

## WES-CS GROUP MEETING #3

### Exercise 1: Using Objects and Methods

A delicious treat known as a *s'more* is constructed from the following ingredients:

- 1 graham cracker (broken in half)
- 2 chocolate rectangles
- 4 mini marshmallows

First, try making (and eating!) one *s'more*.

Now let's design some Java programs where the objects are packages containing graham crackers, chocolate bars, and mini marshmallows. Assume that each package has one method: *numInPackage* that tells you how many items (graham crackers, chocolate bars, or mini marshmallows) are currently in the package.

**Part (a).** Describe in English how you would determine whether it's possible to make one *s'more*. How about two *s'mores*?  $n$  *s'mores*?

**Part (b)** Now assume that the package objects are called *crackerPkg*, *chocolatePkg*, and *marshmallowPkg*. Write Java code that determines whether it is possible to make one, two, or  $n$  *s'mores* and in each case either prints "yes" or "no".

**Part (c).** Describe in English how you would determine the *maximum* number of *s'mores* you can make using the ingredients in the three packages.

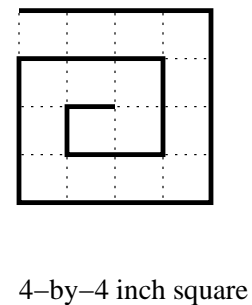
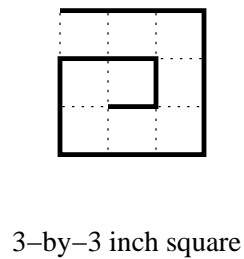
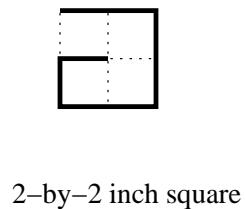
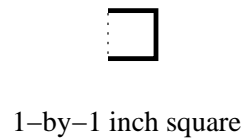
**Part (d).** Write Java code that prints the maximum number of *s'mores* that can be made.

## Exercise 2: Writing a Java Method

Last week you did a long exercise involving the Artist class. One of the problems was to write code to draw a clockwise, square spiral in an n-by-n square. If you didn't write actual code, or you wrote code that works only for a particular n, try writing the method now. The method header is given below:

```
public void drawSpiral( int n )
```

And here are the pictures of some spirals to remind you what they look like.



## Exercise 3: Arithmetic expressions

Translate the following formulas to JAVA code. Assume that all variables are of data type double.

(a) 
$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

(b) 
$$a^2 + b^2 + 2ab \cos(c)$$

(c) 
$$2 \cos^2(a) - 1$$

(d) 
$$\frac{n(n+1)(2n+1)}{6}$$

(e) 
$$ut + 1/2 at^2$$

## Exercise 4: The Time class

For this exercise, we'll work on extending the Time class defined on the next page.

**Part (a).** First, try running the program as it is.

**Part (b).** Now, let's define some new methods for the Time class. Notice that the Time class stores its time in 24-hour format: it has two data members called *hour* and *minute*; *hour* is an integer between 0 and 23 that represents the hours part of the time, and *minute* is an integer between 0 and 59 that represents the minutes part of the time.

Our first new method will be *print12HourTime*, which is like the existing method *print24HourTime*, except that it prints the time in 12-hour format instead of 24-hour format. For example, the 24-hour time 15 20 would be printed as "3:20pm", and the 24-hour time 0 0 would be printed as "midnight".

This method should work as follows:

- If the 24-hour time represents midnight or noon, just print "midnight" or "noon".
- Otherwise, if the time is before noon, print it in 12-hour format (ending with "am").
- Otherwise, print the time in 12-hour format, ending with "pm".

Take a look at the *print24HourTime* method, then write the *print12HourTime* method. Also, add a call to *print12HourTime* in *main*, and compile and run the new program.

**Part (c).** Another useful thing to know is how many minutes there are between two times. For example, it takes 45 minutes to get from noon to 12:45, and it takes 1,395 minutes (23 hours and 15 minutes) to get from 12:45 to noon.

Our next method, *timeRemaining* will tell us how many minutes it takes to get from the time represented by the Time object whose method is called, to a given time. For example, if you have a Time object *now* that represents noon, and another Time object *appointment* that represents 12:45, then the method call *now.timeRemaining( appointment )* will return 45, while the method call *appointment.timeRemaining( now )* will return 1395.

First, figure out (on paper) how to determine the number of minutes it takes to get from one time to another, then write the method.

Add code to *main* to test your method. You may want to do this by reading a time typed in at the terminal, and figuring out how much time there is between that time and the current time (or vice versa). To read one integer from the terminal, use the code on page 5 (i.e., add it to the Time class). You'll also need to put the following line at the beginning of the Time class: `import java.io.*;`

```

import java.util.*;
public class Time {
    /**
     * data members
     */
    private int hour;
    private int minute;

    /**
     * public methods
     */
    /* constructor */
    public Time(int hr, int min) {
        hour = hr;
        minute = min;
    }

    /* print24HourTime */
    public void print24HourTime() {
        System.out.print(hour + ":");
        if (minute < 10) {
            System.out.print("0");
        }
        System.out.println(minute);
    }

    /* main: create a Time object that represents the current time
     * and print that time in 24-hour format
     */
    public static void main(String[] args) {
        int hour = getCurrHour();
        int min = getCurrMinute();
        Time now = new Time(hour, min);
        now.print24HourTime();
    }

    /**
     * private methods
     */
    /* getCurrHour */
    private static int getCurrHour() {
        Calendar cal = new GregorianCalendar();
        return cal.get(Calendar.HOUR_OF_DAY);
    }

    /* getCurrMinute */
    private static int getCurrMinute() {
        Calendar cal = new GregorianCalendar();
        return cal.get(Calendar.MINUTE);
    }
}

```

Code to read one integer value typed in to the terminal.

```
private static int readInt() {
    int num = 0;
    try {
        Reader in = new InputStreamReader(System.in);
        int ch = in.read();
        while (ch != -1 && ch >= '0' && ch <= '9') {
            num = num * 10 + ch - '0';
            ch = in.read();
        }
    } catch (IOException ex) {
        System.out.println("bad number");
    }
    return num;
}
```

## Exercise 5: The Alarm Clock Class

Now we're ready to design a new class that uses the `Time` class. The new class is for your team leader who sometimes parties late into the night and oversleeps the next morning. Your team leader needs an *AlarmClock* class!

The `AlarmClock` class should have three methods:

1. A constructor, whose parameter is the wake-up time (a `Time` object).
2. A *doAlarm* method, that keeps checking the actual time until it is equal to the wake-up time. Then it beeps, and finishes.
3. A *main* method that gets the wake-up time from the user of the program, creates a `Time` object to represent the wake-up time, uses the `AlarmClock` constructor to create a new `AlarmClock` object, tells the user what the current time is and how many minutes remain until the alarm will ring, and finally calls the `AlarmClock` object's *doAlarm* method.

To make the computer beep, use

```
System.out.print('\u0007');
```

Write the `AlarmClock` class and test it.

If you want to improve your `AlarmClock` class, try the following:

1. Your team leader is kind of a heavy sleeper, and so is not likely to be woken up by a single-beep alarm clock. Modify the *doAlarm* method so that it beeps once per second for 60 seconds.
2. Make it easier for your team leader to set the alarm by allowing the wake-up time to be specified in 12-hour format. For example: 1:23pm instead of 13 23. Use the *readInt* method to help you figure out how to read a time in 12-hour format and to convert it to 24-hour format for use by the `Time` constructor.

## Exercise 6: Syntax

For this exercise, you will divide into groups of 2 or 3 to play *concentration*, which will help you review the syntax for different Java constructs.

Each group will get one set of cards; the green ones have an English description of some kind of Java code, the yellow ones have a definition of the syntax for that code, and the pink ones have an example of that kind of code. First, look at the cards and decide which ones match (note that multiple cards from one group may match a single card from another group).

Now put all the cards face down and take turns turning over two cards of different colors at a time. If you turn over matching cards, you take them and go again. The game ends when someone has 4 matching pairs.

(The cards to be used for this exercise can be created by printing the last three pages of this document and then cutting them up. The first page should be printed on green paper, the second on yellow, and the third on pink.)

## Exercise 7: Designing Classes and Methods

The Smith family wants to use a Java program to help them with their weekly grocery shopping. In particular, they want to make sure that the grocery items they plan to buy don't exceed that week's budget, don't include too many junk-food items, and do include enough fruits and vegetables.

So each week, the family wants to create a new shopping plan that includes, for that week, the amount of money available for groceries, the maximum number of junk-food items, and the minimum numbers of fruits and vegetables. Then the family wants to be able to provide a list of grocery items, and to find out whether it is within the budget, and also whether it has an acceptable number of junk-food items, and an acceptable number of fruits and vegetables.

Your job is to design the `ShoppingPlan` and `GroceryItem` classes. For each, decide what data members and methods it should have and draw class diagrams to show your class design. Then say in English what each method should do, and how it should be done (don't forget the constructors). Finally, write Java code for the methods of the `ShoppingPlan` class. You don't have to write the method that would create the weekly list of grocery items if you don't want to; just assume that method works correctly.

You can use an `ArrayList` to hold the list of `GroceryItems` (you should have seen an `ArrayList` already in Assignment 0). To find out how many `GroceryItems` are in an `ArrayList` named `list`, use code like this:

```
int numItems = list.size();
```

And here's how you get the  $k^{\text{th}}$  item from the list:

```
GroceryItem oneItem = (GroceryItem)list.get(k);
```

Note that in Java, counting starts at zero, not one, so to get the first item from the list you need to call `list.get` with `k = 0`, not `k = 1`.

## Exercise 8: Logical Reasoning

Arrange 15 paper clips in 3 rows with 3 in the first row, 5 in the second row, and 7 in the third row.

This is a game for two players. You win by forcing your opponent to pick up the last paper clip. When it's your turn, you play by taking as many paper clips as you like from any row (you may take the whole row if you like) but from one row only. This is the game of NIM and is actually a logical puzzle, for the first player can always win once he knows the winning strategy. The puzzle is to figure out that strategy.

Divide into groups of 2 and try playing the game a few times, alternating who goes first. Think about what strategy to use. If you already know the answer, *don't tell!* You can play to win when you go first; see if your partner can figure out your strategy.

variable declaration  
(no initialization)

use a class constant  
outside the class

variable declaration  
with initialization

object declaration  
(no object creation)

object declaration  
and creation

class declaration

non-constructor  
method declaration

constructor declaration

method call  
(send a message)

assignment statement

<data type> <var name>;

<var name> = <expr>;

<data type> <var name> = <expr>;

<class name>.<class constant>

<class name> <object name>;

<object name> = new <class name>( <args> );

<class name> <object name> =  
new <class name> ( <args> );

```
class <class name> {  
    <class member declarations>  
}
```

<class name>.<method name>( <args> )

```
<modifiers> <ret type> <method name>( <params> ) {  
    <method body>  
}
```

```
<modifiers> <class name>( <params> ) {  
    <method body>  
}
```

<object name>.<method name>( <args> );

```
int x;                               fine = Library.FINE_PER_DAY * 3;
```

```
Artist picasso;
```

```
double d = 5.5;
```

```
Time t = new Time(12, 10);
```

```
public class Time {  
    data member declarations  
    method declarations  
}
```

```
public void drawLineLeft( int len ) {  
    statements  
}
```

```
public Time( int hrs, int mins ) {  
    statements  
}
```

```
picasso.drawLineLeft( 10 );
```

```
picasso = new Artist();
```