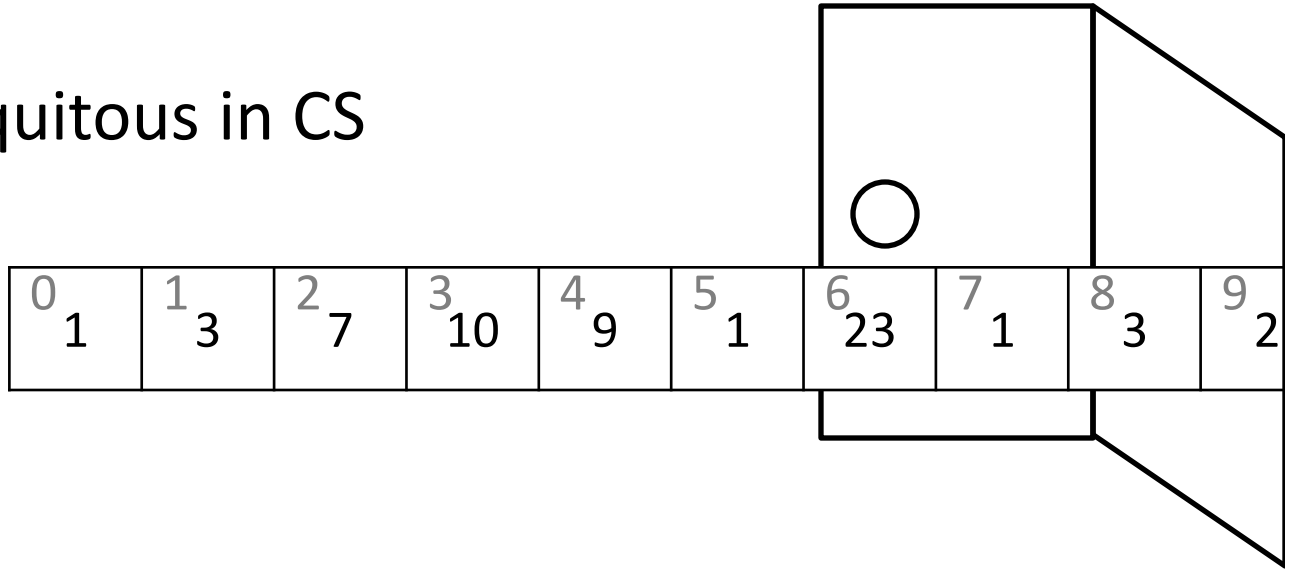
Announcements/Follow-ups

- Midterm 2 graded
  - Average was 52/80 (63%), Std. Dev. was 12
- P5 and Lab8
  - P5 due tomorrow
  - Very light Lab8
    - Practice with collections
    - Due at end of lab period
- P6 posted tomorrow
  - Due Wednesday July 24
- Quiz 4 on Thursday
  - Study questions posted today

“Goodbye” Arrays

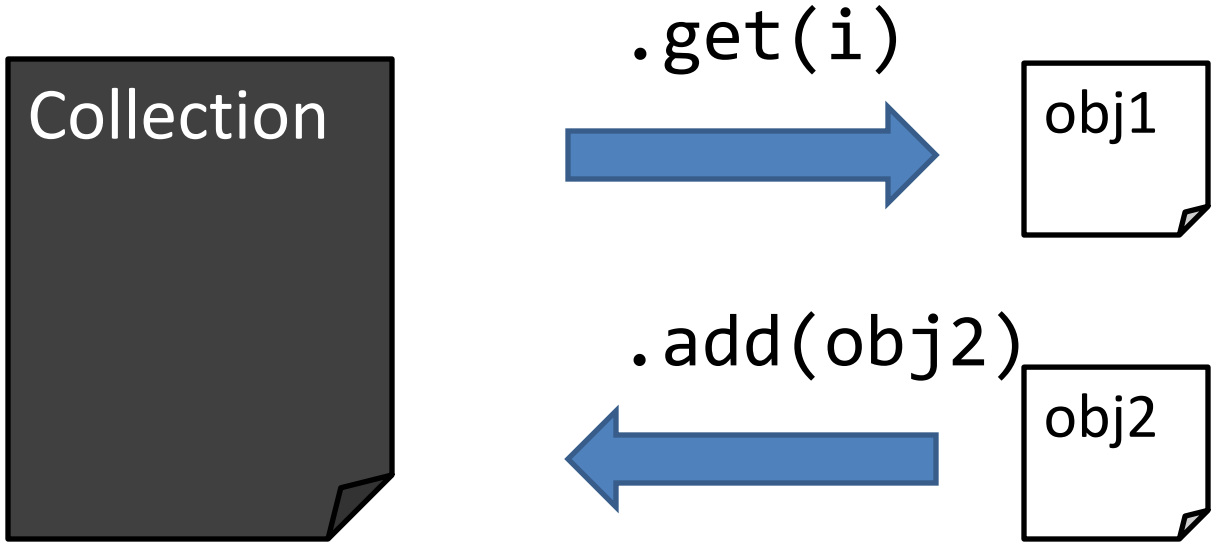
You will see arrays again:

- P6
- Ubiquitous in CS



Hello Collections

- A “Collection” is an encapsulated library used to store collections of objects
- Java contains many built-in collections.



Collections

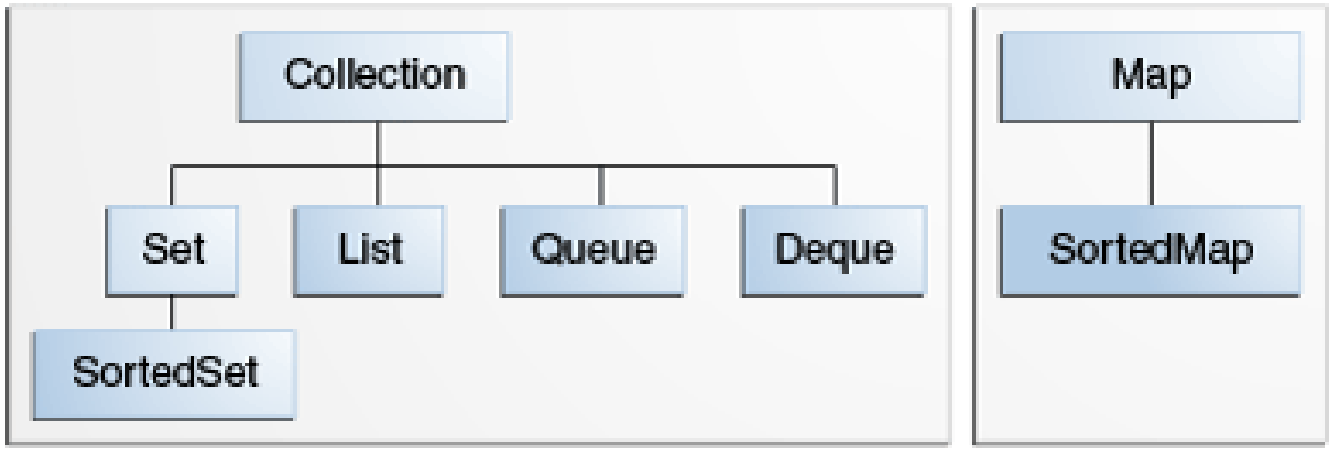
- Some advantages of collections:
  - Code reuse
    - No need to re-invent the wheel for your own applications
  - Interoperability
    - Independently designed libraries can exchange data by storing it in standard built-in collections
  - Performance and Reliability
    - The built-in collections have been well designed and thoroughly tested

Java Collections Framework

- The various collections provided by Java are organized into the “Java Collections Framework”.
  - Interfaces: These define the APIs for each type of collection.
  - Implementations: Each interface is implemented by various classes
  - Algorithms: The framework supports various algorithms for handling collections, such as searching and sorting.

Collection Interfaces

- The collection interfaces are organized into a hierarchy



http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Collection Interfaces

- The collection interfaces are organized into a hierarchy
  - List: an ordered list of objects (perhaps with repeats)
  - Set: an unordered list of distinct objects (without repeats)
    - Although SortedSet is ordered for performance gains
  - Queue: A list of objects intended for sequential processing
    - Next entry to process is removed from one end of the list
  - Deque (“Deck”): Like a queue, but next entry can be removed from either end of the list
  - Map: A collection of (key, value) pairs
    - E.g. (name, telephone number)
    - Keys are used to look up values: like indices, but keys do not need to be numbers

ArrayList

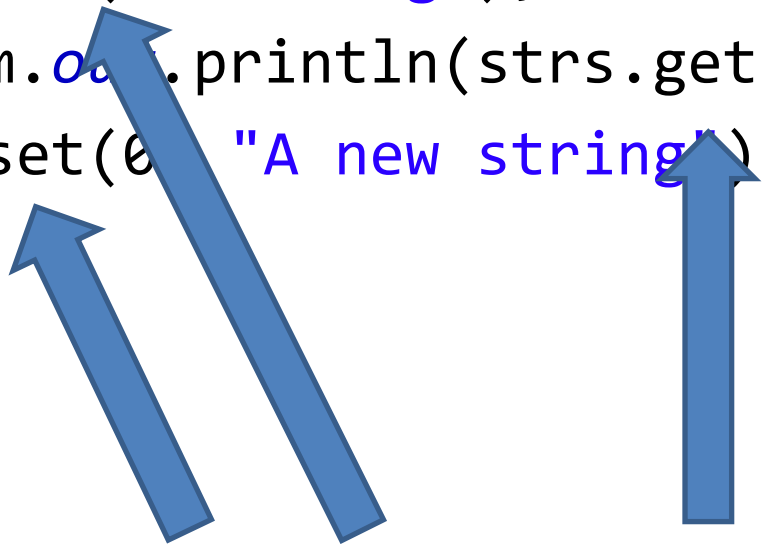
- One (of several) collections that implement the List interface.
  - General-purpose
    - Basic data access: Size(), get(), set(), indexOf()
    - Resizable: add(), remove()
  - Fast
    - Direct, immediate access to data (like an array), no linked-list traversal

Basic usage

```
ArrayList<String> strs = new ArrayList<String>();
strs.add("A string");
System.out.println(strs.get(0));
strs.set(0, "A new string");
```

Basic usage

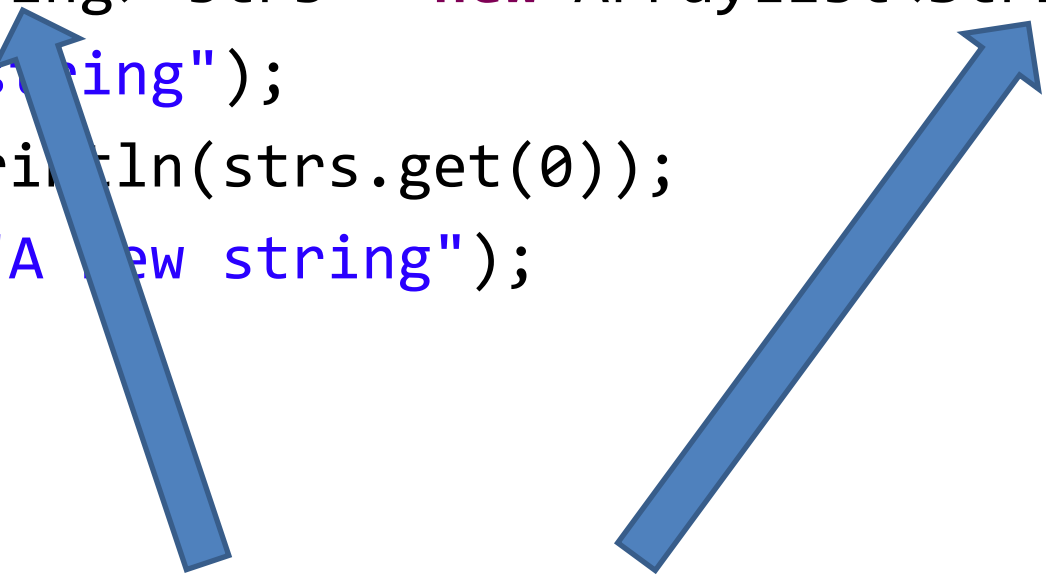
```
ArrayList<String> strs = new ArrayList<String>();
strs.add("A string");
System.out.println(strs.get(0));
strs.set(0, "A new string");
```



Methods can be called as usual

Basic usage

```
ArrayList<String> strs = new ArrayList<String>();
strs.add("A string");
System.out.println(strs.get(0));
strs.set(0, "A new string");
```



The data-type that will be stored in the collection

Generics

- Hard-coding data-types: IntList, CharList, StringList, …
  - Redundant class definitions
  - Maintanance/Extension nightmare
  - ObjectList example
- Soft-coding data-types:
  - List<Integer>, List<Character>, List<String>, …
  - List is only defined once

Generics

"Generic type" "Type argument"

```
ArrayList<String>
```

- The ArrayList class itself is a "generic type"
- The class of each element is a "type argument"
- Any *non-primitive* type argument is allowed
  - One major motivation for primitive wrappers
- Like arrays, all elements must be the same type
  - Although polymorphism is still possible
- ArrayList example

ArrayList capacity

- An ArrayList can have a capacity larger than its current size
  - More efficient for frequent resizing (add, remove)
  - One of the ArrayList constructors lets you specify an initial capacity
  - The method ensureCapacity() can be called after construction
  - Capacity Example

Other collections

- Stack
  - Push(): put a new element on top of the stack
  - Pop(): remove an element from the top of the stack
  - Peek(): return the top element without removing it from the stack
  - Stack example
    - Standard equals

Other collections

- LinkedList
  - May be preferable to ArrayList for lots of resizing
  - LinkedList example
- HashMap
  - Good when non-numeric keys are most natural
  - HashMap example

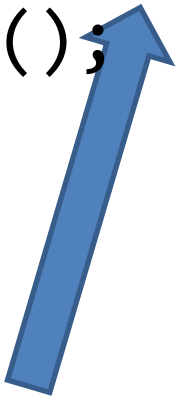
The “for-each” loop

```
ArrayList<MyClass> myCollection;
…
for(MyClass myObj : myCollection) {
      myObj.myMethod();
}
```

- For-each loops are another looping construct that is convenient for processing collections
- Useful when index is unimportant

The “for-each” loop

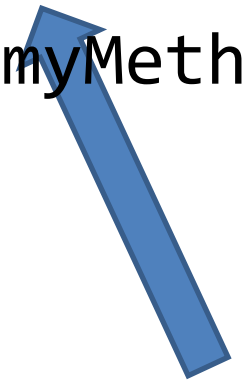
```
ArrayList<MyClass> myCollection;
…
for(MyClass myObj : myCollection) {
      myObj.myMethod();
}
```



A collection to process iteratively

The “for-each” loop

```
ArrayList<MyClass> myCollection;
…
for(MyClass myObj : myCollection) {
      myObj.myMethod();
}
```



A variable that will refer to the current element during iteration

The “for-each” loop

```
ArrayList<MyClass> myCollection;
…
for(MyClass myObj : myCollection) {
      myObj.myMethod();
}
```

What to do with that element (independent of loop counter)

The “for-each” loop

```
ArrayList<MyClass> myCollection;
…
for(MyClass myObj : myCollection) {
      myObj.myMethod();
}
```

- PolymorphicArrayList example

ConcurrentModificationException

- Elements of a collection can be changed during a for-each loop
- But the collection itself cannot be:
  - No add(), remove(), etc.
  - Attempting to change the collection will throw a "ConcurrentModificationException"
  - ForEachAndIterators example
  - Work-arounds: Normal for-loops, iterators

Iterators

- Iterators are the standard way to simultaneously iterate over a collection and remove elements.
- Each collection has a method iterator() which returns the associated iterator.
- Iterators are generic: Their type argument is the same as the associated collection.

The iterator interface

- The iterator interface has three methods:
  - next()
    - Returns the next element in the iteration
    - Ordering of elements depends on the collection
      - Could be random (e.g. for Sets or Maps)
  - hasNext()
    - Returns a boolean: whether or not there are remaining elements to iterate over
  - Remove()
    - No return type, no arguments
    - Removes the element that was most recently returned by next()

Iterator usage

```
//Get the iterator
Iterator<item_type> iter = collection.iterator();
//Iterate while there are more items to cover
while(iter.hasNext()) {
      //Get the next item in the collection
      item_type nextOne = iter.next();
      //Process the item
      if(remove_condition) {
            //Remove the item when appropriate
            iter.remove();
      }
}
```

- ForEachAndIterators example