CS 121 - Intro to Programming: Java - Lecture 10

Announcements

Ch 5 OWL assignment due 10/13

Online midterm coming end of next week - most likely released Thursday evening.

Next week: Tuesday is Monday schedule (so discussions are on Tuesday, no Tuesday lecture)
Principal theme again: **methods**

Methods organize (sub)jobs at the statement level

Fundamental mechanism for combining elementary operations to make reusable, more complex operations

You can build an entire “world” with methods: very complex chores rely on complex chores, which rely on elementary chores, which rely on primitives.. etc.
public class SimpleCoins {

    static final int HEADS = 1;
    static final int TAILS  = 0;

    public int flip(){
        if (Math.random() < 0.5)
            return TAILS;
        else
            return HEADS;
    }

    public int multiFlip(int flips) {
        int total = 0;
        for(int j = 0; j < flips; j++) {
            total += flip();
        }
        return total;
    }
}
public int multiFlip(int flips) {
    int total = 0;
    for (int j = 0; j < flips; j++) {
        total += flip();
    }
    return total;
}
Where does method input come from?

- parameters
- object attributes
- more global sources - Math.PI
What can methods do?

- Produce output - via return statements
- Change object state - alter instance variables
  
  ```java
  myKid.anotherMonth();
  ```
- Write stuff on the wall (ahem: the screen)
  
  ```java
  System.out.println(“Hi Julie”);
  ```
The Hand Off principle

Non-void methods “hand off” a value, either to a variable:

```java
int kidsAge = myKid.getAge();
```

Or to another method:

```java
System.out.println(myKid.getAge());
```

With void methods - NO hand off (though they may write something on the screen...)

Caesar Cipher

Used by Caesar himself to communicate with his generals.

A simple linear shift cipher - nowadays - useless.

So if the shift amount is 2

ZOO → BQQ

Shift of 5? 26? 31?

5: ZOO → ETT

26: Z00 → ZOO

31: ZOO → ETT (26 goes back to ZOO, then 5 more)
import java.util.*;

public class CipherTest{
    public static void main(String[] args){
        Scanner sca = new Scanner(System.in);
        System.out.println("enter a phrase");
        String s = sca.nextLine();
        System.out.println("enter a shift amount");
        int k = sca.nextInt();
        Cipher c = new Cipher(k);
        System.out.println(s);
        System.out.println(c.encrypt(s));
    }
}

public class Cipher { // displayed here over 2 slides
    private int shift;

    public Cipher(int s) {
        shift = s;
    }

    public String encrypt(String plain) {
        plain = plain.toUpperCase();
        String result = "";
        for (int j = 0; j < plain.length(); j++) {
            result = result + letterShift(plain.charAt(j));
        }
        return result;
    }
}
private char letterShift(char c) {
    // expects c to be either a non-letter or upper case
    if (Character.isLetter(c)) {
        int cPos = (c - 'A'); // a position from 0 to 25
        cPos = (cPos + shift) % 26; // deal with overflow
        return ((char) ('A' + cPos)); // remake as caps
    }
    else return c;
}
}
Method `divCount`: you give it a (positive) integer, it returns the number of integers that divide the number evenly.

Example: you give it 10, it should return 3 (since 1, 2, 5 proper divisors of 10)

Write `divCount`

1) Header line
2) Return statement
3) Body
Method `divCount`: you give it a (positive) integer -say k, it returns number of integers that (properly) divide k evenly

Example: you give it 10, it should return 3 (since 1, 2, 5 proper divisors of 10)

Example: 9 -> 2 (1 and 3)

Example: 12 -> 5 (1, 2, 3, 4, 6)

Example: 7 -> 1 (just 1)
public int divCount(int k){  // k > 0
}

public int divCount(int k) {  // k > 0
    int ct = 0;
    return ct;
}
public int divCount(int k){  
    int ct = 0;
    for(int j = 1; j < k; j++)
        if (k % j == 0) ct++;
    return ct;
}

Or

for(int j = 1; j <= k/2; j++)
When she came into the room...

[Later, in the second part of the sentence who the pronoun “she” refers to will be indentified]
When she came into the room, Jane smiled broadly at Filipe.
When she came into the room, Jane smiled broadly at Filipe.

The point: in English (and in most other languages) a pronoun can be used as a placeholder to defer identification of something.

Java has a corresponding pronoun(!)
The java keyword “this”
Suppose the Infant class has another method, called `babyName`

It puts the word “Baby” in front of an Infant’s name

```
Infant myKid = new Infant("Rick",4);
System.out.println(myKid.babyName());
pints--> Baby Rick
```

```
Infant myKid = new Infant("Jo",4);
System.out.println(myKid.babyName());
pints--> Baby Jo
```
In the expression

```
myKid.babyName();
```

The `babyName` method serves the calling object - here the calling object is `myKid`.

For its service `babyName` needs access to the “name” field of the calling object, `myKid`.

That’s how it builds a baby name
String someName = myKid.babyName();

The calling object created after class def is written

the method - it’s written before any object created;

requires Infant obj name
to form its return String
public String babyName()
{
    String realName = this.name;

    String who = ("Baby " + realName);
}

return who;
}
```java
public String babyName() {
    String realName = this.name;
    String who = ("Baby " + realName);
    return who;
}
```

Here, the Java keyword **this** explicitly references the calling object (such as myKid, jillsKid,...)
public class Infant {

    private String name;
    private int age;

    public Infant(String name, int age) {// constructor
        this.name = name; // LHS - from to-be-created obj
        this.age = age;
    }

    public String getName() { return name; }

    public int getAge() { return age; }

    public void anotherMonth() { age = age + 1; }
}

public class Infant {

    private String name;
    private int age;

    public Infant(String name, int age) {
        this.name = name;
        this.age = age;
    }

    Here, the Java keyword **this** explicitly references the calling object that is being created by the call to the constructor. That object's name and age fields are being set equal to the formal parameter values.
Until now: methods are passed, then return primitives, or maybe Strings

We need to study parameter passing more carefully, look at how parameter values can, cannot change.

Let’s look first at methods that return objects
Infant kid = new Infant("Jill",1);
Infant kidTwin = kid.makeTwin("Ivan");

------ a new Infant method ------

public Infant makeTwin(String name){
    int twinAge = this.getAge();
    Infant i = new Infant(name,twinAge);
    return i;
}
public class Infant{
  ..
  public Infant makeTwin(String n)
  {
    twinAge = this.age;
    Or
    twinAge = this.getAge();
  }
  ..
}

p.s.v main(...){
  myKid = new Infant("Jill", 1);
  kidTwin =
    myKid.makeTwin("fred");
Recall the cell model of Java variables:

```java
int number = 5;

number = number + 1;
```

5

number

6

number = number + 1;
Parameter passing in Java

Consider this method:

```java
public void change(int x)
{
    x = x + 1;
}
```

-- now this code -----

```java
int a = 3;
change(a);
System.out.println(a);
```

What's the value of a? (ans: still 3)
Cells and parameter passing: part I

Value of $a$ copied to $x$. Copy works just in one direction!
An aside on computer memory - RAM

It’s laid out as cells/words, and each has an address

A typical memory cell

1001010010

Sometimes these cells have names: n, or pos, s, etc
The way this works for Java primitives, e.g. ints, is straightforward

    int pos = 7;

means:

```
  7

pos
```
Cells and Objects

Key idea: myKid - an Infant object name - does not hold myKid object information.

It holds the address of the location where that information is stored.
This is the **pirate treasure** model:

The box on the ship does **not** hold the treasure. It holds the **map** that tells the location of the treasure.

The data in the myKid cell is the map to the myKid information.
Consider this statement:

```java
myKid.anotherMonth();
```

Where:

```java
public void anotherMonth(){
    age = age + 1;
}
```

The `anotherMonth` method does not alter cell labelled `myKid`.

It alters the information at location referenced by the address in the `myKid` cell.
anotherMonth method increments age by 1

myKid.anotherMonth();

myKid

memory address

101010010110100101

.....

class: Infant
name: Ted
age : 3 -> 4

101010010110100101
Imagine two classes:

An Airport class
A Flight class
**Flight class:**

```java
public class Flight{
    String id;
    String start;
    String end;
    boolean arrived;

    public Flight(String id, String s, String e, boolean here)
    {
        this.id = id;
        start = s;
        end = e;
        arrived=here;
    }

    ....

    myFlight = new Flight("CE777","JFK","LAX",false);
```
Let’s land the flight. airportLAX is in Airport class

airportLAX.landFlight(myFlight);
   //in driver

Airport class includes this method..

public void landFlight(Flight f){
    f.setArrived(true); }

Issue here: we want to change the state of the flight object that’s passed as a parameter.
We want the `landFlight` method to change an attribute of `myFlight`
- we can do it, because `myFlight` is a reference to data, and we aren’t changing that reference (that address).

We’re jumping to that address and changing information there.
Address of myFlight object information

`myFlight` object information

`landFlight` method

`method parameter`
The landFlight parameter **does not** change - it’s the address of the Flight object information. So our calling principle is not violated. But the referenced object itself **does** change state: The plane has arrived.

```
class: Flight
name: CE777
arrived: false -> true
```

```
myFlight
```

```
10111101011010010
10111101011010101
```

```
memory address
```
Formatted printing: printf

double bucks = 129.95;
System.out.println(bucks) - a loser
System.out.printf("$%6.2f", bucks);
Prints $129.95

double bigBucks = 12345.67;
System.out.printf("$%,8.2f", bigBucks); $12,345.67
System.out.printf("$%,8.2f\n", bigBucks);
   $12,345.67 + newline
Writing header lines / return smts

• Count digits in a string
• Any perfect squares between a and b?
• At 6%, how much will $750 earn after 11 years?
• Change Vince’s (an infant’s) name to Cicero.
Algorithms for computing running averages

$\text{Alg}_1$: when $a_n$ comes in, calc $(a_1+\ldots+a_n)/n$, print answer

How many operations with 10,000 $a$'s? 50,000,000 (!)
Alg$_2$: when $a_n$ comes in,
sum = sum + $a_n$
divisor++

print (sum/divisor)

How many operations with 10,000 $a$'s?
30,000 (!)