CSE 142 Section Handout #9

Cheat Sheet

Inheritance (9.1 - 9.2)

```
public class name extends superclass {
```

- **inheritance**: Forming a new class based on an existing class.
- **extend**: To inherit from another class.
- **superclass**: The "parent" class; the class being extended.
- **subclass**: The "child" class; the class extending another.
- **override**: To replace a superclass method with a new one.

`super` keyword (9.3)

Calling an overridden method from the superclass:

```
super.method(parameters)
```

Calling a constructor from the superclass:

```
super(parameters);
```

```
public class Employee {
  private int years;

  public Employee(int years) {
    this.years = years;
  }

  public int getHours() {
    return 40;
  }

  public double getSalary() {
    return 50000.0;
  }

  public int getVacationDays() {
    return 10 + 2 * years;
  }

  public int getYears() {
    return years;
  }
}

public class Lawyer extends Employee {
  public Lawyer(int years) {
    super(years);
  }

  public double getSalary() {
    return super.getSalary() +
           5000.0 * getYears();
  }

  public int getVacationDays() {
    return super.getVacationDays() + 5;
  }
}
```

Polymorphism puzzles

```
public class Class1 {
    public void method1() {
        System.out.print("C1 m1   ");
        method2();
    }

    public void method2() {
        System.out.print("C1 m2   ");
    }
}
```

```
public class Class2 extends Class1 {
    public void method1() {
        super.method1();
        System.out.print("C2 m1   ");
    }

    public void method2() {
        System.out.print("C2 m2   ");
    }
}
```

The following would be the output from creating an object of each of the above classes and calling its methods:

```
Class1 var1 = new Class1();      // Output:
var1.method1();                  // C1 m1   C1 m2
var1.method2();                  // C1 m2

Class2 var2 = new Class2();
var2.method1();                  // C1 m1   C2 m2   C2 m1
var2.method2();                  // C2 m2
```

Other inheritance notes

- Subclasses cannot directly access private fields they inherit.
- Subclasses do not inherit constructors. If the superclass has one that requires parameters, the subclass must also.

CSE 142 Section Handout #9

Sample Final Exam #5

(based on Autumn 2006's final; thanks to Ruth Anderson)

1. Array Mystery

Consider the following method:

```
public static void arrayMystery(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        if (array[i] < array[i + 1]) {
            array[i] = array[i + 1];
        }
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method `mystery` executes if the integer array in the left-hand column is passed as a parameter to `mystery`.

Original Contents of Array	Final Contents of Array
`int[] a1 = {2, 4};` `arrayMystery(a1);`	_____
`int[] a2 = {1, 3, 6};` `arrayMystery(a2);`	_____
`int[] a3 = {7, 2, 8, 4};` `arrayMystery(a3);`	_____
`int[] a4 = {5, 2, 7, 2, 4};` `arrayMystery(a4);`	_____
`int[] a5 = {2, 4, 6, 3, 7, 9};` `arrayMystery(a5);`	_____

CSE 142 Section Handout #9

2. Reference Semantics Mystery

The following program produces 4 lines of output. Write the output below, as it would appear on the console.

```
import java.util.*;   // for Arrays class

public class Mystery {
    public static void main(String[] args) {
        int x = 1;
        int[] a = new int[4];

        x = x * 2;
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));

        x = x * 2;
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));
    }

    public static void mystery(int x, int[] a) {
        x = x * 2;

        if (x > 6) {
            a[2] = 14;
            a[1] = 9;
        } else {
            a[0] = 9;
            a[3] = 14;
        }

        System.out.println(x + " " + Arrays.toString(a));
    }
}
```

CSE 142 Section Handout #9

3. Inheritance Mystery

Assume that the following classes have been defined:

```
public class Ice extends Fire {
    public void method1() {
        System.out.print("Ice 1   ");
    }
}

public class Rain extends Fire {
    public String toString() {
        return "Rain";
    }

    public void method1() {
        super.method1();
        System.out.print("Rain 1   ");
    }
}
```

```
public class Fire {
    public String toString() {
        return "Fire";
    }

    public void method1() {
        method2();
        System.out.print("Fire 1   ");
    }

    public void method2() {
        System.out.print("Fire 2   ");
    }
}

public class Snow extends Rain {
    public void method2() {
        System.out.print("Snow 2   ");
    }
}
```

Given the classes above, what output is produced by the following code?

```
Fire[] elements = {new Fire(), new Snow(), new Rain(), new Ice()};
for (int i = 0; i < elements.length; i++) {
    System.out.println(elements[i]);
    elements[i].method1();
    System.out.println();
    elements[i].method2();
    System.out.println();
    System.out.println();
}
```

4. File Processing

Write a static method named `halfCaps` that accepts as its parameter a `Scanner` holding a sequence of words and outputs to the console the same sequence of words with alternating casing (lowercase, uppercase, lowercase, uppercase, etc). The first word, third word, fifth word, and all other "odd" words should be in lowercase letters, whereas the second word, fourth word, sixth word, and all other "even" words should be in uppercase letters. For example, suppose the `Scanner` contains the following words.

```
The QUick brown foX jumPED over the Sleepy student
```

For the purposes of this problem, we will use whitespace to separate words. You can assume that the sequence of words will not contain any numbers or punctuation and that each word will be separated by one space. For the input above, your method should produce the following output:

```
the QUICK brown FOX jumped OVER the SLEEPY student
```

Your output should separate each word by a single space. The output may end with a space if you like. Note that the `Scanner` may contain no words or may contain an even or odd number of words.

5. File Processing

Write a static method named `countWords` that accepts as its parameter a `Scanner` for an input file, and that outputs to the console the total number of lines and words found in the file as well as the average number of words per line. For example, consider the following input file:

```
You must show: your Student ID card

to 1) a TA or 2) the instructor
before

leaving the room.
```

For the purposes of this problem, we will use whitespace to separate words. That means that some words might include punctuation, as in `"show:"` and `"1)"`. (This is the same definition that the `Scanner` uses for tokens.) For the input above, your method should produce the following output:

```
Total lines = 6
Total words = 19
Average words per line = 3.167
```

The format of your output must exactly match that shown above, including rounding the words per line to 3 decimal places. Notice that some input lines can be blank. Do not worry about rounding the average words per line. You may assume that the `Scanner` contains at least 1 line of input.

CSE 142 Section Handout #9

6. Array Programming

Write a static method named `mode` that takes an array of integers as a parameter and that returns the value that occurs most frequently in the array. Assume that the integers in the array appear in sorted order. For example, if a variable called `list` stores the following values:

```
int[] list = {-3, 1, 4, 4, 4, 6, 7, 8, 8, 8, 8, 9, 11, 11, 11, 12, 14, 14};
```

Then the call of `mode(list)` should return `8` because 8 is the most frequently occurring value in the array, appearing four times.

If two or more values tie for the most occurrences, return the one with the lowest value. For example, if the array stores the following values, the call of `mode(list)` should return `2` despite the fact that there are also three 9s:

```
int[] list = {1, 2, 2, 2, 5, 7, 9, 9, 9};
```

If the array's elements are unique, every value occurs exactly once, so the first element value should be returned. You may assume that the array's length is at least 1. If the array contains only one element, that element's value is considered the mode.

7. Array Programming

Write a static method named `contains` that accepts two arrays of integers *a1* and *a2* as parameters and that returns a `boolean` value indicating whether or not *a2*'s sequence of elements appears in *a1* (`true` for yes, `false` for no). The sequence of elements in *a2* may appear anywhere in *a1* but must appear consecutively and in the same order. For example, if variables called `list1` and `list2` store the following values:

```
int[] list1 = {1, 6, 2, 1, 4, 1, 2, 1, 8};
int[] list2 = {1, 2, 1};
```

Then the call of `contains(list1, list2)` should return `true` because `list2`'s sequence of values `{1, 2, 1}` is contained in `list1` starting at index 5. If `list2` had stored the values `{2, 1, 2}`, the call of `contains(list1, list2)` would return `false` because `list1` does not contain that sequence of values. Any two lists with identical elements are considered to contain each other, so a call such as `contains(list1, list1)` should return `true`.

You may assume that both arrays passed to your method will have lengths of at least 1. You may not use any `String`s to help you solve this problem, nor methods that produce `String`s such as `Arrays.toString`.

CSE 142 Section Handout #9

8. Critters

Write a class `Shark` that extends the `Critter` class from Homework 8. `Shark` objects should alternate between moving to the north and south as follows: first move 1 step north, then 2 steps south, then 3 steps north, then 4 steps south, then 5 steps north, then 6 steps south, and so on, each time moving one farther than previously.

You may add anything needed (fields, other methods, constructors, etc.) to implement this behavior appropriately.

9. Classes and Objects

Suppose that you are provided with a pre-written class `ClockTime` as described at right. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

Write an instance method named **`advance`** that will be placed inside the `ClockTime` class to become a part of each `ClockTime` object's behavior. The `advance` method accepts a number of minutes as its parameter and moves the `ClockTime` object forward in time by that amount of minutes. The minutes passed could be any non-negative number, even a large number such as 500 or 1000000. If necessary, your object might wrap into the next hour or day, or it might wrap from the morning ("AM") to the evening ("PM") or vice versa. A `ClockTime` object doesn't care about what day it is; if you advance by 1 minute from 11:59 PM, it becomes 12:00 AM.

For example, if the following object is declared in client code:

```
ClockTime time = new ClockTime(6, 27, "PM");
```

The following calls to your method would modify the object's state as indicated in the comments.

```
time.advance(1);        //  6:28 PM
time.advance(30);       //  6:58 PM
time.advance(5);        //  7:03 PM
time.advance(60);       //  8:03 PM
time.advance(128);      // 10:11 PM
time.advance(180);      //  1:11 AM
time.advance(1440);     //  1:11 AM  (1 day later)
time.advance(21075);    //  4:26 PM  (2 weeks later)
```

Assume that the state of the `ClockTime` object is valid at the start of the call and that the `amPm` field stores either `"AM"` or `"PM"`.

```
// A ClockTime object represents
// an hour:minute time during
// the day or night, such as
// 10:45 AM or 6:27 PM.

public class ClockTime {
    private int hour;
    private int minute;
    private String amPm;


    // Constructs a new time for
    // the given hour/minute
    public ClockTime(int h,
            int m, String ap)

    // returns the field values
    public int getHour()
    public int getMinute()
    public String getAmPm()

    // returns String for time;
    // example: "6:27 PM"
    public String toString()


    // your method would go here

}
```

CSE 142 Section Handout #9
Solutions

1.

Array	Final contents
{2, 4}	{4, 4}
{1, 3, 6}	{3, 6, 6}
{7, 2, 8, 4}	{7, 8, 8, 4}
{5, 2, 7, 2, 4}	{5, 7, 7, 4, 4}
{2, 4, 6, 3, 7, 9}	{4, 6, 6, 7, 9, 9}

2.

```
4 [9, 0, 0, 14]
2 [9, 0, 0, 14]
8 [9, 9, 14, 14]
4 [9, 9, 14, 14]
```

3.

```
Fire
Fire 2   Fire 1
Fire 2

Rain
Snow 2   Fire 1   Rain 1
Snow 2

Rain
Fire 2   Fire 1   Rain 1
Fire 2

Fire
Ice 1
Fire 2
```

4. Three solutions are shown.

```
public static void halfCaps(Scanner input) {
    boolean odd = true;
    while (input.hasNext()) {
        String next = input.next();
        if (odd) {
            System.out.print(next.toLowerCase() + " ");
        } else {
            System.out.print(next.toUpperCase() + " ");
        }
        odd = !odd;
    }
}
```

```
public static void halfCaps(Scanner input) {
    int count = 0;
    while (input.hasNext()) {
        if (count % 2 == 0) {
            System.out.print(input.next().toLowerCase() + " ");
        } else {
            System.out.print(input.next().toUpperCase() + " ");
        }
        count++;
    }
}
```

```
public static void halfCaps(Scanner input) {
    while (input.hasNext()) {
        System.out.print(input.next().toLowerCase() + " ");
        if (input.hasNext()) {
            System.out.print(input.next().toUpperCase() + " ");
        }
    }
}
```

5.

```
public static void countWords(Scanner input) {
    int lineCount = 0;
    int wordCount = 0;

    while (input.hasNextLine()) {
        String line = input.nextLine();
        lineCount++;
        Scanner lineScan = new Scanner(line);
        while (lineScan.hasNext()) {
            String next = lineScan.next();
            wordCount++;
        }
    }

    double averageWords = (double) wordCount / lineCount;
    System.out.println("Total lines = " + lineCount);
    System.out.println("Total words = " + wordCount);
    System.out.printf("Average words per line = %.3f\n", averageWords);
}
```

6.

```
public static int mode(int[] a) {
    int count = 1;
    int maxCount = 1;
    int modeValue = a[0];

    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] == a[i + 1]) {
            count++;
            if (count > maxCount) {
                modeValue = a[i];
                maxCount = count;
            }
        } else {
            count = 1;
        }
    }

    return modeValue;
}
```

7. Four solutions are shown.

```
public static boolean contains(int[] a1, int[] a2) {
    for (int i = 0; i <= a1.length - a2.length; i++) {
        boolean found = true;
        for (int j = 0; j < a2.length; j++) {
            if (a1[i + j] != a2[j]) {
                found = false;
            }
        }
        if (found) {
            return true;
        }
    }
    return false;
}

// varation of first solution that uses count instead of boolean
public static boolean contains(int[] a1, int[] a2) {
    for (int i = 0; i <= a1.length - a2.length; i++) {
        int count = 0;
        for (int j = 0; j < a2.length; j++) {
            if (a1[i + j] == a2[j])
                count++;
        }
        if (count == a2.length)
            return true;
    }
    return false;
}

public static boolean contains(int[] a1, int[] a2) {
    int i1 = 0;
    int i2 = 0;
    while (i1 < a1.length && i2 < a2.length) {
        if (a1[i1] != a2[i2]) {  // doesn't match; start over
            i2 = 0;
        }
        if (a1[i1] == a2[i2]) {
            i2++;
        }
        i1++;
    }

    return i2 >= a2.length;
}

public static boolean contains(int[] a1, int[] a2) {
    for (int i = 0; i < a1.length; i++) {
        int j = 0;
        while (j < a2.length && i + j < a1.length && a1[i + j] == a2[j])
            j++;
        if (j == a2.length)
            return true;
    }
    return false;
}
```

8. Two solutions are shown.

```
public class Shark extends Critter {
    private int count;
    private int max;
    private Direction direction;

    public Shark() {
        count = 0;
        max = 1;
        direction = Direction.NORTH;
    }

    public Direction getMove() {
        count++;
        if (count > max) {
            count = 1;
            max++;
            if (direction == Direction.NORTH) {
                direction = Direction.SOUTH;
            } else {
                direction = Direction.NORTH;
            }
        }
        return direction;
    }
}
```

```
public class Shark extends Critter {
    private int count = 0;
    private int max = 1;

    public Direction getMove() {
        count++;
        if (count > max) {
            count = 1;
            max++;
        }
        if (max % 2 == 0) {
            return Direction.SOUTH;
        } else {
            return Direction.NORTH;
        }
    }
}
```

9. Two solutions are shown.

```
public void advance(int mins) {
    minute += mins;                 // add into 'minute' field, then pull hours out 1 at a time
    while (minute >= 60) {
        minute -= 60;
        hour++;
        if (hour == 12) {
            if (amPm.equals("AM")) {
                amPm = "PM";
            } else {
                amPm = "AM";
            }
        } else if (hour > 12) {
            hour = 1;
        }
    }
}
```

```
public void advance(int mins) {
    int totalMinutes = hour % 12 * 60 + minute;   // convert into absolute minutes representation
    if (amPm.equals("PM")) {
        totalMinutes += 12 * 60;   // add 12 hours for PM
    }
    totalMinutes = (totalMinutes + mins) % (24 * 60);   // move forward; throw away extra days
    hour = totalMinutes / 60;                           // convert back to hour/minute/amPm
    minute = totalMinutes % 60;
    if (hour >= 12) { amPm = "PM"; }
    else            { amPm = "AM"; }
    if (hour == 0)  { hour = 12; }
    else            { hour = hour % 12; }
}
```