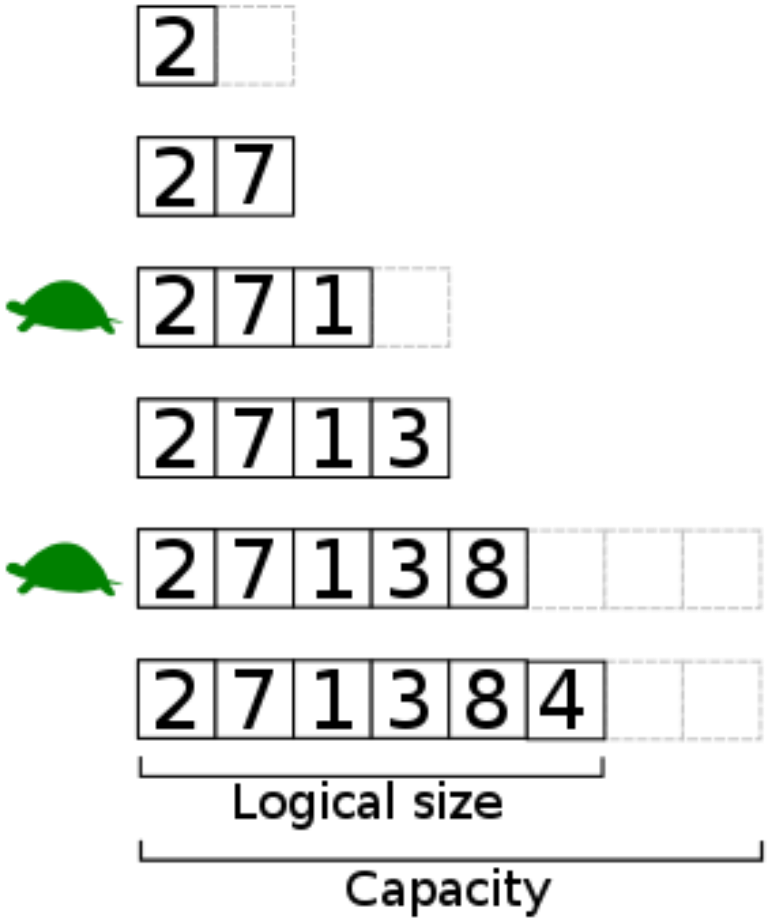
# Dynamically sized arrays




# Overview

- The Java Library
  - Many handy classes and methods
  - Importing a package
- Dynamically sized arrays
  - Java `ArrayList`
  - Wrapper classes for primitive types

# The problem with arrays

- Normal Java arrays:
  - Can hold primitive types
  - Can hold reference types
  - Must declare size when we create

```
int       n = StdIn.readInt();
Charge [] a = new Charge[n];
```

```
int       n = StdIn.readInt();
double [] x = new double[n];
double [] y = new double[n];
```

- What if we need to add another element?
- What if we want to remove an element?
- What if we don't know how big to create?

# Java library

- Java library
  - Tons of useful classes you can use
  - Only the most important are automatically available without excessive typing:
    - Things like `String`, `System.out`, etc.
- Today:
  - Look at one particular class: `ArrayList`
  - Provides dynamically sized arrays

# Java packages

- Packages
  - A collection of classes under one *namespace*
    - Avoids problems if multiple classes have same name
  - Common stuff in `java.lang` package

```
// Two ways to declare a String
String s = "hello world!";
java.lang.String s2 = "hello world!";
```

The `String` class lives in a package called `java.lang`, qualifying it is optional for this package

  - `ArrayList` is in a the `java.util` package
    - Add line outside of class: `import java.util.ArrayList;`

# Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

**See:**

Description

| Interface Summary | |
|---|---|
| Collection | The root interface in the *collection hierarchy*. |
| Comparator | A comparison function, which imposes a *total ordering* on some collection of objects. |
| Enumeration | An object that implements the Enumeration interface generates a series of elements, one at a time. |
| EventListener | A tagging interface that all event listener interfaces must extend. |
| Iterator | An iterator over a collection. |
| List | An ordered collection (also known as a *sequence*). |
| ListIterator | An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. |
| Map | An object that maps keys to values. |
| Map.Entry | A map entry (key-value pair). |
| Observer | A class can implement the `Observer` interface when it wants to be informed of changes in observable objects. |
| RandomAccess | Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access. |
| Set | A collection that contains no duplicate elements. |
| SortedMap | A map that further guarantees that it will be in ascending key order, sorted according to the *natural ordering* of its keys (see the `Comparable` interface), or by a comparator provided at sorted map creation time. |
| SortedSet | A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time. |

| Class Summary | |
|---|---|
| AbstractCollection | This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface. |
| AbstractList | This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). |
| AbstractMap | This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface. |
| AbstractSequentialList | This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list). |
| AbstractSet | This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface. |
| ArrayList | Resizable-array implementation of the `List` interface. |
| Arrays | This class contains various methods for manipulating arrays (such as sorting and searching). |

# Reversing lines in a file

- Goal: Print lines from `StdIn` in reverse order
- Problem: Don't know how many things to expect

```
Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
Delaware
Florida
...
```

states.txt

```
java ReverseLines < states.txt
Wyoming
Wisconsin
West Virginia
Washington
Virginia
Vermont
Utah
Texas
...
```

# Reversing lines in a file

"I want to type ArrayList instead of java.util.ArrayList everywhere."

"I want an empty ArrayList and I promise to only put String objects in it."

"Please add this String to my ArrayList."

```
import java.util.ArrayList;

public class ReverseLines
{
   public static void main(String[] args)
   {
      ArrayList<String> lines = new ArrayList<String>();

      while (!StdIn.isEmpty())
         lines.add(StdIn.readLine());
      for (int i = lines.size() - 1; i >= 0; i--)
         System.out.println(lines.get(i));
   }
}
```

"How many things are in my list?"

"Please return the $i^{th}$ element of the array."

# Reversing lines in a file

```
java ReverseLines < states.txt
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 50, Size: 50
        at java.util.ArrayList.RangeCheck(ArrayList.java:547)
        at java.util.ArrayList.get(ArrayList.java:322)
        at ReverseLines.main(ReverseLines.java:14)
```

```
import java.util.ArrayList;

public class ReverseLines
{
   public static void main(String[] args)
   {
      ArrayList<String> lines = new ArrayList<String>();

      while (!StdIn.isEmpty())
         lines.add(StdIn.readLine());
      for (int i = lines.size(); i > 0; i--)
         System.out.println(lines.get(i));
   }
}
```

Just like normal arrays, `ArrayList` objects use 0-based indexing.

The index to the `get()` instance method must be in `[0, size() - 1]`.

# Reversing numbers in a file

- Goal: Reverse doubles read from StdIn
- Problem: We don't know how many numbers

```
1.0 1.5
2.1 2.7 3.0
3.9
5.5
6.7
8.8
9.99
10.553
22.5
33.74
```

nums.txt

```
java ReverseNums < nums.txt
33.74
22.5
10.553
9.99
8.8
6.7
5.5
3.9
...
```

# Reversing numbers in a file: failure

- Goal: Reverse doubles read from StdIn
- Problem: We don't know how many numbers

This will not work! Java generics like `ArrayList` only take reference data types, not primitive types like `double`.

```
import java.util.ArrayList;

public class ReverseNums
{
   public static void main(String[] args)
   {
      ArrayList<double> nums = new ArrayList<double>();

      while (!StdIn.isEmpty())
         nums.add(StdIn.readDouble());

      for (int i = nums.size() - 1; i >= 0; i--)
         System.out.println(nums.get(i));
   }
}
```

# Using primitive wrapper classes: success

- Goal: Reverse doubles read from StdIn
- Problem: We don't know how many numbers

Double class wraps a primitive double data type into an object so we can put it into the ArrayList.

```
import java.util.ArrayList;

public class ReverseNums
{
   public static void main(String[] args)
   {
      ArrayList<Double> nums = new ArrayList<Double>();

      while (!StdIn.isEmpty())
         nums.add(StdIn.readDouble());

      for (int i = nums.size() - 1; i >= 0; i--)
         System.out.println(nums.get(i));
   }
}
```

# Java primitive wrapper classes

- Wrapper classes
  - Provide a way to use primitives with generics like `ArrayList`
  - Usually primitive type capitalized
  - Stick to primitives unless you actually need a wrapper
    - Less overhead

| Primitive type | Wrapper class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

# Autoboxing

- Autoboxing
  - Java 5.0 converts to/from wrapper classes as needed

This works even though `StdIn.readDouble()` returns a primitive `double` but the `ArrayList` requires a `Double` object.

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

while (!StdIn.isEmpty())
   nums.add(StdIn.readDouble());

for (int i = nums.size() - 1; i >= 0; i--)
   System.out.println(nums.get(i));
```

# Adding and removing

- Adding an element
  - Method: `add(Object o)`
    - Appends the specified object to the end of the list
    - Size of list will increase by one after calling
- Removing an element by index
  - Method: `remove(int index)`
    - Removes element at the specified position in the list
    - Shifts subsequent elements to the left (subtracts one from their indices)
    - Size of list will decrease by one after calling

# Removing (cont'd)

- Removing a specific element
  - Method: `remove(Object o)`
    - Removes the first occurrence of the specified element from the list if present
    - Returns true if the list contained the element
    - Shifts subsequent elements to the left (subtracts one from their indices)
    - Size of list will decrease by one if element found
- Removing all elements
  - Method: `clear()`

# ArrayListExample

```
import java.util.ArrayList;
public class ArrayListExample
{
   public static void main(String[] args)
   {
      ArrayList<String> names = new ArrayList<String>();
      names.add("alice");
      names.add("bob");
      names.add("bob");
      names.add("carol");
      System.out.println(names);

      names.remove(2);
      System.out.println(names);

      names.remove("bob");
      System.out.println(names);
      names.remove("bob");
      System.out.println(names);

      names.clear();
      System.out.println(names);
   }
}
```

```
[alice, bob, bob, carol]
[alice, bob, carol]
[alice, carol]
[alice, carol]
[]
```

# Removing in a loop: failure

```
import java.util.ArrayList;
public class ArrayListRemoveLoop
{
   public static void main(String[] args)
   {
      ArrayList<String> names = new ArrayList<String>();
      names.add("alice");
      names.add("bob");
      names.add("bob");
      names.add("carol");
      System.out.println(names);

      for (int i = 0; i < names.size(); i++)
      {
         if (names.get(i).equals("bob"))
            names.remove(i);
      }
      System.out.println(names);
   }
}
```

```
[alice, bob, bob, carol]
[alice, bob, carol]
```

This doesn't work since when we remove the first "bob", the list is shortened by one inside the loop. We end up skipping over the second "bob".

# Removing in a loop: success

```
import java.util.ArrayList;
public class ArrayListRemoveLoop
{
   public static void main(String[] args)
   {
      ArrayList<String> names = new ArrayList<String>();
      names.add("alice");
      names.add("bob");
      names.add("bob");
      names.add("carol");
      System.out.println(names);

      for (int i = names.size() - 1; i >= 0; i--)
      {
         if (names.get(i).equals("bob"))
            names.remove(i);
      }
      System.out.println(names);
   }
}
```

```
[alice, bob, bob, carol]
[alice, carol]
```

Going backwards through the list fixes the bug. Removing something in the loop doesn't affect what elements we'll see as we move earlier in the list.

# Summary

- Java `ArrayList`
  - Like an array but extra-powerful
  - Has no fixed sized
  - Add/remove elements dynamically as needed
  - Contains objects of a specified reference type
  - Cannot hold primitive types (e.g. `double`, `int`)
    - Wrapper objects used instead (e.g. `Double`, `Integer`)
  - Be careful when you add/remove in a loop!