## What is an array?

- General answer: a *fixed* number of consecutive *memory* locations, all of the *same type*.
  - Can refer to all as a group by array's `name`
  - Can refer to any one by `name[position]`
    - Position is called array "subscript" or "index"
    - First position is `0` (others are "offset" from 0)
- Additional Java answer: an *object* whose purpose is to store collections of items of the same type
  - Either primitive data *values* of the same type
  - Or *references* to any one class of objects

## Arrays are objects in Java

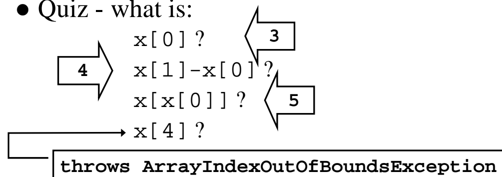
- Even a public instance variable: `length`
  - Range of positions: `0 ... length-1`
  - Length is *fixed* after created (instantiated)
- Declare, instantiate – separate steps:
  - `int x[];` // declare array of int named x
    - `int[] x;` // same thing (clear that x is an int array)
  - `x = new int[4];` // instantiate array of length 4
  - Both steps can be done with one statement: `int x[] = new int[4];`
- Assign values in a later step:
  - `x[0] = 53;` // first element set to 53

## Accessing array elements

- First, another way to instantiate:
  - And initialize at the same time

```
int x[] = { 3, 7, 4, 5 };
```

- Quiz - what is:




## Using arrays

- `for` loops are especially useful:

```
for (int i=0; i < x.length; i++)
    x[i]=getValue(); // access each x_i in order
```

- Copying can be "deep" or "shallow"
  - Shallow copy: a new reference to same array
    - `int[] a = x;` // if `x` is an `int` array already
  - Deep copy: a new array with copies of all values

```
int[] a = new int[x.length]; // same length as x
for (int i=0; i < x.length; i++)
      a[i] = x[i];
```

- Using arrays to count: RollDie.java (Fig. 7.7, p. 262)

## Enhanced **`for`** loop: Java 5

- Actually a "for each" loop

```
for (int element : array)
```

  - Reads "for each element in array"
- e.g., array of strings: `String words[] = …`

```
for (String s : words)
    System.out.println(s);
```

- Note the loop control variable is the array element itself, not its array index
  - So not applicable if index value is required
    - Like deep copy algorithm, and many others

## Some basic array operations

- Summing array elements:

```
int sum = 0; // initialize before loop starts
for (int item : x) // for each integer item in array
    sum += item;
```

- Finding a maximum (or other extreme):

```
int max = x[0]; // initialize to first value
for (int i=1; i < x.length; i++)
    if (x[i] > max) max = x[i];
```

- Printing on one row of standard output:

```
for (int item : x) System.out.print(" " + item);
System.out.println(); // newline after row is done
```

  - Q: How to print in reverse?

## More array techniques

- Finding a value

```
int i = 0, target = (some number);
boolean found = false;
while (i < x.length && !found)
   if (x[i] == target) found = true;
   else i++;
if (found) ... // know target is at x[i]
else ... // know target is not in x
```

- Removing an element – 2 cases
  - 1. If order doesn't matter, replace removed item with last item
  - 2. Otherwise, must move all trailing items forward one slot
- Inserting an element – same two basic cases in reverse

## Arrays as parameters

- Imagine hypothetical methods, `f1` and `f2`:

```
void f1(int a) { … }
void f2(int[] a) { … }
```

  and some data:

```
int x = 5, y[] = {3, 92, 17};
```

- `f1` works with a copy of a primitive value, so:

```
f1(x); // f1 cannot change x
f1(y[0]); // f1 cannot change y[0]
```

- `f2` works with a copy of a reference

```
f2(y); // f2 cannot change y, can change elements of y
```

- See PassArray.java (Fig. 7.13, p. 271)
- Note: command line arguments passed to main as array of String objects – see InitArray.java (Fig. 7.21, p. 291)

## Arrays of objects

- Arrays of objects require 3 steps to use:

```
Rectangle[] boxes; // 1. declare array of references
boxes = new Rectangle[3]; // 2. instantiate array
// 3. instantiate each object in the array:
for (int i=0; i<boxes.length; i++)
   boxes[i] = new Rectangle(5,5,5,5);
```

- Infinite applications
  - Imagine: `Car[] myFleet = ...`
    - Then: `for(...) myFleet[i].draw(g);`
- See Card and DeckOfCards (Fig. 7.9,10, pp. 266-7)
- Advice: choose array of objects over parallel arrays

## Arrays of arrays

- Arrays store anything, including arrays!
  - Not exactly multidimensional, but workable
  - e.g., `int table[][] = new int[10][4];`
    - A "table" of integers, with 10 rows and 4 columns
    - `table.length` is 10
    - Each `table[i].length` is 4, for all `i`
  - Component array sizes can vary
    - `table[2] = new int[6];` // now 3rd row has 6
- Typically use *nested for loops* to process
  - See updated GradeBook.java (Fig. 7.18, pp. 283-286)

## Handling array size limitations

- Issue: array size is fixed after construction
  - Don't always know what size to allocate at start
- Solutions
  - Allocate "way more than enough"
    - *Absolutely limits* the size of the problem – not a good idea
  - Create new, larger array, and copy values

```
if (dataSize >= data.length) {
     int[] newData = new int[2 * data.length];
     ... // here: deep copy up to (data.length - 1)
     data = newData; // copy reference (discard old array)
}
```

  - Even better – use an **`ArrayList`** instead

## **`java.util.ArrayList`**

- An array-like data structure
  - Fill with `add` method – adds element to end
    - Size is not fixed (grows *dynamically* as necessary)
  - Also an `insert` method – inserts element anywhere
    - Specify position `0..size` (like arrays) where element goes
  - Use `set` and `get` *methods* to change and access:
    - Cannot use `=` or `[]` notation like arrays
- New with Java 5 – is a generic class
  - Specify particular data type to store
  - Insures all are same type – so easier to handle

## How to use `ArrayLists`

- Declare/create ArrayList (no need to size it):
  `ArrayList<T> a = new ArrayList<T>();`
  // where `T` is an object type – not a primitive data type
- Add objects to end, or set and get specific objects
  `ArrayList<Rectangle> a = new ArrayList<Rectangle>();`
  `a.add(new Rectangle(5,5,5,5));`
  `Rectangle r = a.get(0);` // gets first
  `a.set(0, new Rectangle(0,0,10,10));` // replaces first
- Simple insert and remove too
  `a.insert(i, r);` // inserts in position `i`
  `a.remove(i);` // removes element in position `i`

## `ArrayList` and primitive types

- Must use "wrapper" classes for primitive data types
  - `Byte, Short, Integer, Long, Float, Double, Character, Boolean`
- E.g., to store `double` values in list:
  `ArrayList<Double> list = new ArrayList<Double>();`
  `list.add( new Double(17.64) );` // what really happens
  `list.add( 0.74 );` // what Java 5 "autoboxing" feature allows
- Convert back to primitive type on retrieval:
  `double d = list.get(0).doubleValue();` // what really happens
  `double d = list.get(0);` // with Java 5 "auto-unboxing" feature

## More `java.util` collections

- `List` – actually an interface
  - Defines a set of common methods like `add`, `size`, `iterator`
    - Shared by `ArrayList`, `LinkedList`, and others
  - Note: `Collections` methods to manipulate `List` objects:
    `Collections.shuffle(list);` // randomly shuffles the list
    `Collections.sort(list);` // assuming items are `Comparable`
- `Stack` – a LIFO (last in, first out) data structure
  `Stack<String> s = new Stack<String>();`
  `s.push("dog"); ...` // push objects onto top of stack
  `while (!s.isEmpty())`
  `... s.pop();` // removes/returns top object
  - e.g., use a stack to print array in reverse order (ReverseArgs.java)
- Also trees, sets, hash tables, … – covered in CS 20