Method Practice

GEEN163

"The question of whether a computer can think is no more interesting than the question of whether a submarine can swim."

Edsger Dijkstra

Blocks

- In Java, a block of code is any set of Java statements inside **{**curly brackets**}**
- The brackets of a class definition defines a block containing the whole class
- The brackets after a method header define a block of that method

Variable Scope

- The *"Scope"* of a variable refers to where you can use a variable
- Where you declare a variable determines where it can be used
- The variable names used in a method are a completely separate set of names from the names used in any other method, including the main method

Variable Range

- Variables defined in a block can only be used in that block following the variable declaration

```
{
        // you cannot use the variable moth here
        double moth = 72.5;
        // you may use the variable moth here
        // this is the scope of moth
}
// The variable moth is not allowed here
```

Out of Scope

```
int cat;
if (whatever == 0) {
    int dog = 1;
} else {
    int dog = 2;
}
cat = dog * 3; // Error, dog is out of scope
```

This works

```
int cat, dog;
if (whatever == 0) {
    dog = 1;
} else {
    dog = 2;
}
cat = dog * 3; // Only one dog variable
```

Method Parameter Scope

- The parameters defined in a method header can be used throughout that method
- Method parameter variables cannot be used in another method

```
void aMethod( int dog, double cat ) {
You can use dog and cat throughout this method
}
```

Declarations First

- Java allows you to put variable declarations anywhere in a program
- It is usually advantageous to declare a variable at the beginning of a block
- If you declare a variable at the beginning of a block, you can use it throughout the block
- Some older programming languages required you to put all variable declarations at the beginning of a block

for Loop Scope

- In a for loop, you can declare the variable that is used as the loop counter
- The variable only has scope in the loop

```
for (int cow = 0; cow < 28; cow++) {
        // loop body
        // the loop counter, cow, can be used here
}
// You cannot use cow after the loop
```

Local Variables

```
public static void main( ) {
    double    cat = 5, bird = 47;
    cat = myfunc( bird );
    System.out.print(cat);
}
```

Scope of cat and bird

```
double myfunc(double cow) {
    double bull;
    bull = cow * 2.0;
    return bull;
}
```

Scope of cow and bull

Multiple Variable with the Same Name

- In different parts of a program, you can use the same variable name, but it will mean a different variable
- This can be confusing and should be avoided

Local Variables *(Tricky)*

```
public static void main( ) {
    double   cat = 5, bird = 47;
    cat = myfunc( bird );
    System.out.print(cat);
}
```

Scope of cat and bird

```
double myfunc(double cow) {
    double bird;
    bird = cow * 2.0;
    return bird;
}
```

Scope of cow and bird (different bird)

Use Unique Names

- Use a new and different name for every variable in your program
- There are 228,698,418,577,408 possible different Java variable names of 8 characters
- Names can be as long as you want
- Use variable names that mean something in the context of the program *(not dog, cat, etc.)*

What will be displayed?

```
int mouse = 3, rat = 5;
rat = myMethod( mouse );
System.out.println( mouse+"   "+rat);
…      …      …      …      …
int myMethod( int rat ) {
        int mouse = 7;
        return mouse * rat;
}
```

A.3
B.5
C.7
D.15
E.21

Variables

- A method can use three different types of variables
  - local variables defined in the method
  - parameter variables
  - object instance variables
- Object instance variables can be used in any of the methods of the class

Overlapping Names

- Java allows you to declare a variable with the same name in an inner block
- The declaration closest to the variables use is the one that applies

- This can be ***very*** confusing and should be avoided

Name Collision

```
public class Collide {
  int rat = 3;
  public int square( int rat ) {
     return rat * rat;
  }
}
```

```
Collide thing = new Collide();
int turtle = thing.square( 2 );
```

- turtle is set to 4

Variable Priority

- A method cannot have a local variable the same name as a parameter variable
- If a class instance variable has the same name as a local variable or parameter, the method will always use the local variable or parameter

More Name Collisions

```
public class Collide {
  int rat = 3;
  public int square( int mouse) {
     int rat = mouse * mouse;
     return rat;
  }
}
```

- The class instance variable rat is a different memory location from the local variable rat

this

- The Java keyword "**`this`**" means this object
- You can use **`this`** to access anything of the object, but not the class
- If you have an instance variable named **`xyz`**, then you can access the instance variable as **`this.xyz`**
- Object instance variables can always be accessed by putting the name of the object followed by a period and the instance variable name

Name Collision

```
public class Collide {
  int rat = 3;
  public int square( int rat ) {
     return this.rat * rat;
  }
}
Collide thing = new Collide();
int turtle = thing.square( 2 );
```

- turtle gets the value 6

Static Variables

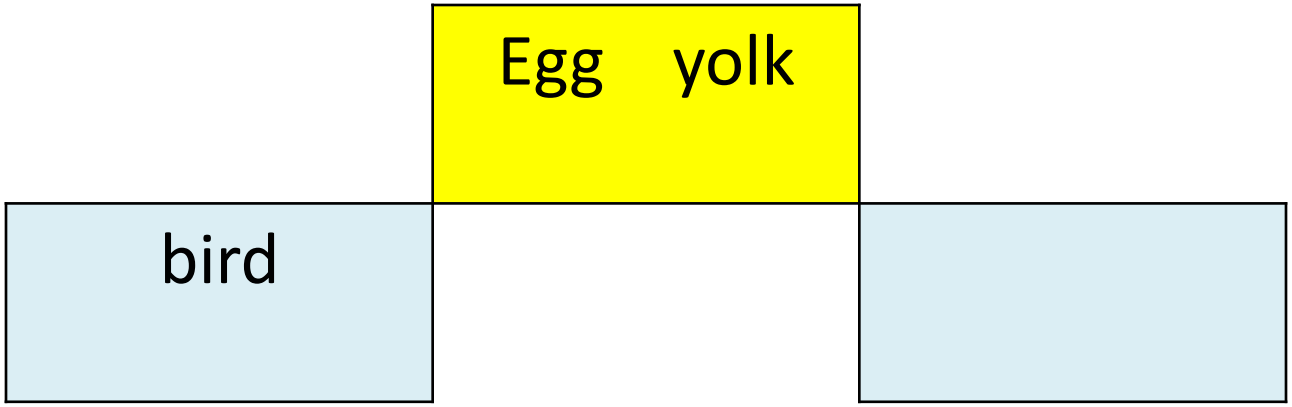
- Class variables can be specified as static

```
public class electric {
   static  int cat;   // static class variable
   int dog;           // non- static class variable
}
```

- There is only one copy of a static variable shared by all objects of that class
- There is a separate copy of non-static variables for each object

Only One Copy of Static Variables

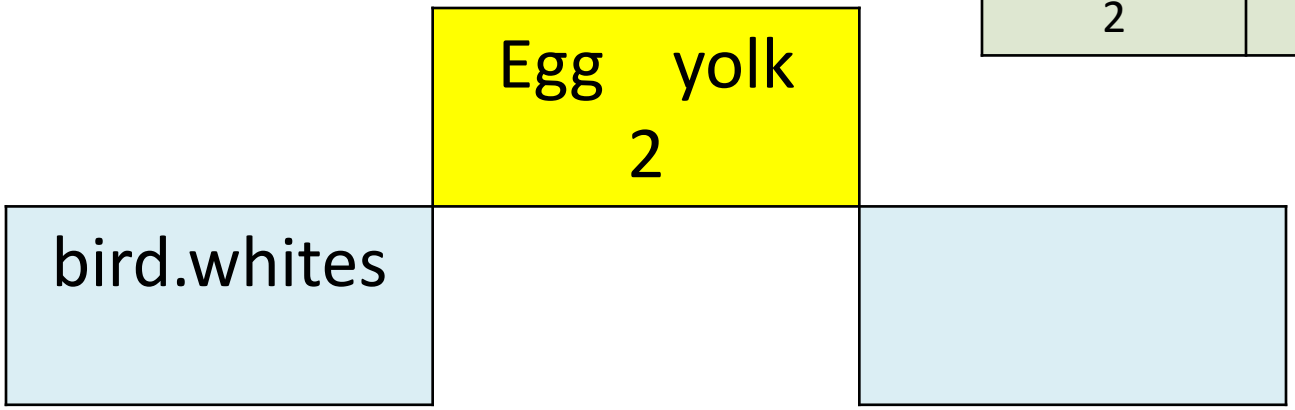
```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```



Only One Copy of Static Variables

```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```

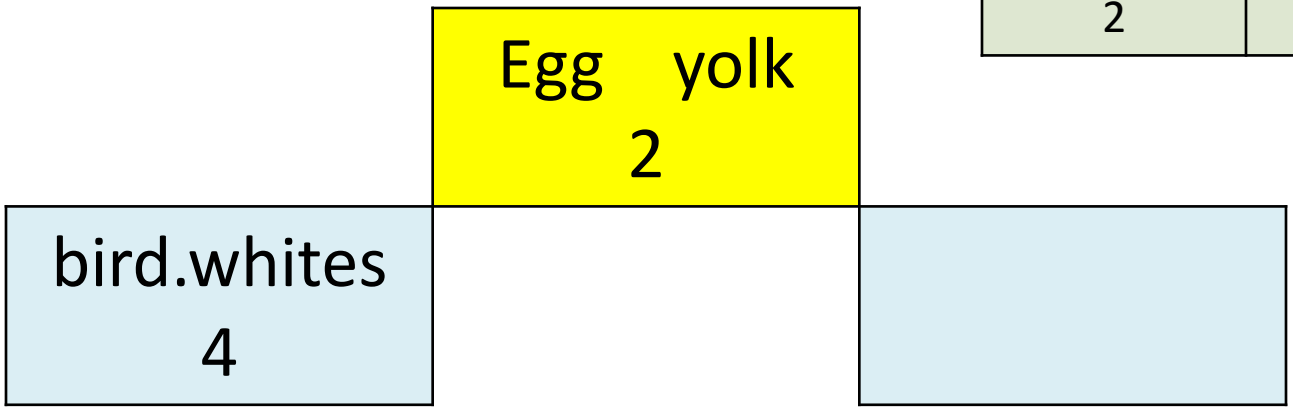
frog	toad
2	4



Only One Copy of Static Variables

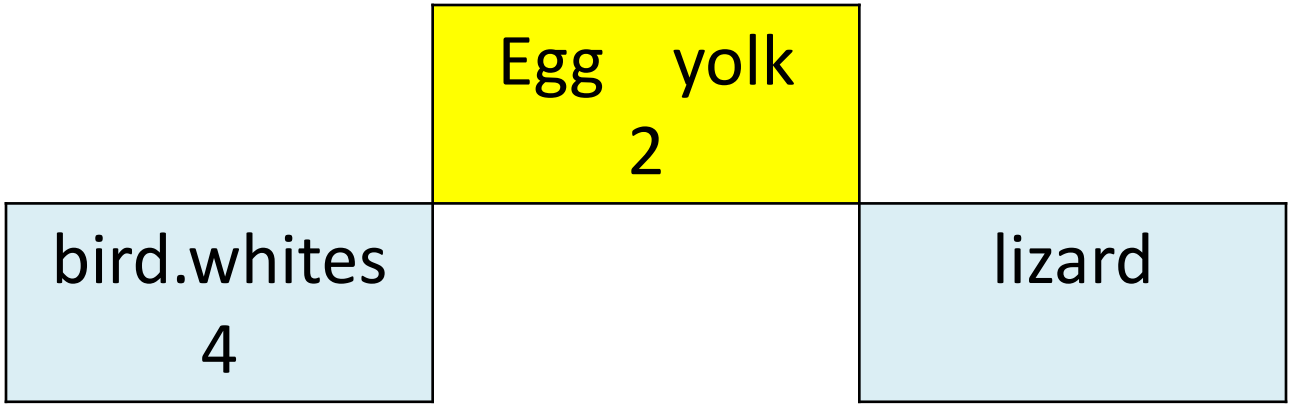
```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```

frog	toad
2	4



Only One Copy of Static Variables

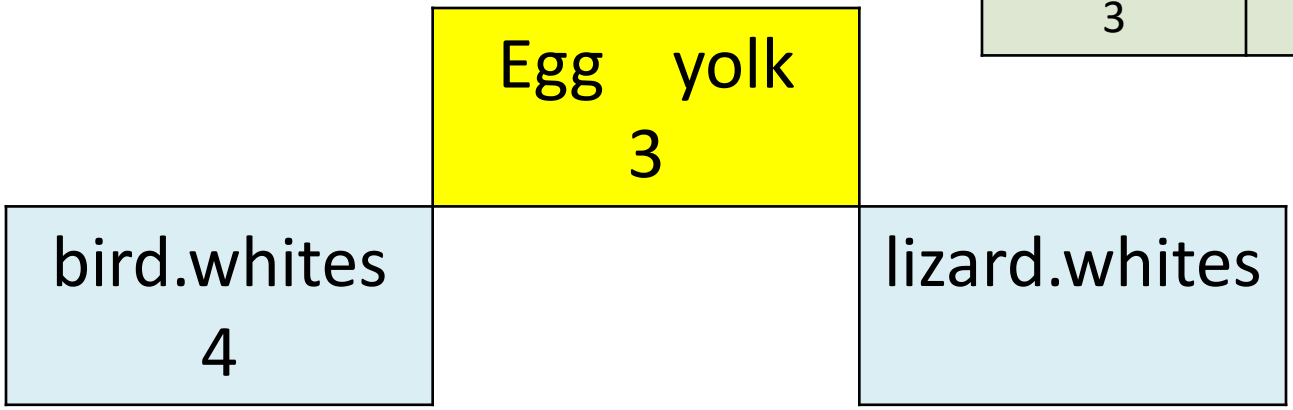
```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```



Only One Copy of Static Variables

```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```

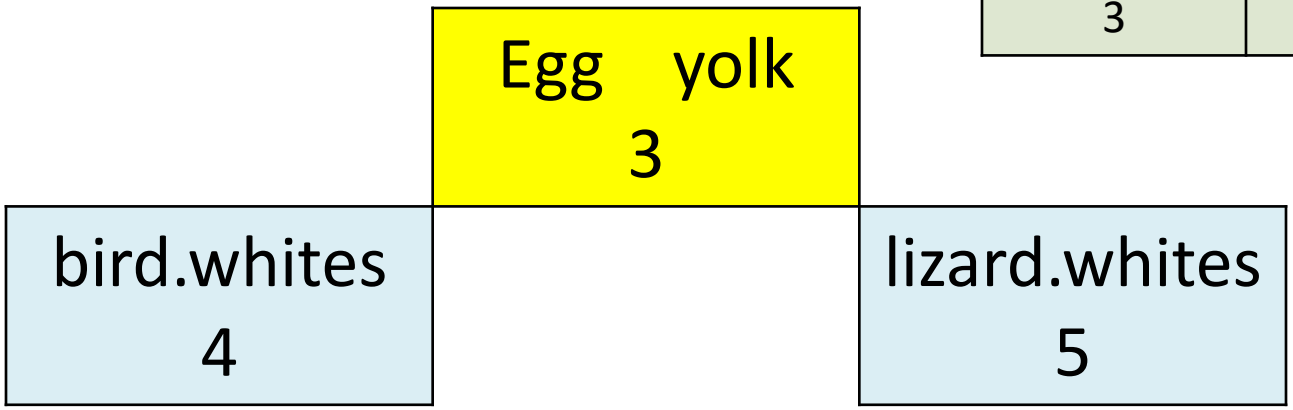
frog	toad
3	5



Only One Copy of Static Variables

```
public class Egg {
    static int yolk;
    int whites;
    public Egg(int frog, int toad) {
        yolk = frog;
        whites = toad;
    }
}
// --------- In another class -----------
Egg bird = new Egg( 2, 4 );
Egg lizard = new Egg( 3, 5 );
```

frog	toad
3	5



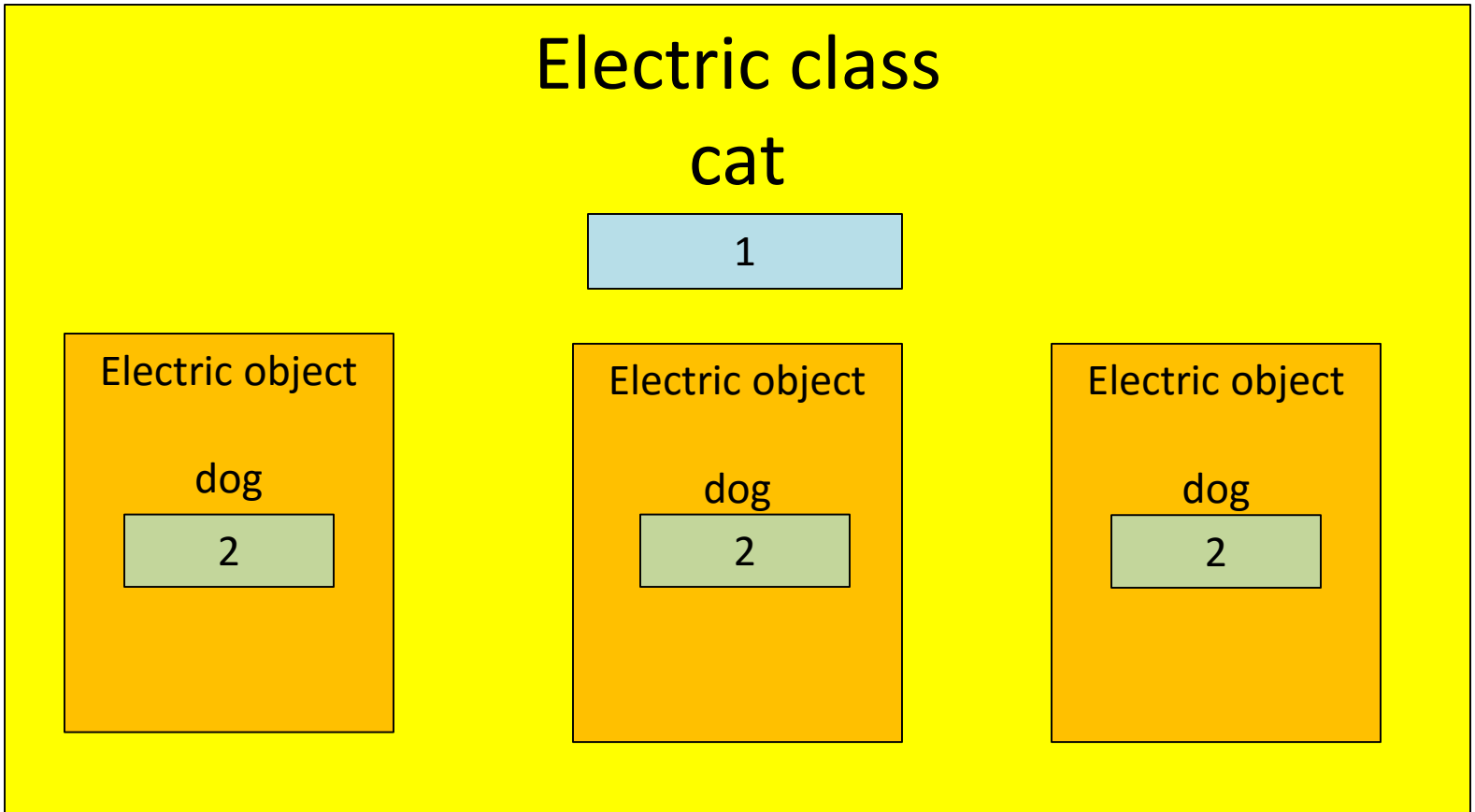
Multiple Objects

```
public class Electric {
   static  int cat = 1;
   int dog = 2;
}

Electric possum = new Electric();
Electric rat    = new Electric();
Electric mouse  = new Electric();
```

- There are three Electric objects
- Each object has its own copy of dog
- There is only one cat variable

Static and Instance Variables

- There is only one cat variable, but there is a dog for every object



Using Instance Variables

```
public class Electric {
   static  int cat = 1;
   int dog = 2;
   public void cat2() {
       cat = cat + 2;
   }
   public void dog3() {
     dog = dog + 3;
   }
   public void prtCat() {
     System.out.println(cat);
   }
   public void prtDog() {
     System.out.println(dog);
   }
}
```

```
Electric volt = new Electric();
Electric watt = new Electric();
volt.cat2();
watt.dog3();          displays
volt.prtCat();            3
volt.prtDog();            2
watt.prtCat();            3
watt.prtDog();            5
```

```
public class Stest {
   static int svar = 1;
   int dvar = 1;
   void incBoth() {
      svar++;
      dvar++;
      System.out.println(svar+" "+dvar);
   }
   static public void main(String[] x) {
      Stest cat = new Stest();
      Stest dog = new Stest();
      cat.incBoth();
      dog.incBoth();
   }
}
```

What is displayed?

A. 1 1
   1 1

B. 2 2
   2 2

C. 2 2
   3 3

D. 2 2
   3 2

Static and Dynamic

- A non-static variable can only be accessed by non-static methods
- Non-static variables belong to the object

- A static variable can be accessed by any method
- Static variables belong to the class

Static and Dynamic Example

```
public class Program {
  static int  statData = 3;
  int         dyData = 1;
  void inc() {                    // non-static method
        dyData++;
        statData++;
  }
  public static void main(String[] x) {
        Program thing = new Program();
        thing.inc();
        statData += 2;
        dyData   += 5;          // not Allowed
        thing.dyData += 7; // permitted
  }
```

Static or Not

- If all objects are going to share the variable, make it static
- If each object should have its own value, do not make it static
- If a method uses instance variables, it cannot be static
- If a method does not use any instance variables, it can be static

final variables

- A variable declared as final cannot be changed
- If you want to define a constant value, it should be declared final

```
final int MAXTHINGS = 10;
```

- Variables that are final must be initialized with a value
- It is good programming practice to put constants at the beginning of the program instead of using a number throughout the code

`final` is final

- final variables must be initialized when they are declared
- You will get a compiler error if you try to change a final variable

```
final double dragon = 47.0;
dragon = -3.5;              // this will get an error
```

static Constants

- Typically a program constant would apply to all objects in a class
- Therefore it is useful to make constants static so there is only one copy

```
public class Both {
        private static final int MAXSIZE = 12;
```

- Variable attributes can be in any order

```
        final static private int MAXSIZE = 12;
```

What is displayed?

```
public class Teach {
   public int cat = 2;
   public Teach(int dog) {
      cat = dog;
   }
   public void prtCat() {
      System.out.print(cat);
   }
}
```

```
public class MyProg {
   public static void main(…) {
      Teach mine = new Teach(5);
      Teach yours = new Teach(3);
      mine.prtCat();
      yours.prtCat();
   }
}
```

A. 2 2

B. 3 3

C. 5 3

D. 5 5

Review of Variable Attributes

- public – variable can be used anywhere
- private – variable can only be used in the same class
- static – variable belongs to the class not objects. Only one copy
- final – variable cannot be changed

Attributes on Instance Variables

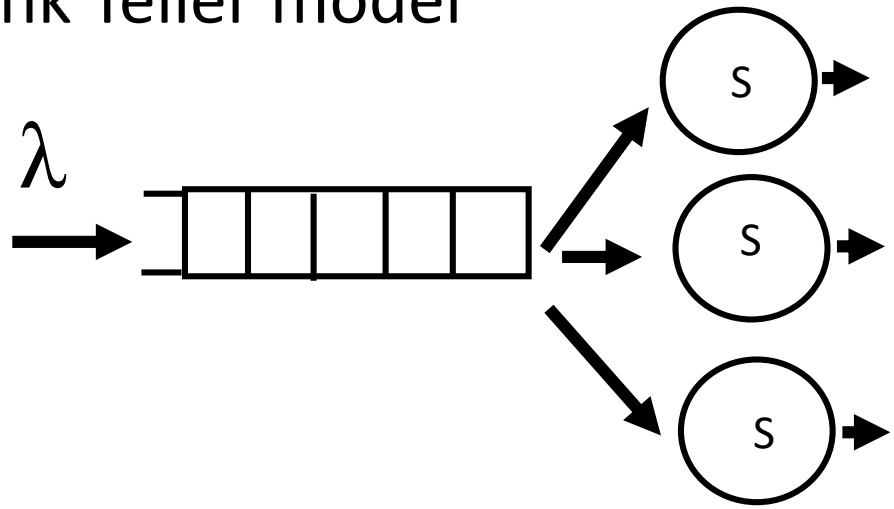
```
public class Modified {
        public static int dog;
        private final double cat;

        public void doIt( int bull) {
                public int cow;    // not allowed
                cat = (double)bull;
        }
}
```

Queuing Theory

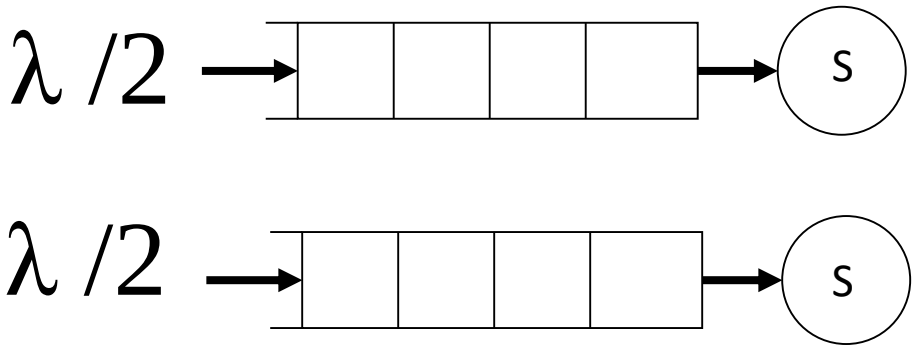
- Queuing theory is the mathematics of waiting lines
- It is extremely useful in predicting and evaluating system performance

Single or Multiple Queues

M/M/N Bank Teller model



M/M/1 Grocery Store model



Queuing Theory

- The probability that all n servers are busy in an M/M/N system (bank teller model) is C

$$C = \frac{1 - K}{1 - \frac{\lambda s K}{N}}$$

- where

$$K = \frac{\sum_{i=0}^{N-1} \frac{(\lambda s)^i}{i!}}{\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}}$$

Example Method

- Write a method that returns the value C given the three input parameters **lambda**(λ), **s** and **n**

- The method will have the header

```
double probAllBusy( double lambda, double s, int n)
```

Solution Outline

- In the equation for K, the denominator is a sum that is the same as the numerator except that it is one term longer
- If we have a method to calculate the sum, the rest of the method is simple

probAllBusy method

```
double probAllBusy( double lambda,
                                    double s, int n) {
        double k = kSum(lambda, s, n-1) /
                        kSum(lambda, s, n);
        return (1.0 - k) / (1.0 – lambda*s*k/n);
}
```

$$C = \frac{1 - K}{1 - \frac{\lambda s K}{N}}$$

$$K = \frac{\sum_{i=0}^{N-1} \frac{(\lambda s)^i}{i!}}{\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}}$$

kSum method

- The kSum method calculates this sum given the parameters lambda, s and n

$$\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}$$

- We will need a method to calculate factorials

factorial method

- The factorial function in mathematics, n!, is

$$1 * 2 * 3 * 4 * \ldots * n$$

- For the special case of zero, 0! = 1

The header for factorial should be

A. int factorial(int n)

B. void factorial(int n, int fac)

C. static factorial(int n)

D. none of the above

Write the factorial method

- With your team, write a method with one integer parameter, n, that returns n!

- Don’t forget the special case for zero
  - You might not have to do anything special

Possible Solution

```
int factorial( int n ) {
        int fac = 1;
        for (int i =1; i <= n; i++) {
                fac = fac * i;
        }
        return fac;
}
```

Write the kSum method

- Write a method to calculate this sum given the parameters lambda, s and n

$$\sum_{i=0}^{N} \frac{(\lambda s)^i}{i!}$$

double kSum(double lambda, double s, int n)

Possible Solution

```
double kSum( double lambda, double s, int n ) {
   double sum = 0.0;
   for (int i = 0; i <= n; i++) {
      sum += Math.pow( lambda*s, (double)i ) /
                                          factorial( i );
   }
   return sum;
}
```

It looks Greek to me

```
double kSum( double λ, double s, int n ) {
   double sum = 0.0;
   for (int i = 0; i <= n; i++) {
      sum += Math.pow( λ*s, (double)i ) /
                                        factorial( i );
   }
   return sum;
}
```

Small Steps

- The value of using methods is that you can break a larger problem into small steps
- The probAllBusy method called the kSum method twice
- kSum called factorial
- Note that we wrote the program using methods we had not yet written, but knew we could write

Recursion

- (N-1)! is 1 * 2 * 3 * 4 * … * N-1

- N! is 1 * 2 * 3 * 4 * … * N-1 * N

or (N-1)! * N

- If you have a factorial method that computes (N-1)!, you can calculate N! by multiplying the result of the method by N
- You need to be mindful that N-1 might drop to zero

Recursive Solution

```
int factorial( int n ) {
        if (n <= 1) return 1;
        return n * factorial( n-1 );
}
```

Methods and Object Data

- Methods belong to a class or object
- Methods can use all instance variables, parameters and local variables
- In a typical program, you keep the data in instance variables which is accessed and modified by methods

Encapsulation

```
public class Temp {
        private double celsius;
        public void setC(double therm) {
                celsius = therm;
        }
        public void setF(double therm) {
                celsius = (therm – 32.0) * 5.0/9.0;
        }
        public double getC() {
                return celsius;
        }
        public double getF() {
                return celsius * 9.0/5.0 + 32.0;
        }
}
```

Using the Temp class

```
Temp weather = new Temp();
weather.setC( 20.0 );
double f = weather.getF();          // f = 68.0
double c = weather.getC();          // c = 20.0
```

No Food or Drinks in the Lab