More Method Practice

GEEN163

"People think that computer science is the art of geniuses, but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

Donald Knuth

TuringsCraft

- Read chapter 8 of the textbook on arrays
- Answer the questions in section 8 of the TuringsCraft tutorial system
  - 4 points for each correct answer
  - maximum of 100 points
- Due by midnight on **Tuesday, April 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

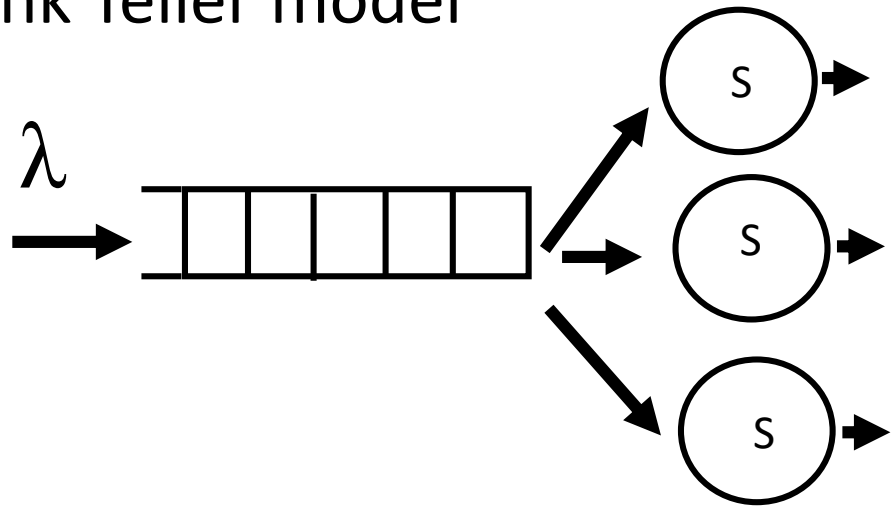
```
public class Modified {
        public static int dog;
        private final double cat;

        public void doIt( int bull) {
                public int cow;    // not allowed
                final int GERBIL = 7;
                dog = (double)bull + GERBIL;
        }
}
```

Queuing Theory

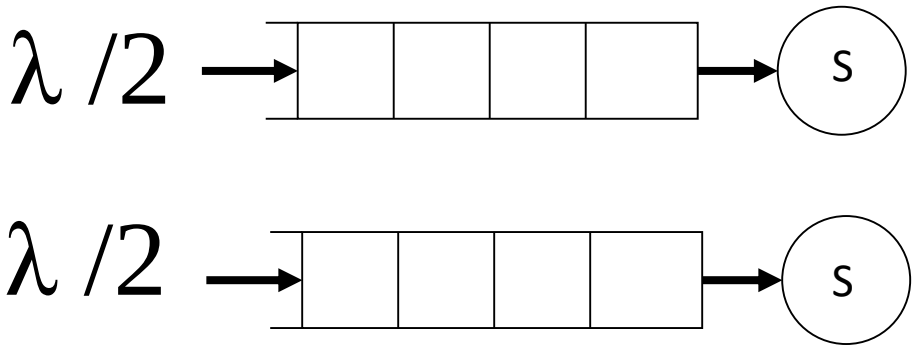
- Queuing theory is the mathematics of waiting lines
- It is extremely useful in predicting and evaluating system performance

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;
}
```

You should make the methods

A. KSum and factorial non-static
B. KSum and factorial static
C. KSum static and factorial non-static
D. KSum non-static and factorial static

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

- A class keeps the data about a specific concept
- The methods of a class define how we can use the data
- If the instance variables are private, all access to the class is through the methods
- The programmer who uses the class does not have to know what is inside
- This is the concept of **encapsulation**

Encapsulation

```
public class Temperature {
        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 Temperature class

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

Methods calling Methods

```
int dog = 5, cat;                    // dog = 5
cat = methA( dog );
```

```
int methA( int cow ) {
   int bull =  2 * cow;
   steer = methB( bull )
   return steer
}
```

```
int methB( int lizard ) {
   return lizard + 3;
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {                    // cow = 5
   int bull =  2 * cow;
   steer = methB( bull )
   return steer
}
```

```
int methB( int lizard ) {
   return lizard + 3;
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {              // cow = 5
   int bull =  2 * cow;             // bull = 10
   steer = methB( bull )
   return steer
}
```

```
int methB( int lizard ) {
   return lizard + 3;
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {          // cow = 5
   int bull =  2 * cow;         // bull = 10
   steer = methB( bull )
   return steer
}
```

```
int methB( int lizard ) {       // lizard = 10
   return lizard + 3;
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {          // cow = 5
   int bull =  2 * cow;         // bull = 10
   steer = methB( bull )
   return steer
}
```

```
int methB( int lizard ) {       // lizard = 10
   return lizard + 3;           // returns 13
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {          // cow = 5
  int bull =  2 * cow;          // bull = 10
  steer = methB( bull )         // steer 13
  return steer
}
```

```
int methB( int lizard ) {       // lizard = 10
  return lizard + 3;            // returns 13
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );
```

```
int methA( int cow ) {          // cow = 5
  int bull =  2 * cow;          // bull = 10
  steer = methB( bull )         // steer 13
  return steer                  // returns 13
}
```

```
int methB( int lizard ) {       // lizard = 10
  return lizard + 3;            // returns 13
}
```

Methods calling Methods

```
int dog = 5, cat;
cat = methA( dog );                    // cat = 13
```

```
int methA( int cow ) {                 // cow = 5
   int bull =  2 * cow;                // bull = 10
   steer = methB( bull )               // steer 13
   return steer                        // returns 13
}
```

```
int methB( int lizard ) {              // lizard = 10
   return lizard + 3;                  // returns 13
}
```

What is displayed?

```
static int methA(int dog) {
   int cat = methB( dog + 2 );
   return cat;
}
static int methB(int goat) {
   return goat / 2;
}
public static void main(String[] x) {
int fish = methA( 4 );
System.out.println( fish );
}
```

A. 2
B. 3
C. 4
D. 6
E. none of the above

Method Example

- We want to create a method that will return a string with the full name properly capitalized

Example: if first = "fred" and last = "smiTH", the method should return "Fred Smith"

String makeName(String first, String last)

Useful Method

- We need to adjust the case of both the first and last name
- The below method capitalizes the first letter of a string and lowercases the rest

```
static String  titleCap( String name ) {
   String first = name.substring(0,1);
   String rest = name.substring( 1 );
   return first.toUpperCase() + rest.toLowerCase();
}
```

Write with your team

- Complete this method return a string with the full name properly capitalized
- *use* String titleCap(String name)

Example: if first = "fred" and last = "smiTH" the method should return "Fred Smith"

```
static String makeName(String first, String last) {
```

Possible Solution

```
static String makeName(String first, String last) {
   return titleCap( first ) +" "+ titleCap( last );
}
```

Another Possible Solution

```
static String makeName(String first, String last) {
   String fn = titleCap( first );
   String ln = titleCap( last );
   String together = fn +" "+ ln;
   return together;
}
```

Avoid Repeating Yourself

- When you have to perform the same action of different data, it is usually beneficial to write a method to do the action
- In the previous example, we had to modify the case of both the first and last name
- Writing a method once is usually easier than writing the code twice the main

Small Simple Steps

- Breaking a problem into small steps makes it easier to solve
- Once you have written a method and gotten it to work, you can use it over and over again (and it should continue to work)

No Food or Drinks in the Lab

TuringsCraft

- Read chapter 8 of the textbook on arrays
- Answer the questions in section 8 of the TuringsCraft tutorial system
  - 4 points for each correct answer
  - maximum of 100 points
- Due by midnight on **Tuesday, April 5**