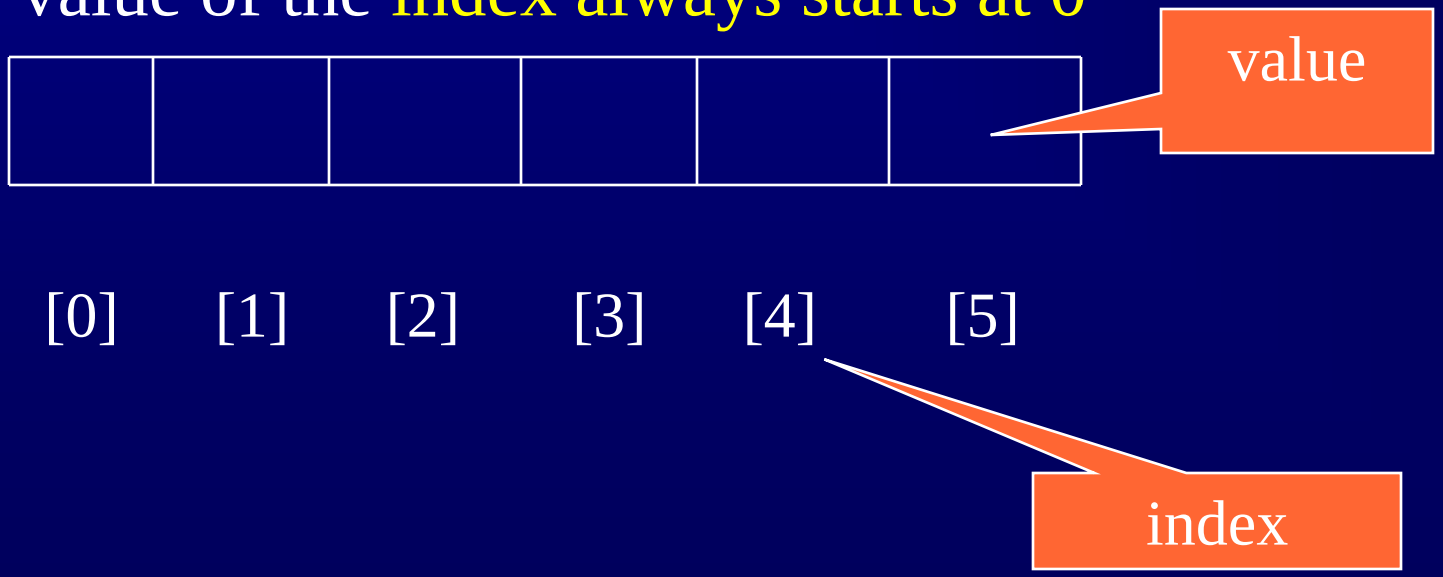
ArrayLists

- Arrays
- ArrayList Class
- Lists

Array Components

- Static data structure
  - Fixed size in memory
- Each element has
  - value & type
  - an index 0 .. length-1
- The value of the index always starts at 0



Using Arrays

```
double grades[] = new double[100];
String cities[] = new String[50];
```

- Once you access a particular element, it's like any other variable
- Examples

```
if (grades[k] >= 90)
max = grades[k];
cities[k] = "Pittsburgh";
System.out.println(cities[k] );
```

Array Access

- Sequential access
  - print, search, scan, one element at the time
  - use a loop to step through the indexes
- Direct or random access
  - when the value of the index is known

Warnings

- Must validate the value of the index
  - Remember to check for ArrayIndexOutOfBounds exception
- Remember to read/write one element of the array at the time

ArrayLists

- It's called a *collection* in Java
  - Very similar to a *list* in Python

- It is a Java Class
  - It has constructors
  - It has methods
  - It is dynamic

Arrays vs ArrayLists

- Arrays
  - Reference type but has no methods
  - Stores primitives or references types (objects)
  - Size fixed when created
  - Has special syntax
  - No need for *casting*

- ArrayLists
  - Object with methods
  - Stores references types (objects) only
  - Grows as add elements and shrinks as remove elements
  - Has constructors and methods
  - May require *casting*

Using ArrayLists

```
ArrayList aList = new ArrayList();
```

- It has several built-in methods:
  - add(object), add(index, object)
    - adds element (an object) to the end of the list
    - adds element (an object) at specified index, relocating the rest
  - set(index, object)
    - changes element (an object) at specified index
  - get(index)
    - returns an element (an object) at specified index
  - remove(index)
    - deletes an element (an object) at specified index
  - size()

How does the ArrayList work?

- We could provide our own implementation or use the corresponding Java class.
- If providing our own implementation, must make sure to follow the same specifications (behavior).
- In general, be aware that the implementation of a data structure can have different models. Consequently, similar methods may have different runtimes!

Generic vs nongeneric

- *nongeneric* lists (ArrayList) contains values of multiple types

```
ArrayList aList = new ArrayList();
aList.add(33);  //adds an integer
aList.add("today");
```

- *generic* lists (ArrayList) contains values of the same type

```
ArrayList<String> aList = new ArrayList<String>();
aList.add(33);  //adds a String
aList.add("today");
```

Why do we use generic lists?

- with *generics* we can do

```
ArrayList<Apple> theApples = new ArrayList<Apple>();
Apple oneApple = theApples.get(0);
```

- to do the same without *generics*

```
ArrayList theApples = new ArrayList();
Apple oneApple = (Apple) theApples .get(0);
```

Let the compiler keep track of types parameters, perform the type checks and the casting operations: the compiler guarantees that the casts will never fail.

Parameterized Types

- ArrayList<E> where E is any object type. E cannot be a primitive type!

```
ArrayList<String> cities;
cities = new ArrayList<String> ();

ArrayList<Double> grades;
grades = new ArrayList<Double> ();
```

Why do we need these?

Storing values in ArrayLists

- Can only store *reference types* (no primitives!)
- ArrayList<String>
  - Convert int to String and then back to int
- ArrayList<int[]>
  - Each element is an array of exactly 1 int
- ArrayList<Integer>
  - Use the Integer wrapper class

Auto-boxing and Auto-unboxing

Integer x =3; object or primitive?

- This is the same as Integer x = new Integer(3);
- It's called *auto-boxing*

int n = x; object or primitive?

- This is the same as int n = x.intValue();
- It's called *auto-unboxing*

```
ArrayList<Integer> list;
list.add(3);                // auto-boxing
int n = list.get(0);        // auto-unboxing
```

Some tricky stuff …

In Java all objects are instances of the Object Class.

```
Object x = new Integer(5);
x.toString();
x.intValue();
Integer y = x;
Integer y = (Integer) x;
String s = (String) x;
Integer z = null;
int n = z;
```

Copying arrays

```
int numbers [] = new int [20];
 …fill the array …
int numbers2[] = numbers;
```

- How many arrays do we really have?

```
int numbers [] = new int [20];
 …fill the array …
int numbers2[] = (int []) numbers.clone();
```

- How many arrays do we really have?

Copying in Java

- The assignment operator (=) in Java = is a built-in operator. It comes with every Java class!

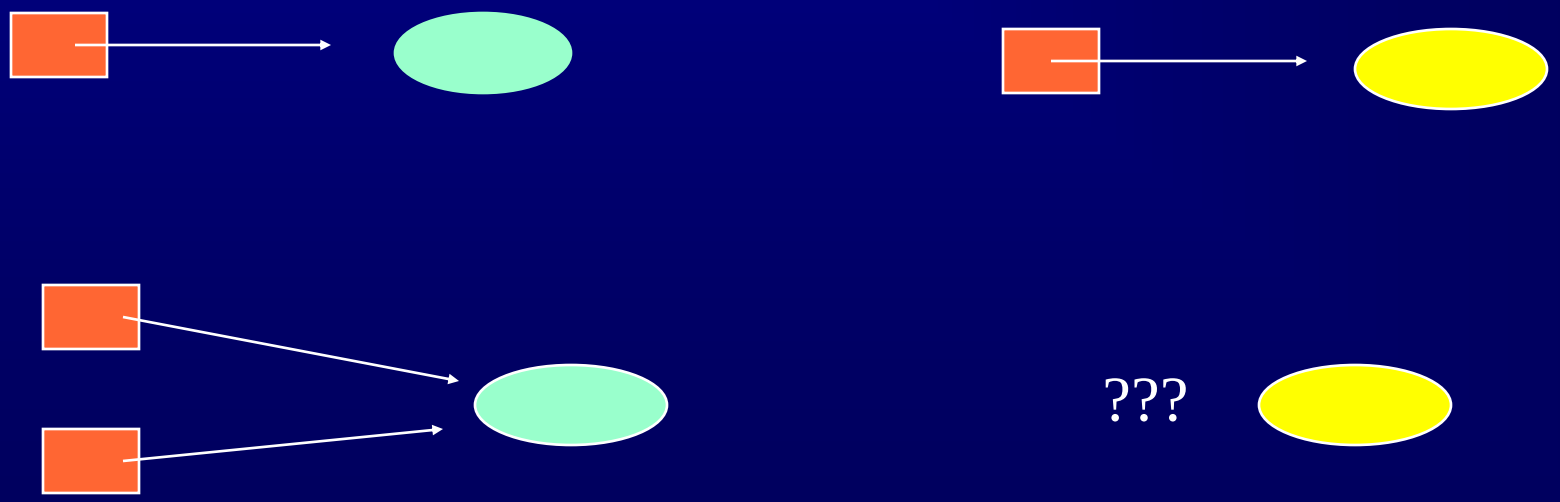
- In Java, we have two behaviors when we copy two objects
  - Shallow copying
  - Deep copying

Shallow Copying

- Shallow copying makes two reference variables to have the same value, as in

```
object1 = object2;
```

- In this case, the reference variables are the same, but the fields have different values.



Deep Copying

- Deep copying, on the other hand, copies the instance (class) variables of one object into a second object of the same class.

```
object1.field1 = object2.field1;
object1.field2 = object2.field2;
object1.field3 = object2.field3;
```

- In this case, the reference variables are different, but the fields have the same values.

Readings

- Java API for ArrayLists Class
- Java API for List Interface (not List Class!!)

Homework

- Homework #1 tonight at 11:50 pm
- Quiz #1 tomorrow in recitation
  - 10 first minutes of the recitation -- be on time!
  - String, Math, Random classes
  - 1D & 2D arrays
- Homework #2 due on 9/15