HOMEWORK PROBLEMS 11

1) Write a constructor for the class `Fox` that calls the constructor of the superclass of `Fox`, passing it 5.

2) Write a constructor for the class `Chicken` that initializes the `x` instance variable with the parameter `x`.

3) Write a constructor for the class `Owl` that calls another constructor in `Owl` passing it 5.

4) Why does a call with late binding take more time at runtime than a comparable call with compile-time binding?

5) If you define an `equals` method for a class, why should its parameter be type `Object`?

6) What is unusual about the classes below? Is there anything wrong with them? Does the `f` in `Bye` override the `f` in `Ave`? Compile a test program to check your answers.

```
class Ave
{
   public void f()
   {
      System.out.println("hello");
   }
}
//================================================
class Bye extends Ave
{
   public int f()
   {
      return 5;
   }
}
```

Write test programs similar to the above to determine the answers to the following questions: Can a public `int f()` method override a `protected int f()` method? Can a `protected int f()` method override a `public int f()` method? If `B` is a subclass of `A`, can a `public A f()` method override a `public B f()` method? Can a `public B f()` method override a `public A f()` method?

7) What does the following program display? *Hint*: Late binding is in effect. Run the program to check your answer.

```
 1 class C11h7
 2 {
 3    public static void main(String[] args)
 4    {
 5       Arc a = new Arc ();
 6       Bam b = new Bam ();
 7       a.g();
 8       b.g();
 9    }
10 }
11 //================================================
12 class Arc
13 {
14    void f()
15    {
16       System.out.println("f in Arc");
17    }
```

```
18    //---------------------------------
19    void g()
20    {
21       f();
22    }
23 }
24 //================================================
25 class Bam extends Arc
26 {
27    void f()
28    {
29       System.out.println("f in Bam");
30   }
31 }
```

8) Create a class `Any` and a subclass `Bod`. Both `Any` and `Bod` should have public `f` methods with identical signatures. Both `Any` and `Bod` should have a public `int` field `x` but with different initial values. Thus, a `Bod` object has two `f` methods and two `x` instance variables. What happens when you execute in `main` the following code:

```
Any a;
a = new Bod();              // a points down
a.f();                      // which f() is executed?
System.out.println(a.x);    // which x is displayed
```

When does the binding in the call of `f` occur? When does the binding of `x` occur (i.e., when is the identifier associated with a storage location)?

9) What will be displayed by the following program? Run the program to check your answer.

```
class C11h9
{
   public static void main(String[] args)
   {
      Cut a = new Cut();
      String s = "hello";
      a.f(s);
      Integer i = 1;
      a.f(i);
   }
}
//================================================
class Cut
{
   public void f(String s)
   {
      System.out.println("String parameter");
   }
   //---------------------------------
   public void f(Object o)
   {
      System.out.println("Object parameter");
   }
}
```

10) Can a private method be overridden? Compile a test program to check your answer.

11) Can a private instance variable be shadowed. Run a test program to check your answer.

12) `finalize` is a method in `Object` that is inherited by the `C` class in the program below. It is executed when the memory in use by an object is reclaimed by the garbage collector. `System.gc` activates the garbage collector. What happens when the program below is executed?

```
class C11h12
{
   public static void main(String[] args)
   {
      Can c = new Can ();
      c = null;        // object is now subject to reclamation
      System.out.println("Calling garbage collector");
      System.gc();
   }
}
//=======================================================
class Can
{
   int x = 1;
   protected void finalize() throws Throwable
   {
       super.finalize();
       System.out.println("in finalize");
   }
}
```

13) Create a class in which you include the following method:

```
public Class getClass()
{
   System.out.println("hello");
}
```

What happens when you compile the class? Explain.

14) If `f` is not final, when does binding occur for the call

```
r.f();
```

If `f` is final in class `Gum`, `r` has type `Gum`, and `r` points to a class `Gum` object, when does binding occur? If `f` is final in class `Gum`, `r` has type `Gum`, and `r` points down to a class `Drop` object, when does binding occur? What is the advantage of marking methods `final`?

15) Is `String` a final class? Compile a test program to check your answer.

16) Is it legal for a constructor to first execute some code and then call another constructor of the same class. For example, is the constructor below legal? Compile a test program to check your answer.

```
public Cry()
{
   System.out.println("hello");
   this(5);           // does this statement have to be first?
}
```

17) Will the compiler insert a call of the superclass constructor in a constructor if the constructor calls another constructor of the same class on its first line. For example, suppose `Bat` is a subclass of `Att`. Does the compiler insert a call of the `Att` constructor at the beginning of the `Bat` constructor below?

```
public Bat()
{
    // Does the compile insert a call of an Att constructor here?
    this(7);
}
```

If it does, would not that mean that the `Att` constructor would be called twice (once by each `Bat` constructor)? Run a test program to check your answer.

18) Incorporate the following code in a program and execute it:

```
String s1 = new String("hello");
String s2 = new String("hello");
ArrayList<String> q1 = new ArrayList<String>();
q1.add(s1);
int index = q1.indexOf(s2);
System.out.println(index);
```

`s1` and `s2` are distinct but identical objects. What does `indexOf` return in the code above? Does it find a match for `s2`? What can you conclude about the operation of `indexOf`? Specifically, does it compare the references or does it compare the objects to which those references point? Now repeat with

```
Cold h1 = new Cold();
Cold h2 = new Cold();
ArrayList<Cold> q2 = new ArrayList<Cold>();
q2.add(h1);
index = q2.indexOf(h2);
System.out.println(index);
```

where `Cold` is defined as

```
class Cold
{
   private int x = 3;
}
```

Why do the two cases above give different results?

19) Modify the program below by adding an `equals` method to the `Dude` class. Test you `equals` method to make sure it is working correctly.

```
class C11h19
{
   public static void main(String[] args)
   {
      Dude d1 = new Dude();
      Dude d2 = new Dude();
      System.out.println(d1.equals(d2)); // should display true
      d1.set(9, 100);
      System.out.println(d1.equals(d2)); // should display false
   }
```

```
}
//===============================================
class Dude
{
   private int[] ia;
   //---------------------------------
   public Dude()
   {
      ia = new int[10];
   }
   //---------------------------------
   public void set(int index, int value)
   {
      ia[index] = value;
   }
}
```

20) Same as homework problem 19 except add a copy constructor.

21) Write a class that has three `int` fields: `x`, `y`, and `z`. This class should have four constructors: one with no parameters, one with one parameter, one with two parameters, and one with three parameters. The constructor with three parameters should set `x`, `y`, and `z` to the values of its parameters. The one with two parameters should set `x` and `y` to the values of its parameter, and set `z` to 30. The one with one parameter should set `x` to the value of its parameter, and `y` and `z` to 40 and 50, respectively. The one with no parameters should set `x`, `y`, and `z` to 60, 70, and 80 respectively. Each constructor, except for the one with three parameters, should contain exactly one statement (not three statements on one line). Include a `display` method in your class that displays the values of `x`, `y`, and `z`. Write a program that invokes each constructor and displays the resulting values of `x`, `y`, and `z`.

22) Write a program with classes `M1`, `M2`, and `C11h22` defined as follows:

`M1`: `M1` has a private instance variable `m2` whose type is `M2`. The constructor for `M1` creates an `M2` object and assigns its reference to `m2`.

`M2`: `M2` has a private instance variable `m1` whose type is `M1`. The constructor for `M2` creates an `M1` object and assigns its reference to `m1`.

`C11h22`: `C11h22` contains `main`. `main` creates an `M1` object and assigns its reference to `r1` whose type is `M1`. `main` also creates a `M2` object and assigns its reference to `r2` whose type is `M2`.

What is wrong with this program?

23) Here is a possible fix for the problem with the program in homework problem 22:

Delete the constructor in `M1`. Thus, `M1` will have only the default constructor inserted by the compiler. To the `M1` class, add a static variable `m1` of type `M` and a static method `getReference`. `M1.getReference` checks if `m1` is `null`. If `m1` is `null`, `M1.getReference` calls the constructor for `M1`, assigning the reference returned to `m1`, and it calls `M2.getReference`, assigning the reference returned to `m2`. `M1.getReference` terminates by returning `m1`. Make the parallel changes to `M2`. Change the body of `main` to

```
M1 r1 = M1.getReference();
M2 r2 = M2.getReference();
```

Draw a diagram that shows `r1`, `r2`, the objects created. In your diagram, show to what objects `r1`, `r2`, `m1`, and `m2` point. Does the proposed fix work?

24) Here is another possible fix for the problem with the program in homework problem 22:

The constructor for `M1` should call the constructor for `M2`, passing it `this`. It should assign the reference returned to `m2`. Add a public accessor method `getm2` to `M1` that returns the value of `m2`. The constructor for `M2` should assign the parameter it is passed to `m1`. Change the body of `main` to

```
M1 r1 = M1();
M2 r2 = r1.getm2;
```

Does the proposed fix work?

25) Write a copy constructor for the following class:

```
import java.util.Random;
class Rue
{
   private Random r;
   //---------------------------------
   public Rue()
   {
      r = new Random();
   }
}
```

Test your copy constructor. Does it create a copy that behaves exactly like the original? Is a `Random` object immutable?

26) Create classes `Aye`, `Boo`, and `X`. `Boo` is a subclass of `Aye`. `X` is unrelated to `Aye` and `Boo`. Suppose the following statements are executed:

```
Aye a1 = new Aye();
Aye a2 = new Aye();
Boo b = new Boo();
X x = new X();
```

Which of the following expressions are then `true`:

```
b instanceOf Boo
b instanceOf Aye
b instanceOf X
a1.getClass() == a2.getClass()
a1.getClass() == b.getClass()
a1.getClass() == x.getClass()
```

Run a test program to check your answers. Based on the results, describe how the `instanceOf` operator works. How does it differ from the `getClass` method?

27) Create two classes `Art` and `Bag`. `Bag` is a subclass of `Art`. Both contain an instance method named `shell` with no parameters whose body is empty. `main` should execute:

```
Art a = new Bag();
for (long i = 1; i < N; i++)
  a.shell();
```

Adjust the value of `N` so total execution is roughly 5 seconds. Write and run a second program in which `main` executes

```
for (long i = 1; i < N; i++)
   shell();
```

where `shell` is a static method with an empty body. Use the same value for `N` as in the first program. How does the execution time compare with that of the first program. Is there a noticeable difference? If so, explain why.

28) Suppose an OO program has objects with a high degree of **coupling**. Coupling results when one object accesses the data in another object. Some coupling is generally necessary. But too much coupling is bad. It makes objects inter-dependent, which, in turn, makes, modification or reuse more difficult. Suggest a way to reduce coupling. Specifically, suppose object `A` accesses the data in object `B`. How can this interaction be modified to reduce coupling?

29) Suppose `B` is a subclass of `A`. Which of these overloadings are legal? Which are illegal?

```
public int f()    // in superclass
protected int f() // in subclass (less access)

protected int f() // in superclass
public int f()    // in subclass (more access)

public A f()      // in superclass
public B f()      // in subclass (return type is subclass)

public B f()      // in superclass
public A f()      // in subclass (return type is superclass)
```