Recitation 3:
Wrapper classes.
The Java Collections Framework: Overview.

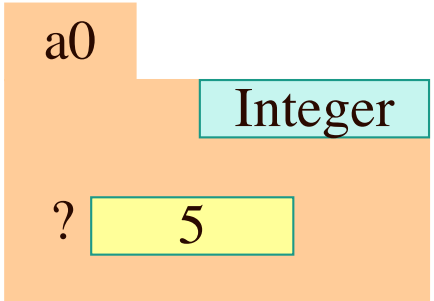
Recitation TA Names Here

We will talk a bit about wrapper classes, and auto-boxing. You have seen it before

We will talk about arrays. You have seen them briefly, our use of them here should help you understand them.

Wrapper class Integer.

There are places in a Java program where only objects are allowed, not primitive values, like int values 5 and 6.



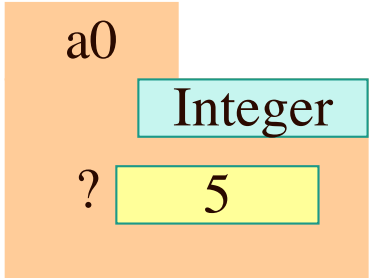
Therefore, Java has class Integer, which wraps a single **int** value. Objects of class Integer are *immutable*: cannot change the value.

Integer is called a *wrapper class*, because it wraps a value.

Here are other wrappers for a sandwich, cup cake, and spring rolls

Wrapper class Integer.

If you have to explicitly create an Integer object, don't use a constructor. Instead, use this:

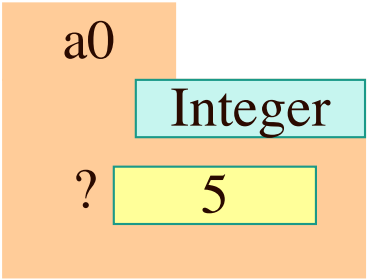



```
Integer v= Integer.valueOf(5);
```

Integer.valueOf: A static function in class Integer.
"Factory method"

Here are other wrappers for a sandwich, cup cake, spring rolls

Wrapper class Integer autoboxing and unboxing.



To make it easy to go back and forth between a primitive **int** value and a wrapper that wraps it:

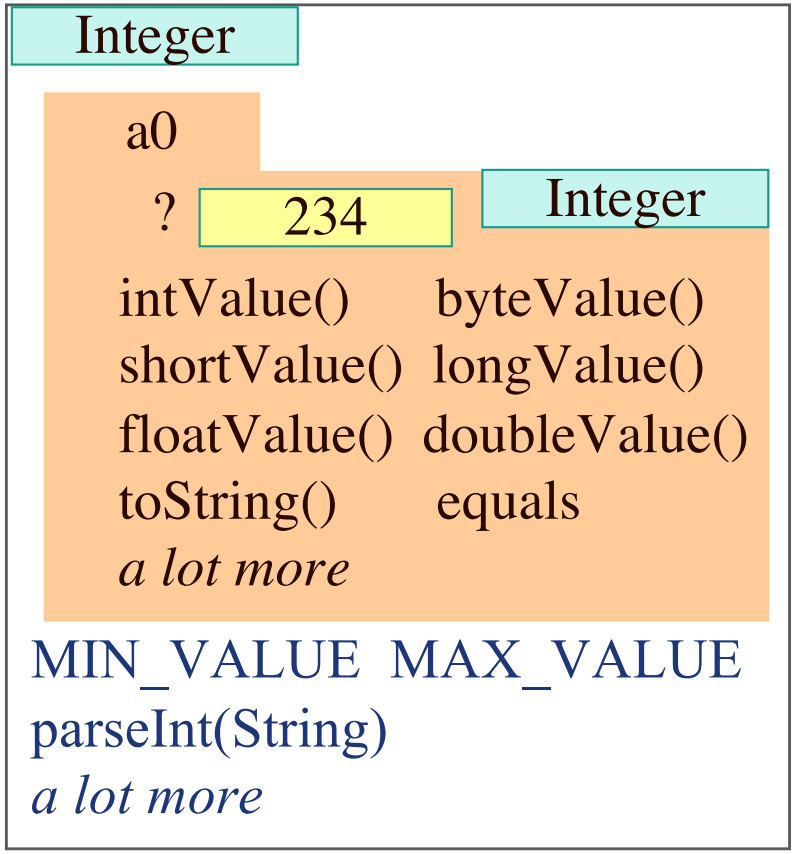
```
Integer v=  5;  // automatically create Integer object to contain 5

int x=  v;      // automatically take 5 out of object, store it in x
```

Called
auto boxing / unboxing
(why not autowrapping?)

Wrapper class Integer. Immutable

Reason for wrapper class Integer: To be able to handle an **int** as an object.

Reason for wrapper class Integer: To have a place to put methods and static components that deal with int values.



Wrapper class for every primitive type

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

Java Collections Framework

Below are some kinds of data types that
we often need to work with in a program

- Bag: bunch of values, with duplicates allowed. E.g. a bag of coins
- Set: bag with no duplicates (a unibag)
- Map: set of (key, value) pairs. Also called a
  Dictionary: set of words with meanings
- List: a bag in which the values are ordered
- Stack and Queue: lists with restricted ways of changing them

Collections Framework

The Java Collections Framework has classes that implement these things, so that you don't have to implement them yourselves.
They are in package java.util.

- Bag
- Set
- Map/Dictionary
- List
- Stack
- Queue

Our goal: Give you an intro to at least one of these so you see how helpful they can be

Motivation: Implementing a list in an array

To maintain a list of values in an array, you need **TWO** variables:
(1) the array and (2) its size

```
/** The elements of the list are in b[0..n-1]*/  class invariant
int[] b= new int[100];
int n= 0; // list initially empty; we write the list as ()

// Add 5, 8, and 2 to the list
b[n]= 5; n= n+1;
b[n]= 8; n= n+1;
b[n]= 2; n= n+1;
// list is now (5, 8, 2)
```

```
Remove the first element of the list.
e.g. change (5, 8, 2) to (8, 2)
for (int k= 1; k < n; k= k+1)
    { b[k-1]= b[k]; }
n= n-1;
```

Motivation: Implementing a list in an array

Maintaining a list of values in an array, requires **TWO** variables:
(1) the array and (2) its size

```
/** The elements of the list are in b[0..n-1]*/
int[] b= new int[100];
int n= 0; // list initially empty

// Add 5, 8, 2 to the list
b[n]= 5; n= n+1;
b[n]= 8; n= n+1;
b[n]= 2; n= n+1;
// list is now (5, 8, 2)
```

Issues

1. Size of list is limited to 100 (or whatever size of array is)
2. Have to write code to search for a value, remove a value, etc.

Instead, use Collections class ArrayList

Class ArrayList solves the problems!!

```
// Create empty list of values named ob.
// Any object can be stored in list ob.
ArrayList ob=  new ArrayList();

// Store the integers 0..n-1 in ob
for (int k= 0; k < n; k= k+1) {
    ob.add(k);
}

// Put -5 in position 0 of list
ob.add(0, -5);
```

To find documentation, search for
java 11 arraylist

Only objects can be placed in an ArrayList, so k is autoboxed

Changes the list from (0, 1, …, n-1) to (-5, 0, 1, … n-1). Since the list is maintained in an array, this causes the n elements 0, 1, …, n-1 to be moved up to make room for -5 at the beginning.

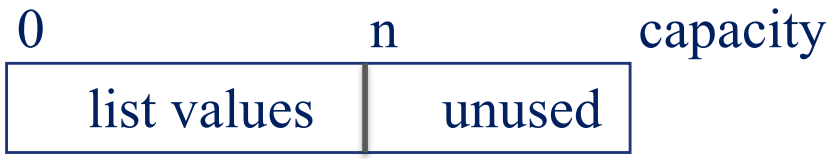
Class ArrayList

```
// create empty list named al
ArrayList al=  new ArrayList();
```

Some methods:

```
al.size()          // size of list al (number of elements in it)
al.add(5)          // append 5 to list al
al.add(3, 6)       // insert 6 as element number 3 ---pushing others up
al.get(3)          // get element number 3 (the first one is number 0)
al.contains(6)     // return true iff list al contains 6
al.remove(8)       // remove element number 8 from the list
… many more …
```

About the implementation of an ArrayList al



ArrayList al maintains a *backing array*, which contains the list of values.

al.size(): the number n of values in the list

If n = capacity and another element is about to be added, an array of twice the size (or more) is created, the values in the list are are copied into it, and the new array is used from them on.

Making the size of the new array be 2n or more, instead just n+1, is a lot more efficient. We study this when we study *algorithmic complexity* later.

ArrayList advantages

ArrayList

- You don't have maintain its size (n) —done for you.
- You don't have to worry about how big the backing array is.
- It has many methods, and you don't have to write them:
  add, contains, remove, size, and many more.

Main disadvantage: can't use array notation b[k], assignment b[k]= …

Example

```
/** Return an array of negative numbers in b.
  * Precondition: b is not null. */
public static Object[] findNegatives1(int[] b) {
    ArrayList negs= new ArrayList ();
    for (int i= 0; i < b.length; i++ ) {
        if (b[i] < 0) negs.add(b[i]);
    }
    return negs.toArray();
}
```

1. The array returned has to have type Object[] . See next slide

2. We don't know how big the final array will be, so we store the negative values in an ArrayList.

3. Method toArray creates an array of the right size, stores all the values in negs in it, and returns that array, of type Object[]

Problem: with new ArrayList()

```
ArrayList  al=  new ArrayList();
```

Creates a list of elements of class Object
We want a list of elements of class Integer

```
ArrayList<Integer> al= new ArrayList<>();
```

Type parameter. It says that al can contain only objects of class Integer

Has to do with "generics", which we discuss in lecture 6.

You can put Integer in here too, but it's not needed.

Example using generics

```
/** Return an array of negative numbers in b.
  * Precondition: b is not null. */
public static Integer[] findNegatives2(int[] b) {
    ArrayList<Integer> negs= new ArrayList <>();
    for (int i= 0; i < b.length; i++ ) {
        if (b[i] < 0) negs.add(b[i]);
    }
    return negs.toArray(new Integer[0]);
}
```

1. Returns an Integer[]

2. negs contains only Integer objects

3. The argument has type Integer[] and contains 0 elements. It is here *only* to tell toArray what type of array to return (no auto-unboxing for arrays).

Generics

```
ArrayList<Integer> negs= new ArrayList <>();
```

We will discuss generics in lecture 6.

We now introduce stacks and queues

java.util.stack

stack: a list of elements, but it can be changed only in restricted ways:

add to top,

remove from top.

Called a LIFO list (Last In First Out)

Useful methods

```
push(e)     // put e on top of stack
peek()      // look at top stack element
pop()       // remove and return top stack element
size()      // number of elements on the stack
```

Queues

queue: a list of elements, but it can be changed only two ways:

add to end,

remove from beginning.

Called a FIFO list (First In First Out)

Useful methods

```
add(e)      // put e to end of queue
peek()      // look at top queue element
remove()    // remove and return first element
size()      // number of elements on the queue
```

The Brits stand in a queue; Americans stand in a line.