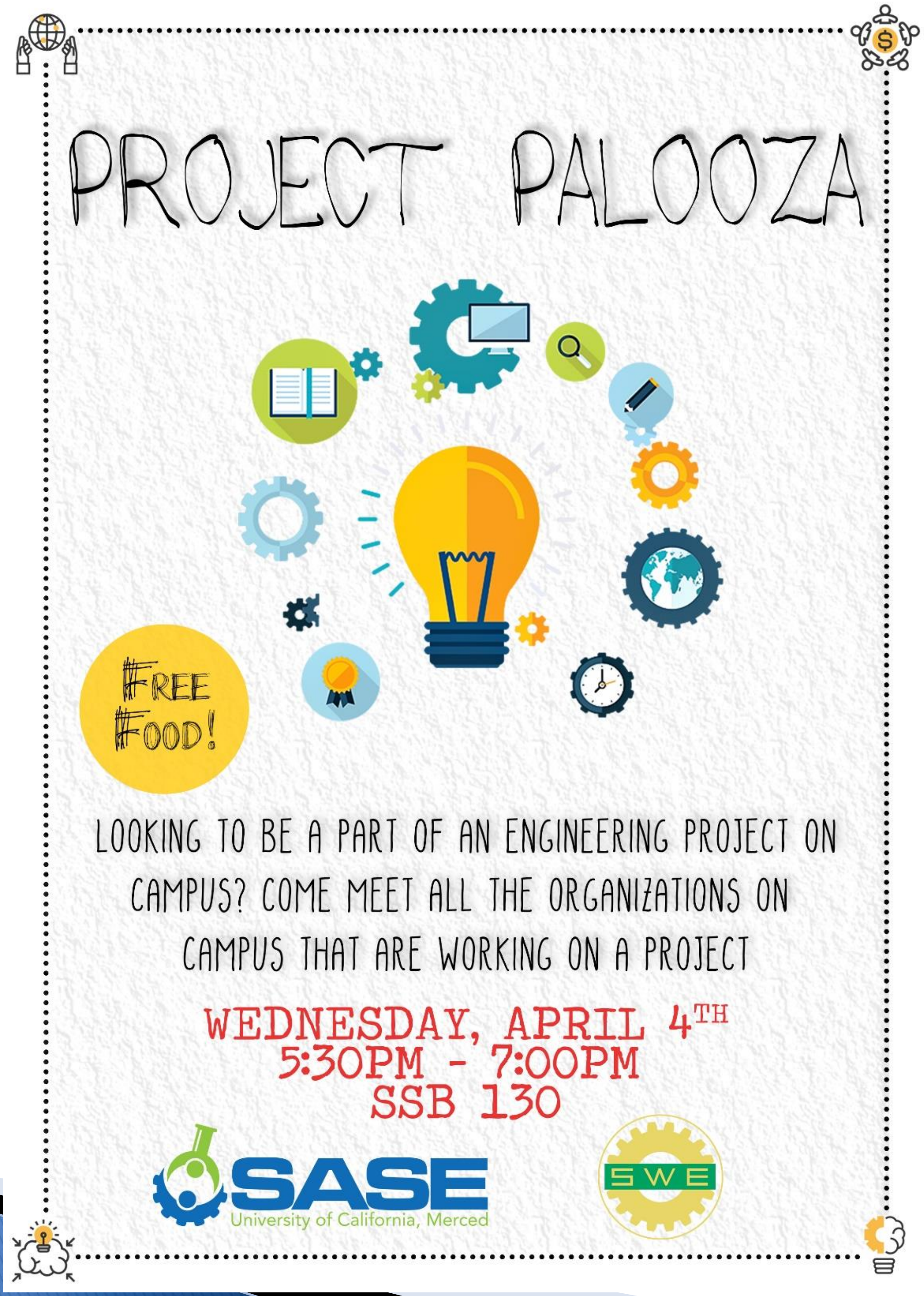
CSE 21
Intro to Computing II

Lecture 8 – Inheritance

Today

- Inheritance
- Lab
  - Lab 9 due this week (4/1 – 4/7)
  - Lab 10 assigned this week
    - Inheritance
    - Due in one week
    - **Required** to show work to a TA (or me) for full credit
  - Project 2 due next week on Friday, 4/13
    - **Required** to show work to a TA (or me) for full credit
- Reading Assignment
  - Sections 7.11 – 7.14 (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



PROJECT PALOOZA

FREE FOOD!

LOOKING TO BE A PART OF AN ENGINEERING PROJECT ON CAMPUS? COME MEET ALL THE ORGANIZATIONS ON CAMPUS THAT ARE WORKING ON A PROJECT

WEDNESDAY, APRIL 4TH
5:30PM - 7:00PM
SSB 130

SASE
University of California, Merced

SWE

Common Methods in a Class (review)

- Methods common to many classes
  - *Constructors* are called if you ask for a ***new*** object
    - Java provides a ***default*** constructor (with no arguments)
  - *Accessors*, or "get methods", or "getters" are used to read/retrieve the values of instance variables
    - Including predicate methods returning booleans
  - *Mutators*, or "set methods", or "setters" are used to set the values of instance variables
  - **toString** method creates a String representation of the contents of the object
    - **System.out.println(obj)** calls object's **toString**

Static vs Non-Static (review)

```
/* A Counter that remembers the number of
 * times it has been asked to increment itself,
 * and how many counters exist.
 */
public class Counter {
    // Member variable
    int myCount = 0;

    // Class variable
    public static int numCounters = 0;

    // Override the default constructor to keep
    // track of how many counters created
    public Counter() {
        numCounters++;
    }

    // Modify counter by incrementing itself.
    public void increment() {
        myCount++;
    }

    // Static method increments numCounters for
    // ALL OBJECTS.
    public static void incCounters(int amt) {
        numCounters = numCounters + amt;
    }

    // Return the current counter reading.
    public int getCount () {
        return myCount;
    }

    // Return a String representation of a Counter
    public String toString() {
        return ("" + myCount);
    }
}
```

Static variables and methods accessed using CLASS NAME

```
Counter.numCounters;
Counter.incCounters(5);
```

Accessors and Mutators (review)

```
public class Date {
    public int day, year;
    private int month; // Only accessible inside class

    // Constructor 1
    public Date(int day, int year) {
        this.day = day;
        this.year = year;
    }

    // Constructor 2 ensures valid month
    public Date(int month) {
        setMonth(month);
    }

    // Error checking (defensive programming)
    public void setMonth(int month) {
        if (month > 0 && month <= 12)
            this.month = month;
        else
            System.out.println("Invalid month");
    }

    // Allow access to month outside class
    public int getMonth() {
        return month;
    }
}
```

- The private access level for **month** means that ***only code belonging directly to the class*** may use that data member directly.
- All other code must access that data member through some class methods (***as long as the method itself is not private***).
    - Mutator - allows us to ***incorporate error checking*** with our data members
    - Accessor – retrieval method to use value from private member variable ***outside the class***

Question?

```
public class Date {
    public int day;
    public int month;
    public int year;

    public Date(int year) {
        day = month = 0;
        year = year; // instead of this.year = year;
    }
}

Date birthdate = new Date(2017);
System.out.println("Birth Year: " + birthdate.year); ???
```

Birth Year: 0

The "this" implicit parameter (review)

```
public void display( ) {
            System.out.print(month + "/");
            System.out.print(day + "/");
            System.out.println(year);
      }
```

is the same as …

```
public void display( ) {
            System.out.print(this.month + "/");
            System.out.print(this.day + "/");
            System.out.println(this.year);
}
```

Assume 2 objects:

```
    Date alice = new Date();
    Date bob = new Date();
```

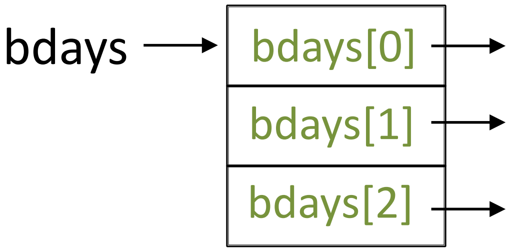
Method calls by alice and bob know whose variables to use

- If we want to work with the data in the alice object variable, we specify that:
  alice.display();
- If we want to work with the data in the bob object variable, we specify that also:
  bob.display();

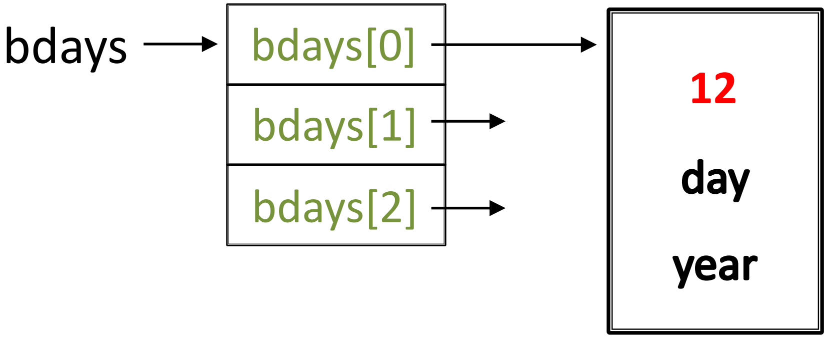
- Compiler converts **objectReference.method(…);** to **method(objectReference, …);**
- Implicitly-passed object reference is accessible via **this**

Array of Objects (review)

- Date alice = new Date();
  - Creates an object pointed to by variable alice
- Date[] bdays = new Date[3];
  - Creates a set of 3 Date pointers
  - Does not have objects yet
  - Not valid to use bdays[0].setMonth(12) yet



- bdays[0] = new Date();
  - Now we can access
  - bdays[0].setMonth(12);



- Need to instantiate two things for arrays (new)
  - Pointers using Square brackets
  - Objects using parenthesis

Object as Method Parameters (review)

- public void intro(Scanner input)
  - Takes in a Scanner object named input
- Date alice = new Date();
  - Creates an object pointed to by variable/pointer alice
- Date twin = alice;
  - Both **twin** and **alice** point to the **SAME** object
- Date twin = new Date(alice);
  - Assume we have a constructor as shown below:

```
public Date(Date original)  {
     this.setDay(original.getDay()); // this.day = original.getDay();
     this.setMonth(original.getMonth()); // this month= original.getMonth();
     this.setYear(original.getYear()); // this year = original.getYear();
}
```

  - Creates a copy of the original object
    - Get the original value and put it in the new object
    - **twin** and **alice** are different objects with same values for member variables

Inheritance : Motivation

- 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

```
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 {


}
```

```
ModNCounter c = new ModNCounter();
c.increment(); // THIS IS CORRECT
```

Counter Class Example

```
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;

}
```

Additional instance variable

Counter Class Example

```
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;
        }



}
```

Needs its own constructor

Counter Class Example

```
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;
        }
}
```

Overriding (overloading) a method

Counter Class Example

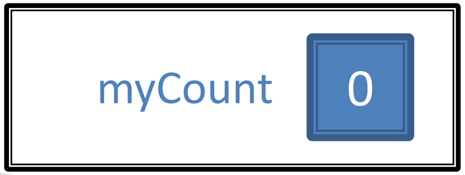
```
public class Counter {
        private int myCount;
        public Counter() {
                myCount = 0;
        }
        public void increment(){
                myCount++;
        }
        public void reset() {
                myCount = 0;
        }
        public int value() {
          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

New method

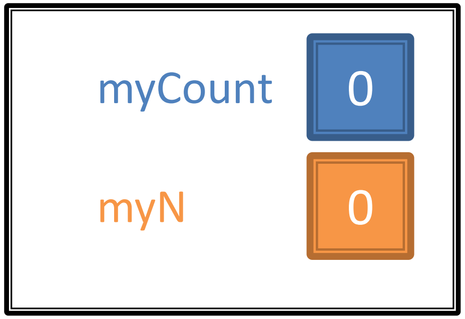
Counter Class Example

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



myCount 0

```
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;
    }
}
```



myCount 0

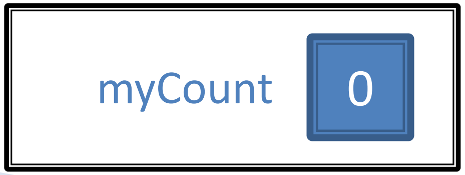
myN 0

Protected Access Specifier

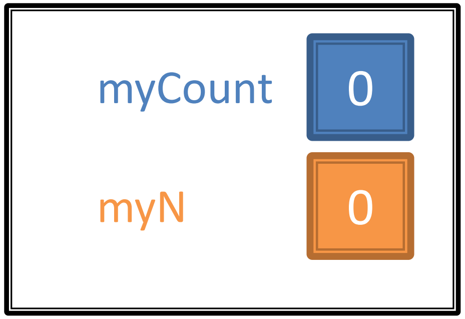
- As written, *ModNCounter* will not compile!
- The *myCount* variable is private (only accessible in the *Counter* class)
- We can fix this by making it **protected**:
  - Only classes that "extend" *Counter* can access its protected variables/methods
- Three different Access types:
  - **public**: 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

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



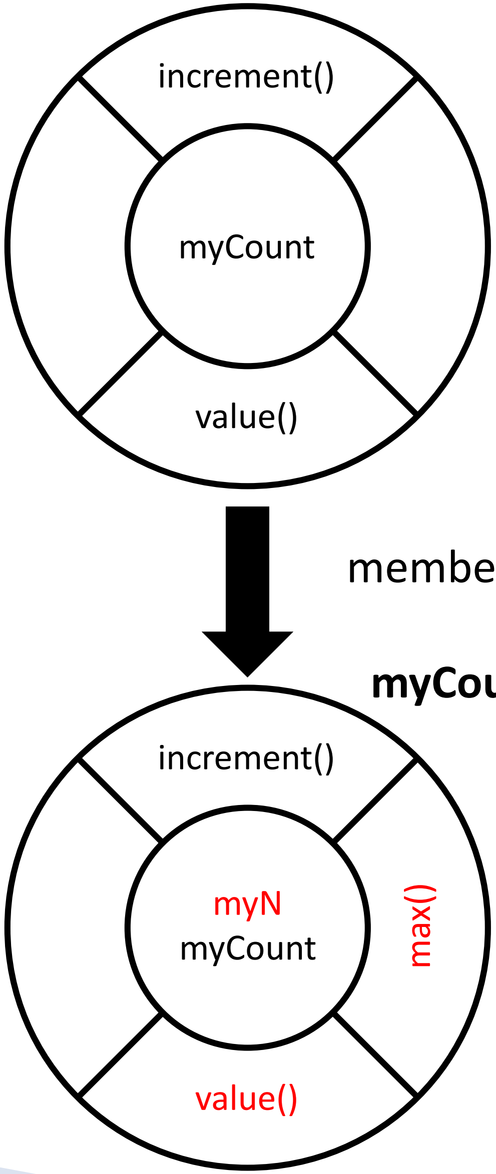
```
public class ModNCounter extends Counter {

        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;
        }
}
```



Inheritance

Superclass

class Counter



Subclass inherits
members from superclass (public
or protected)
myCount, increment(), value()

Subclass

class ModNCounter

Type Casting in Inheritance

- 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

- It is OK to pass an object of a class, say *EgClass,* as argument to a method that expects an object of *EgClass*'s superclass 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

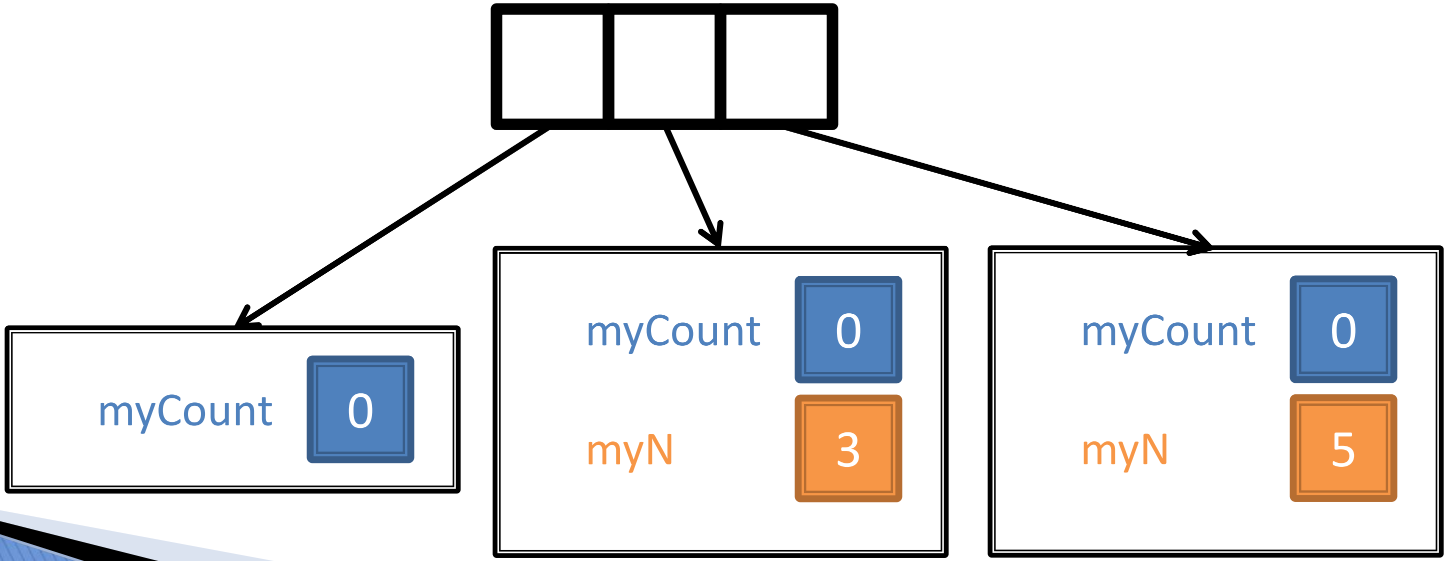
```
mc.max();                        // OK, because mc is a ModNCounter
((ModNCounter)c).max();  // ERROR: because c may
                                 // or may not be ModNCounter
```

Arrays of Objects from Class/Superclass

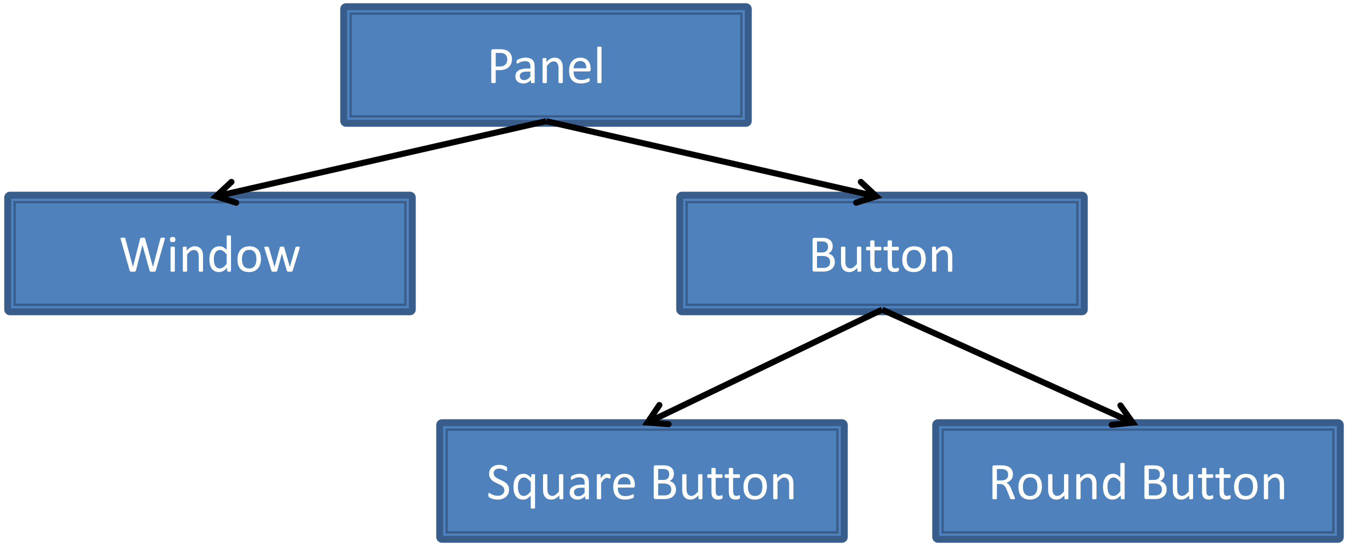
- Build an array of 3 Counters

```
Counter[] a = new Counter [3];
a[0] = new Counter();
a[1] = new ModNCounter(3);
a[2] = new ModNCounter(5);
```

Remember: need to use multiple new to create objects inside an array!



Inheritance Can be Multiple Levels

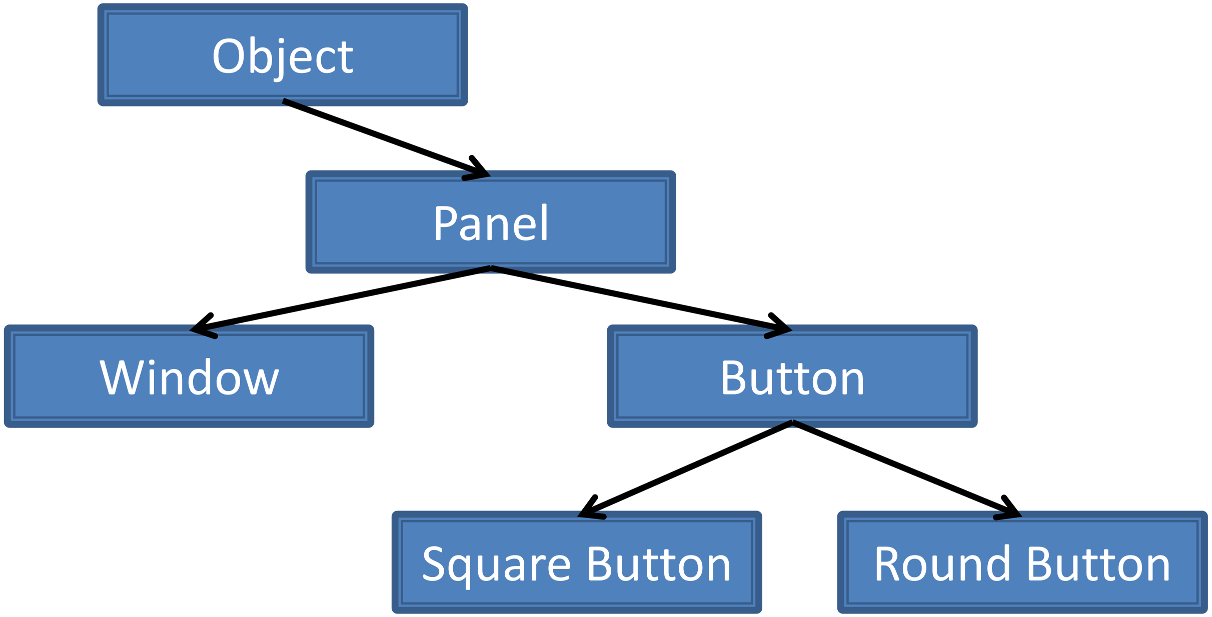


- Complex class hierarchies can be created

```
Panel[] p = new Panel [3];
p[0] = new Panel();
p[1] = new RoundButton();
p[2] = new Window();
```

Storing the graphical components of a program in an array

Object is at the Top in Java



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