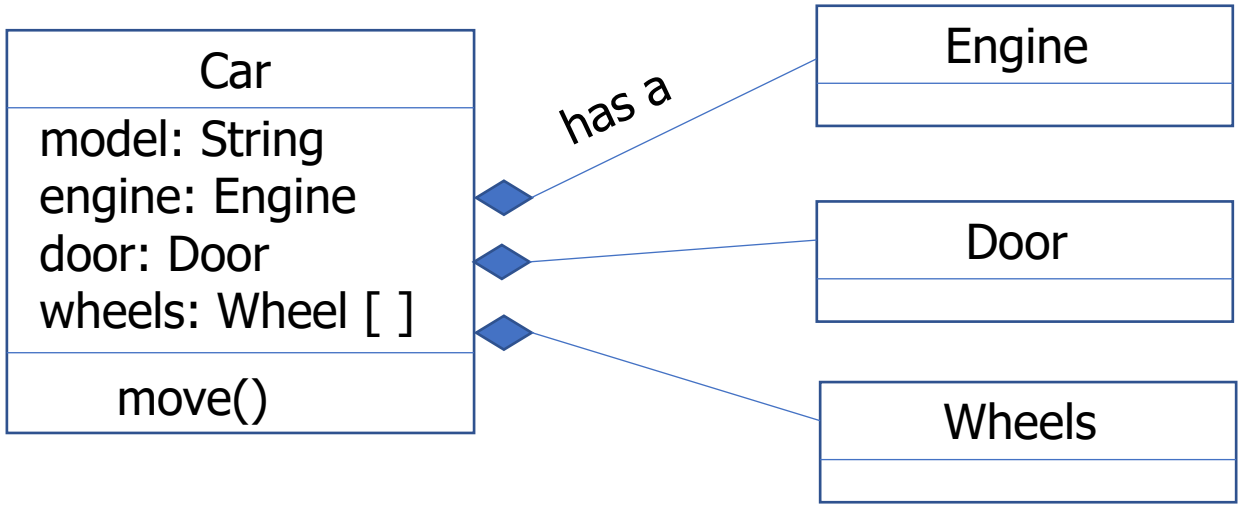
```
public class Hospital {
    private String name;
    private Patient[] patients;
    int numPatients;
    int capacity;

    public Hospital(String name, int capacity) {
        this.name = name;
        patients = new Patient[capacity];
        this.capacity = capacity;
    }

    public void addPatient(Patient p) {
        if (this.numPatients < this.capacity)
            this.patients[numPatients++] = p;
        else
            System.out.println("…");
    }

    public void cureAll() {
        for(int i=0; i<numPatients; i++)
            patients[i].cure();
    }
```

Patient class is defined in a separate file, that can be written by another programmer

Patients know how to cure themselves

Advantages of Composition: abstraction

"Has a" relationship between new class and included classes



People who build engines do not have to know how to make wheels

- While combining elementary objects we ensure that we expose only important properties and capabilities of these objects (contract, public interface, abstract away details)
- We can divide work among many programmers: each programmer can concentrate on correct implementation of their own small piece

Mutable and Immutable Objects

- Many Java classes contain *mutator* methods: these are methods that allow us to change the property of an object
- Objects that can be changed via mutators are said to be **mutable**
  - Ex: **`StringBuilder`**

    `append()` method adds characters to the current **`StringBuilder`**
  - Ex: **`ArrayList`**

    `add(), remove(), set()`

- Some classes do not contain mutator methods: objects from these classes are said to be **immutable**
  - Ex: String
    - Cannot alter the string once the object is created
  - Ex: wrapper objects (Integer, Float, etc)
    - Allow accessors but no mutators

Complications with Immutable Objects

- If we cannot modify the properties of an object we are forced to create new objects to solve a particular problem
  - Ex: Concatenating Strings

```
String S1 = "Hello ";
S1 = S1 + "there";
```

  - We must create and assign a new object rather than just append the string to the existing object
  - If done repeatedly this can cause **a lot** of overhead

Example: Mutable string?

```
public class MuString {
 private String name;

 public MuString(String s, int i) {
     name = new String(s);
 }

 public void setCharAt(int i, char c) {
     StringBuilder b = new StringBuilder(name);
     b.setCharAt(i, c);
     name = b.toString();
 }
}
```

We cannot mutate the inner representation of the String, so we must change it in a rather convoluted way

Complications with Mutable Objects

- Consider a new class that is a subclass of an *ArrayList*
- Let's say we store **mutable** objects inside it
- When we add an object to a collection, it doesn't mean we give up outside access to the object
  - Ex. As in *ArrayList* we can call *get*(i) and get the element at position *i* into a reference variable
  - If this object happened to be mutable, we can alter the object outside of our class, and we could destroy a desired property of the collection

Example:
Sorted collection of mutable objects

- Assume our new subclass of ArrayList is a ***SortedArrayList*** intended to keep the data sorted
- The elements of an array are mutable: ex. *StringBuilder*
- We can get access to a *StringBuilder* at position *i* and modify it outside the *SortedArrayList* class
- This could make the ordering invalid, but our class would not be aware of the change

Solution: use copies

- What to do? How to prevent the change outside of our class? How to maintain order?
  - We could use only immutable objects for the *SortedArryList*
  - Or we can put copies of our original objects into the collection
  - And to be very safe – our accessors should themselves return copies of the objects rather than references to the originals

And here we face another challenge:

- How to make a copy of an object?

Copying Objects in Java

- Syntactically, Java objects can be copied using:
  1. clone() method
  2. copy constructor

1.Clone()

```
A a = new A();
A a1 = a.clone();
```

- *clone*() is defined in class *Object*, so it will work for all Java classes
- By default it will only copy the values of instance variables
- This works well for primitives, but as you are aware – when we copy reference variables they will contain the same address value in both copies and thus they will refer to the same object!

- So if you want to also create copies of nested objects, you must **reimplement *clone* for new classes to work properly**

- *clone*() is often already defined for Java arrays, StringBuilders (and some other classes), so we can use it for them without overriding

2. Copy constructor

```
A a = new A();
A a1 = new A(a);
```

- We can implement a copy constructor for any new Java class that we write

- Some predefined classes already have them

```
String s = new String("hello");
String t = new String(s);
```

Shallow vs. Deep Copy

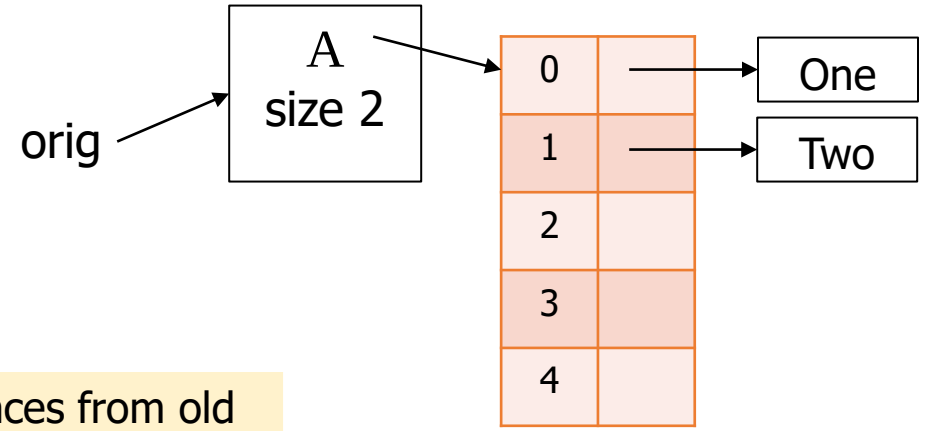
- Recall that many classes are built using composition
  - Thus an object may contain references to other objects
    - These can in turn contain references to other objects

- If we copy an object, what do we do about any references within that object?

- In a shallow copy, we just copy those references to the new object: as a result, both copies of the object refer to the same "nested" objects
- In a deep copy, all nested objects must also be copied

Example 1: Shallow Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // shallow copy
   public SBArray(SBArray old)
   {
      A = old.A;
      size = old.size;
   }
}
```

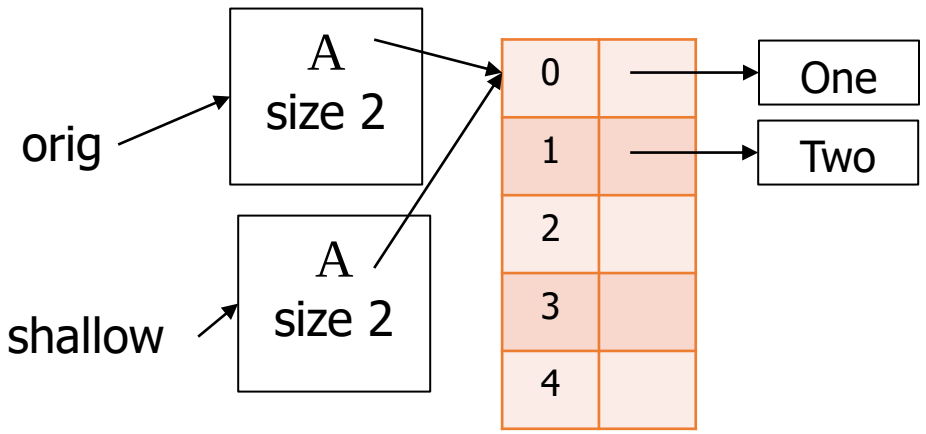


- Copy references from old object to new object

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));
```

Example 1: Shallow Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // shallow copy
   public SBArray(SBArray old)
   {
      A = old.A;
      size = old.size;
   }
}
```

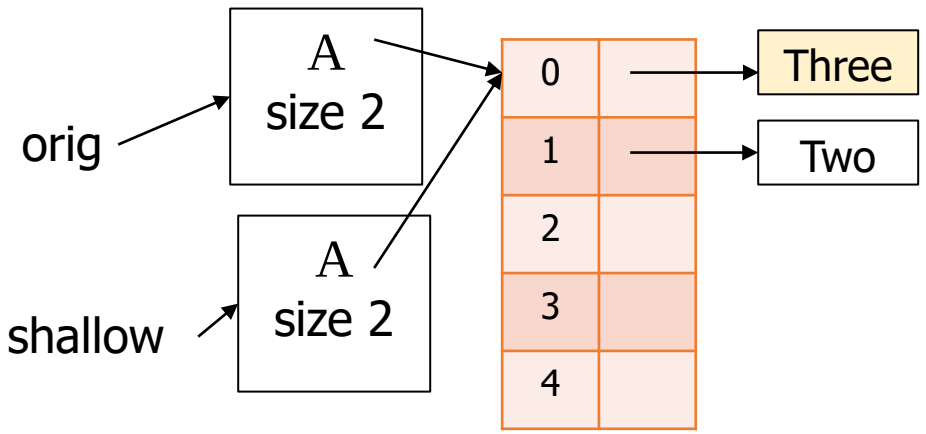


- The entire array is shared by original and copy

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray shallow = new SBArray(orig);
```

Example 1: Shallow Copy

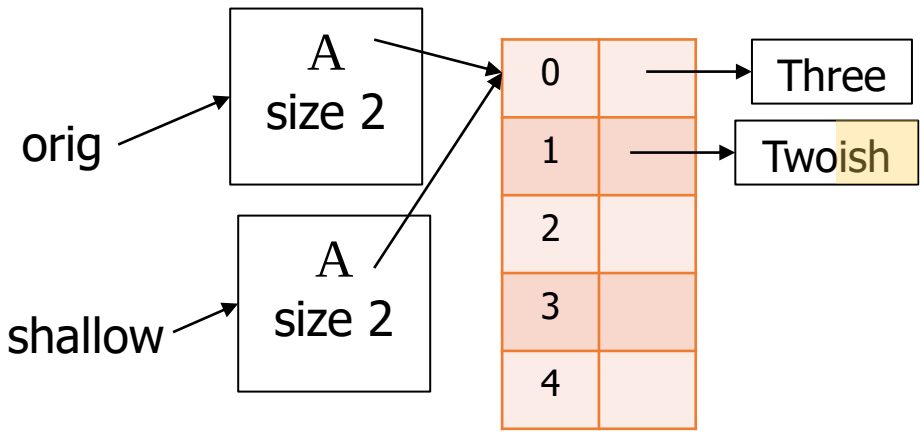
```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // shallow copy
   public SBArray(SBArray old)
   {
      A = old.A;
      size = old.size;
   }
}
```




```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray shallow = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
```

Example 1: Shallow Copy

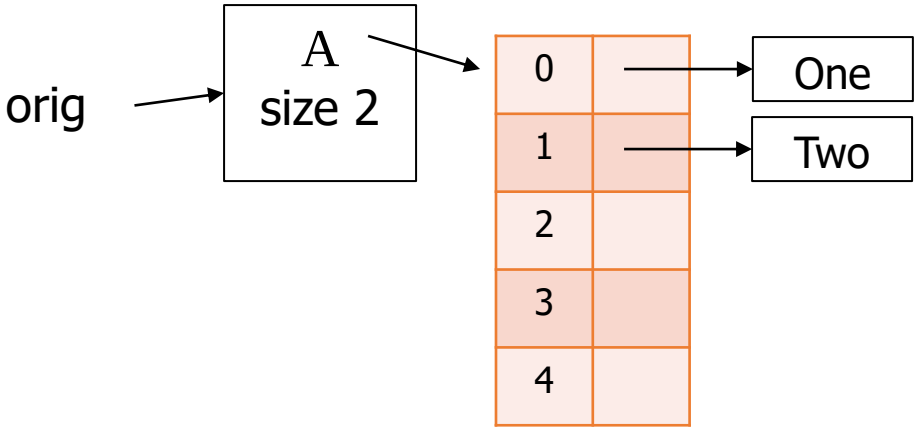
```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // shallow copy
   public SBArray(SBArray old)
   {
      A = old.A;
      size = old.size;
   }
}
```




```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray shallow = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
shallow.get(1).append("ish");
```

Example 2: Deeper Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deeper copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder
                [old.A.length];
      size = old.size;
      for (int i=0; i<size; i++)
          A[i] = old.A[i];
   }
}
```



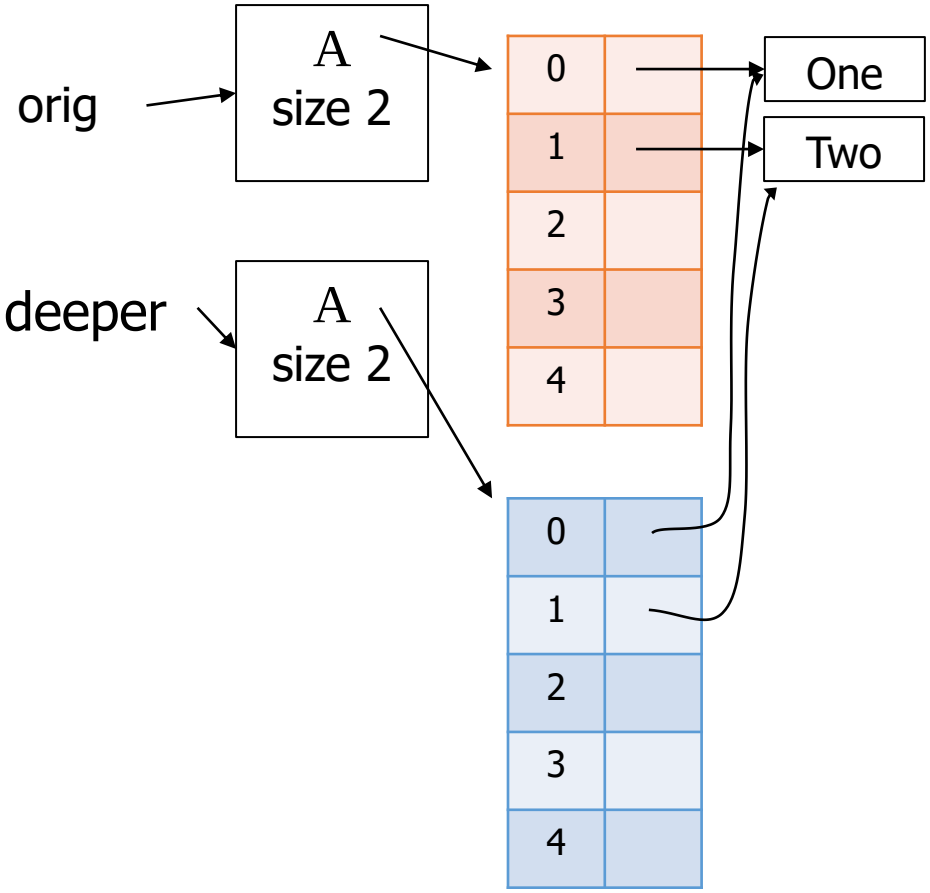
- We made a copy of the array elements
- But the references within both arrays still point to the identical StringBuilder objects

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));
```

Example 2: Deeper Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deeper copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder
                [old.A.length];
      size = old.size;
      for (int i=0; i<size; i++)
          A[i] = old.A[i];
   }
}
```

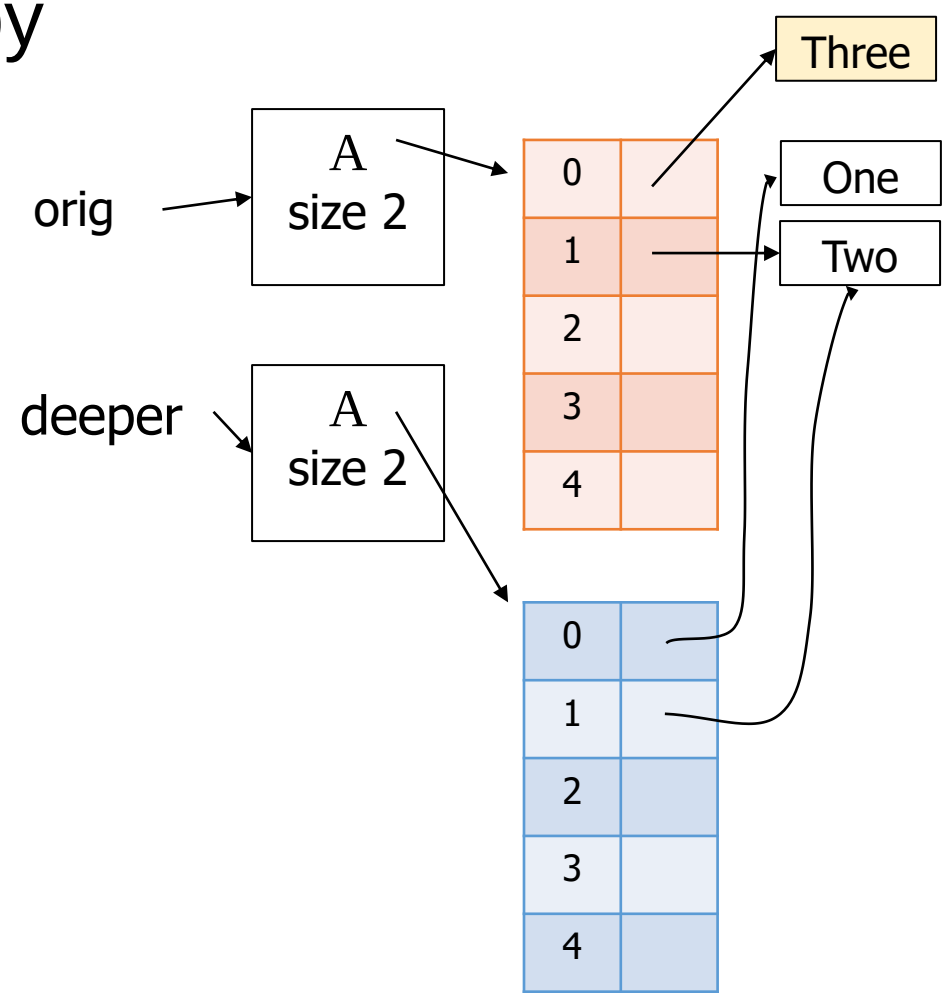
```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deeper = new SBArray(orig);
```



Example 2: Deeper Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deeper copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder
                [old.A.length];
      size = old.size;
      for (int i=0; i<size; i++)
          A[i] = old.A[i];
   }
}
```

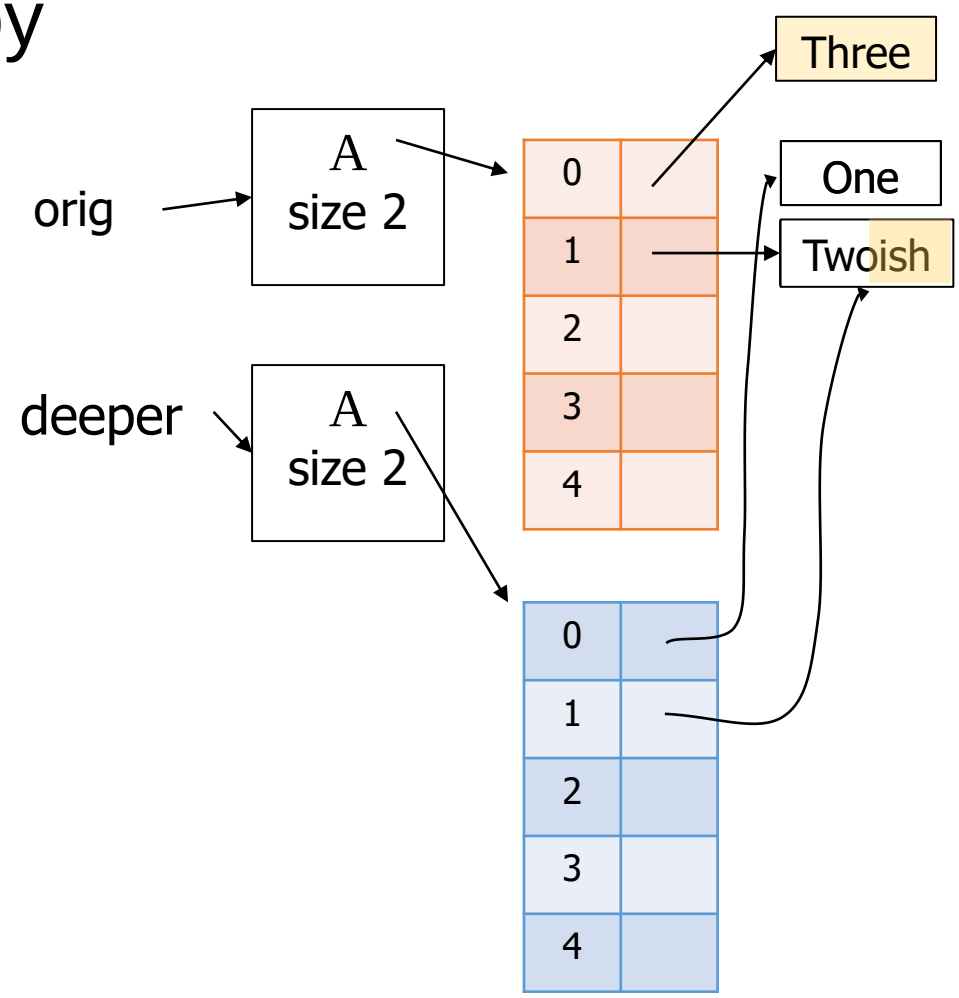
```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deeper = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
```



Example 2: Deeper Copy

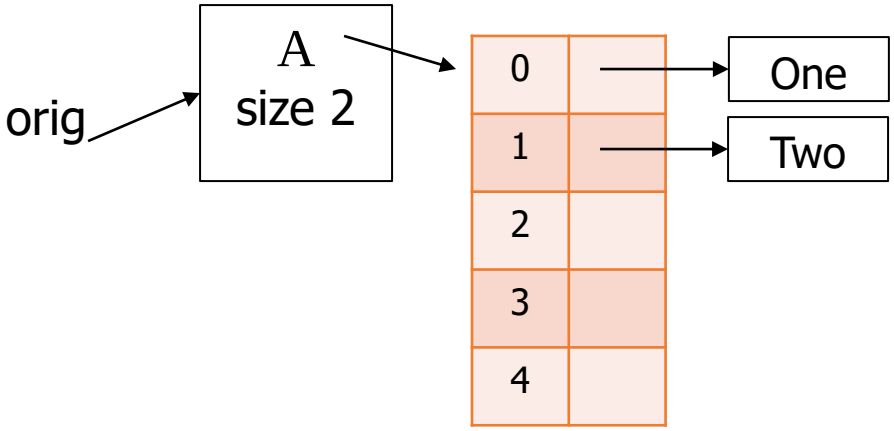
```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deeper copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder
                [old.A.length];
      size = old.size;
      for (int i=0; i<size; i++)
          A[i] = old.A[i];
   }
}
```

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deeper = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
deeper.get(1).append("ish");
```



Example3: Deep Copy

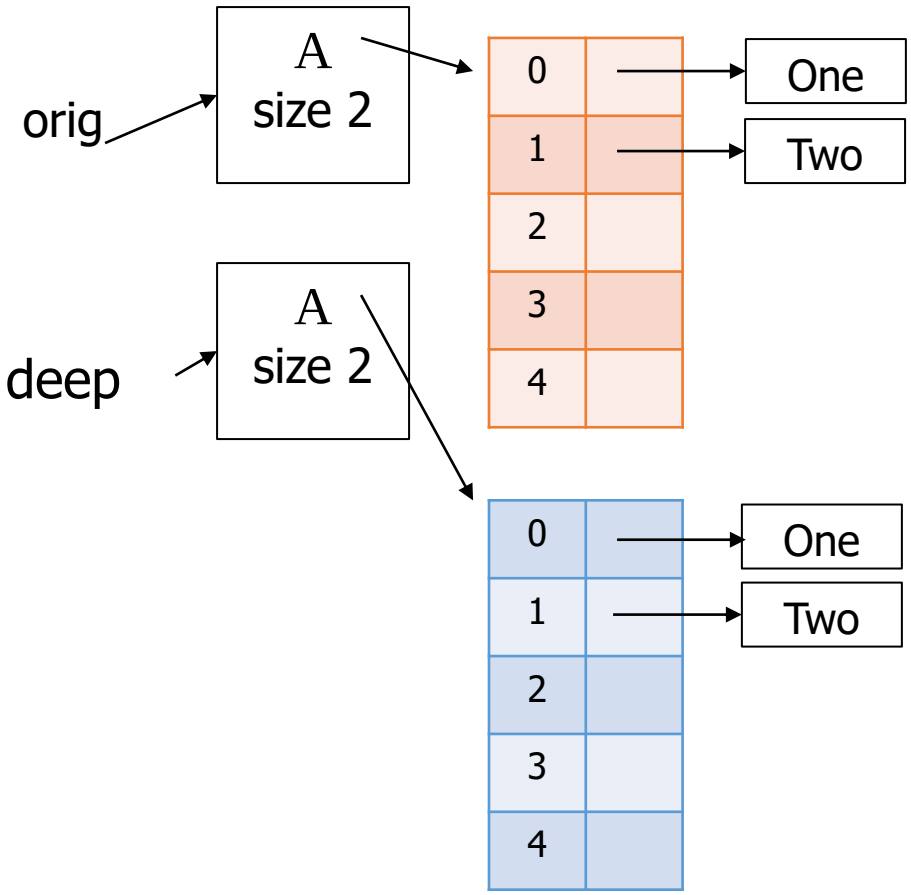
```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deep copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
              old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = new StringBuilder(
                      old.A[i]);
   }
}

SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));
```



- We made a copy of the array
- We also created a copy of all the StringBuilders stored in the array
- Original and copy are completely separated from each other

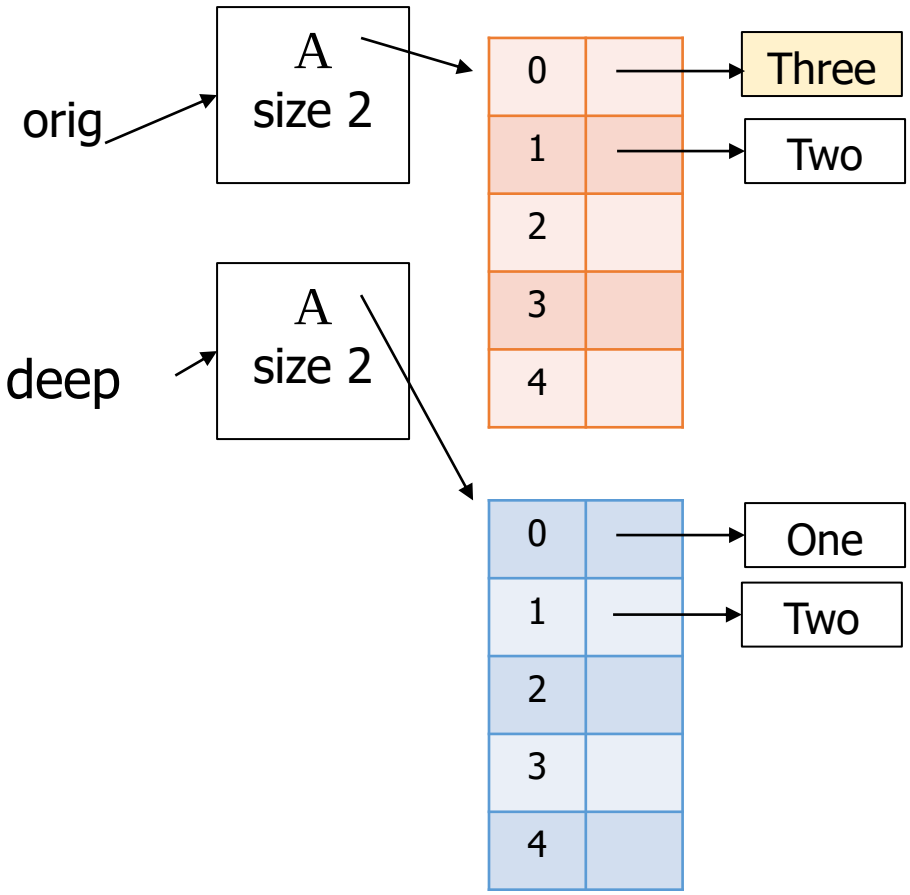
Example3: Deep Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deep copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
              old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = new StringBuilder(
                      old.A[i]);
   }
}

SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deep = new SBArray(orig);
```



Example3: Deep Copy

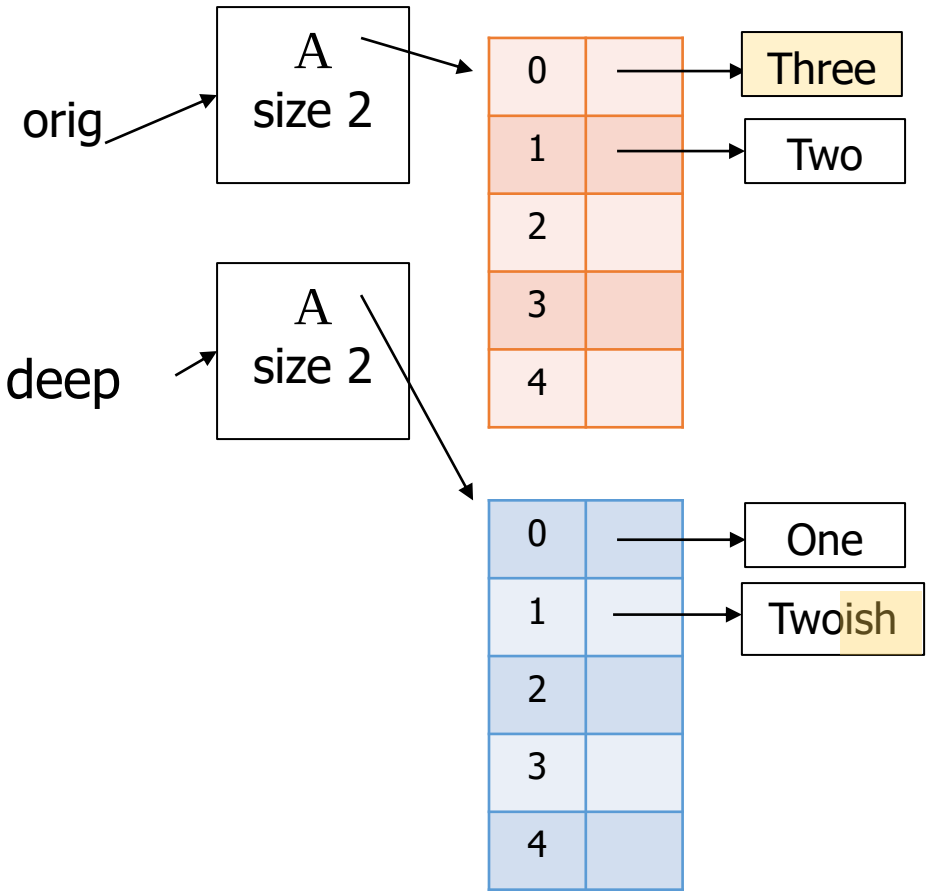
```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deep copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
              old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = new StringBuilder(
                      old.A[i]);
   }
}

SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deep = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
```



Example3: Deep Copy

```
public class SBArray
{
   private StringBuilder [] A;
   private int size;

   // deep copy
   public SBArray(SBArray old)
   {
      A = new StringBuilder[
              old.A.length];
      size = old.size
      for (int i=0; i<size; i++)
          A[i] = new StringBuilder(
                      old.A[i]);
   }
}
```

```
SBArray orig = new SBArray(5);
orig.add(new StringBuilder("One"));
orig.add(new StringBuilder("Two"));

SBArray deep = new SBArray(orig);
orig.set(0, new StringBuilder("Three"));
deep.get(1).append("ish");
```



Deep vs. Shallow Copy: summary

- In general, (true) deep copying is more difficult than shallow copying
  - We need to follow all references in the original and make copies for the copy
  - This could be several levels deep
  - Ex: A linked list
    - The linked list object has a reference to front node
    - A shallow copy would only copy this single reference
    - A deep copy would have to traverse the entire list, copying each node AND copying the data in each node AND …
    - We don't know it is truly deep unless all copies made are deep

- See:

Example2.java

SBArray.java