CSE 21
Intro to Computing II

Lecture 9 – Inheritance (2)

ArrayList

Today

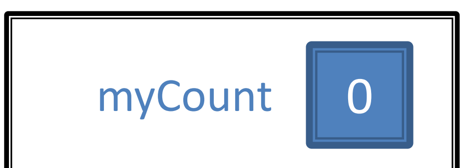
- Inheritance (2) and ArrayList
- Lab
  - Lab 10 due this week (4/8 – 4/14)
  - Lab 11 assigned this week
    - More Inheritance with Polymorphism
    - Due in one week
    - **Required** to show work to a TA (or me) for full credit
  - Project 2 due next week **POSTPONED TILL** Friday, 4/20
    - **Required** to show work to a TA (or me) for full credit
- Reading Assignment
  - Sections 7.11 – 7.14, 10.6, 9.1 – 9.5 (including participation activities)
    - Work on the **Participation Activities** in each section to receive participation grade at the end of semester (based on at least 80% completion)
    - Work on **Challenge Activities** to receive extra credit
  - Participation and Challenge activities evaluated at the end of semester

Inheritance : Motivation (review)

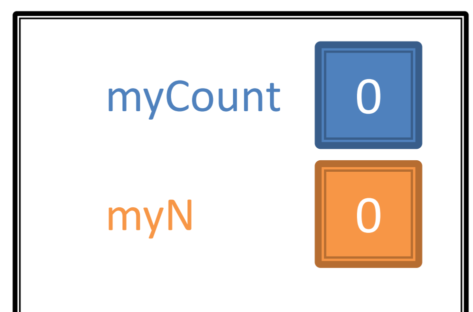
- Imagine you need an Object that is slightly different from the existing one
- Instead of re-designing an entire new object from scratch, you can inherit (or derive) the existing object and just "add" the needed modifications.
- Lets look at the Counter class
  - Counts how many times it's been incremented (++)
  - Modulo Counter inherits from Counter
    - Will reset myCount when it reaches a certain value, say **N**
  - Call the new class ModNCounter

Counter Class Example (review)

```
public class Counter {
	private int myCount;
	public Counter() {
		myCount = 0;
	}
	public void increment(){
		myCount++;
	}
	public void reset() {
		myCount = 0;
	}
	public int value() {
		return myCount;
	}
}
```




```
public class ModNCounter extends Counter {

	private int myN;
	public ModNCounter(int n){
		myN = n;
	}
	public int value(){
		// Cycles from 0 to (myN – 1)
		return myCount % myN;
	}
	public int max(){
		return myN-1;
	}
}
```

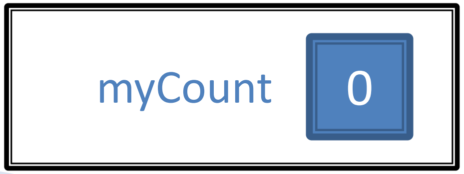


Protected Access Specifier (review)

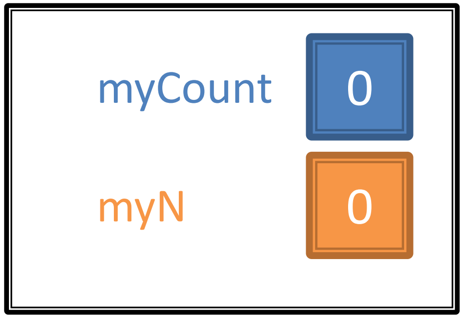
- As written, *ModNCounter* will not compile!
- The *myCount* variable is private (only accessible in the *Counter* class)
- We can fix this by making it **protected**:
  - Only classes that "extend" *Counter* can access its protected variables/methods
- Three different Access types:
  - **public**: any class can read/modify
  - **protected**: only this class, classes within the same package, and subclass descendants can read/modify
  - **private**: only this class can read/modify
  - **No modifier:** Only this class, and classes within same package. No access by subclasses.

Counter Class Example (review)

```java
public class Counter {
    protected int myCount;
    public Counter() {
        myCount = 0;
    }
    public void increment(){
        myCount++;
    }
    public void reset() {
        myCount = 0;
    }
    public int value() {
        return myCount;
    }
}
```



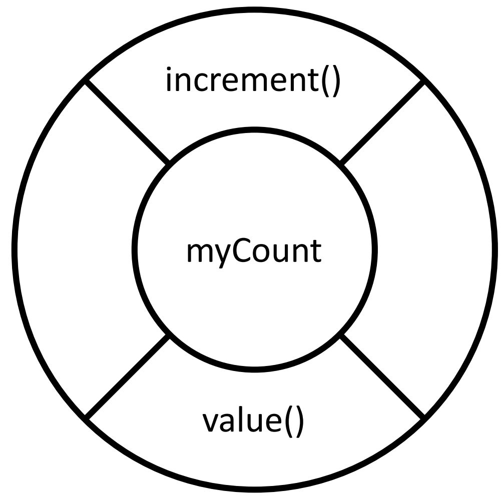

```java
public class ModNCounter extends Counter {

    private int myN;
    public ModNCounter(int n){
        myN = n;
    }
    public int value(){
        // Cycles from 0 to (myN – 1)
        return myCount % myN;
    }
    public int max(){
        return myN-1;
    }
}
```



Inheritance (review)

Superclass

class Counter

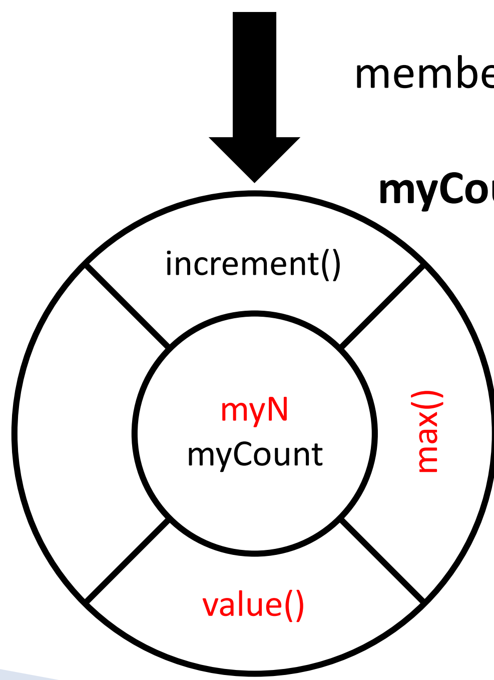


Subclass inherits members from superclass (public or protected)

myCount, increment(), value()

Subclass

class ModNCounter



Type Casting in Inheritance (review)

- Java automatically (or implicitly) *Up-Converts* some types (int → double)
- Class types using inheritance follow the same rules
- Parent class is "higher" type than the child's

```
Counter c = new ModNCounter(3);   // legal (up)
ModNCounter mc = new Counter(); // not legal
ModNCounter mc = (ModNCounter) c;        // legal (down, explicit)
```

- Anything you can do with a *Counter* you can also do with a *ModNCounter*
  - Not vice versa

Type Checking (review)

- It is OK to pass an object of a class, say ***SubClass,*** as argument to a method that expects an object of ***SubClass***'s superclass ***SupClass*** as parameter.
- In a method call, you get the version associated with the object, not the declared type.

```
ModNCounter mc = new ModNCounter(3);
Counter c = mc;
c.increment();
c.value(); // get the ModN version of value
```

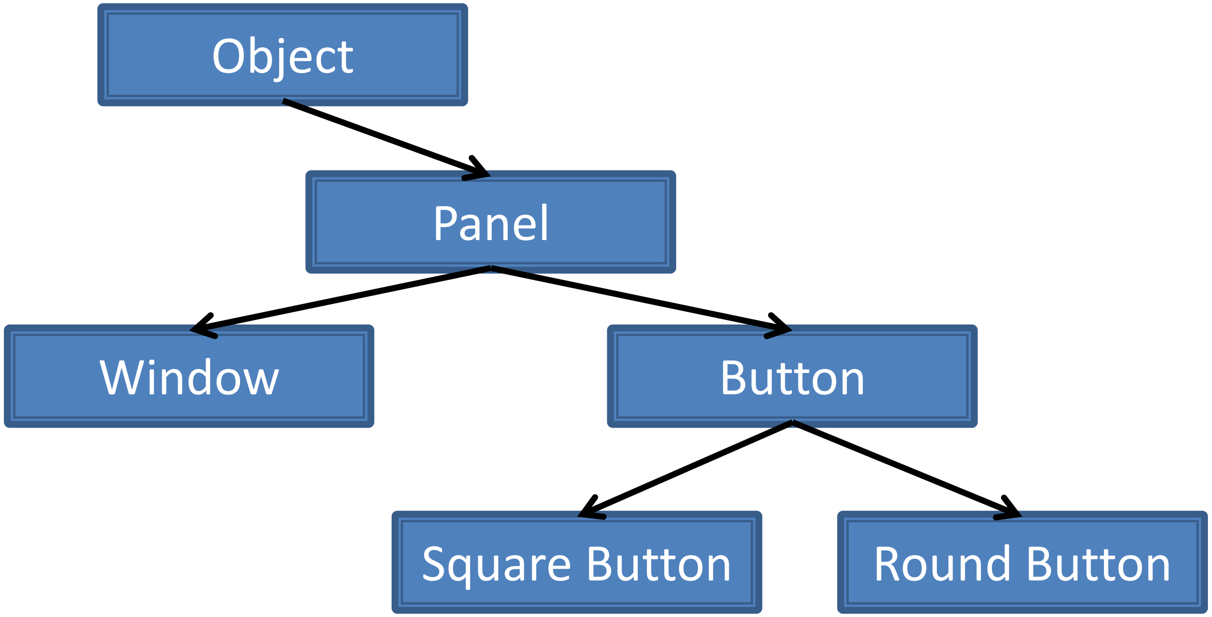
- But you cannot call a method that may not exist:

```
c.max();  // illegal, because Counter does not have max()
```

- Why? Because Java is conservative

```
mc.max();                  // OK, because mc is a ModNCounter
((ModNCounter)c).max();    // ERROR: because c may
                           // or may not be ModNCounter
```

Object is at the Top in Java (review)



- A the top of Java’s inheritance hierarchy is the special type ***Object***
- It comes with a few predefined methods such as ***toString***

Problem: a Generic Search Algorithm

- We want a generic search algorithm to search for any kind of object in an array
- The **Object** class provides an ***equals()*** method to test whether one object is equal to another
  - What does it mean when two objects are equal?
  - Simply checks if the 2 object references point to the same area of memory
    - Not very useful in practice
  - Compares the states of the 2 objects
    - Problem: different types of objects have different types of states
- We need to provide an ***equals()*** method in the class of the particular object type we are searching for
  - This is called ***Polymorphism***: a function that works on many types
  - Book definition: Determining program behavior to execute based on data type

Equals on Counters

- To check whether two *Counters* are equal:

```
public boolean equals (Object c) {
    return this.myCount == ((Counter) c).myCount;
} // Checks if myCounts are the same.
```

Down cast to Counter type

- Overriding equals for *ModNCounter*:

```
public boolean equals (Object o) {
    ModNCounter mc = (ModNCounter) o;
    return (this.myCount == mc.myCount && this.myN == mc.myN);
} // Checks if myCounts AND myN are the same.
```

A new pointer pointing at the same (typecasted) object

A Search Algorithm

- This search code will work on any array of Objects
- As long as ***equals*** is properly defined

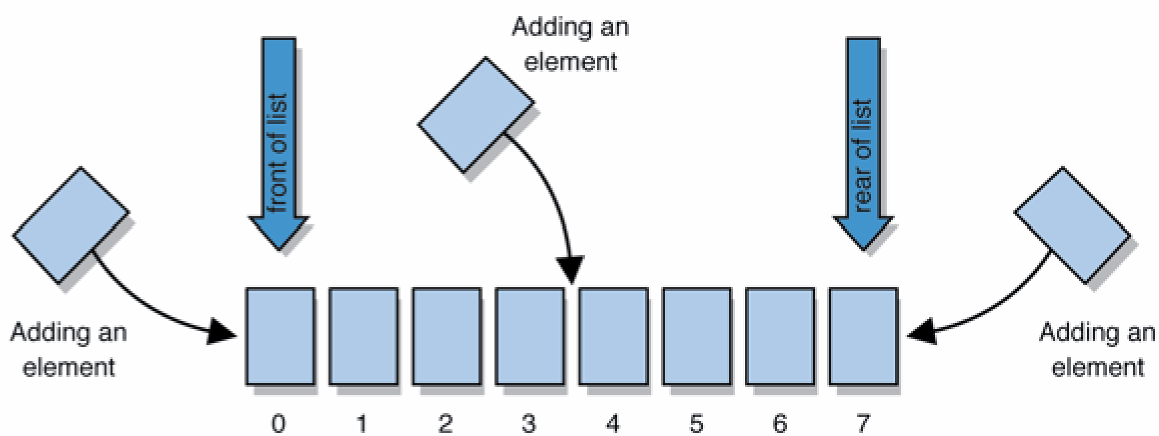
```
public class SearchAlg {
    public static int linearSearch(Object[] a, Object b) {
        for (int i = 0; i < a.length; i++) {
            if (a[i].equals(b))
                return 1; // Found
        }
        return –1; // Not Found
    }
}
```

Problems with Arrays

- The size is pre-defined
  - It cannot be changed once declared.
  - We can initialize it with a large size: *int[1000]*, but memory will be wasted if not all spaces are used.
- Difficult to insert or delete elements in the middle of array
  - Elements need to be shifted around when new elements are inserted or existing elements are deleted.

List of Objects

- An ordered sequence of elements:
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **ArrayList** object



Contents of a List

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

  { }

- You can add items to the list.
  - The default behavior is to add to the end of the list.

    {first, second, third, fourth}

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

ArrayList Methods (1)

add(**value**)	appends value at end of list
add(**index**, **value**)	inserts given value just before the given index, shifting subsequent values to the right
clear()	removes all elements of the list
indexOf(**value**)	returns first index where given value is found in list (-1 if not found)
get(**index**)	returns the value at given index
remove(**index**)	removes/returns value at given index, shifting subsequent values to the left
set(**index**, **value**)	replaces value at given index with given value
size()	returns the number of elements in list
toString()	returns a string representation of the list such as "[3, 42, -7, 15]"

ArrayList Methods (2)

addAll(**list**) addAll(**index**, **list**)	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
contains(**value**)	returns true if given value is found somewhere in this list
containsAll(**list**)	returns true if this list contains every element from given list
equals(**list**)	returns true if given other list contains the same elements
lastIndexOf(**value**)	returns last index if value is found in list (-1 if not found)
remove(**value**)	finds and removes the given value from this list
removeAll(**list**)	removes any elements found in the given list from this list
retainAll(**list**)	removes any elements not found in given list from this list
subList(**from**, **to**)	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
toArray()	returns the elements in this list as an array

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an **ArrayList**, you must specify the type of elements it will contain between **<** and **>.**
  - This is called a *type parameter* or a *generic class*.
  - Allows the same **ArrayList** class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();
names.add("John Smith");
names.add("Jerry West");
```

ArrayList vs. Array

- Construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- Storing a value

```
names[0] = "Alice";
list.add("Alice");
```

Using index values to access contents

- Retrieving a value

```
String s = names[0];
String s = list.get(0);
```

Conditionals

- Doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {
    if (names[i].startsWith("B")) { ... }
}

for (int i = 0; i < list.size(); i++) {
    if (list.get(i).startsWith("B")) { ... }
}
```

- Seeing whether the value "Bob" is found

```
for (int i = 0; i < names.length; i++) {
    if (names[i].equals("Bob")) { ... }
}

if (list.contains("Bob")) { ... }
```

ArrayList as a parameter

```
public static void methodName(ArrayList<Type> param) { … }
```

- Example:

```
// Removes all plural words from the given list.
public static void removePlural(ArrayList<String> list) {
        String str;
        for (int i = 0; i < list.size(); i++) {
                str = list.get(i);
                if (str.endsWith("s")) { // or if (list.get(i).endsWith("s")) {
                        list.remove(i);
                        i--;
                }
        }
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params) { … }
```

ArrayList of primitives?

- The type you specify when creating an **ArrayList** must be an object type; it cannot be a primitive type.

```
// illegal -- int cannot be a type parameter
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use **ArrayList** with primitive types by using special classes called wrapper classes in their place.

```
// creates a list of Integers
ArrayList<Integer> list = new ArrayList<Integer>();
```

We can make an Integer object out of int!

Wrapper classes

Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A wrapper is an object whose sole purpose is to hold a primitive value.
- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<Double>();
grades.add(3.2);
grades.add(2.7);
...
double myGrade = grades.get(0);
```