AP Computer Science Name Key. Per ____

Chapter 5 Notes – Enhancing Classes

5.1 Java Programs

Java program – *Software* written to solve a problem.

A Java program is made up of *classes*. One of the classes is where the program starts when it is run and is called a *driver* or *tester* class. The driver or tester class contains the `main` method, no objects are created of this class.

The driver class creates *objects* of the other classes in the program. Many objects can be created or instantiated from each class. This is called having many *instances of a class* and each instance is called an *object.*

Each instance of a class (or object) has it's own copy of all non-static *instance variables* and *methods*. To invoke or execute an objects methods, the object name is followed by the *dot* operator and method name.

Classes consist of the following:

- **instance variables** - store the *state* of an object, what an object *knows* (student name, gpa, etc)
- **class variables** – variables at the class level are marked `static.` Only one copy is created and is used by all objects of the given class.
- **constructor methods** – tells how an object will be *initialized* when created using the `new` reserved word
- **methods** – provide *behavior* for the object, what the object can *do.* Some methods perform a function and return information. Other methods can perform a function and not return information, these methods are `void` methods.

Constructor – Special method in class and is invoked (called) to create an *object.*

- name of the class is used as the constructor name
- constructor does not have a return type
- responsible for initializing the object, all instance variables are initialize in the constructor

Constructors are marked `public`. What would happen if the constructor was marked `private`?

Couldnt instantiate class – no object could be created.

Data scope – Where variables can be used or referenced.

- *Instance* and *class variables* can be used in any method in the class.
- Variables created in a method are called *local variables* and only exist while the method or block in the method {...}, is executing. As soon as the method or block completes, the variable is deleted and removed from memory.

```
public class SomeClass {

     private int a,b;

     public double average(int a, int b) {
        double avg;
        avg = (a + b) / 2.0;
        {
           int x = 4;
        }
        return avg + x;
     }
}
```

not same var.

– local var.

← x only exists in { }

compile error

x doesnt exist.

5.2 Method Parameters

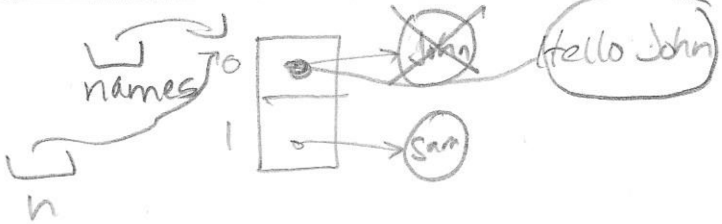
Object parameters – Objects can be passed as parameters to a method call. When objects are passed, the address of the actual object is passed. Any changes made in the method call are made to the actual object. The reference variable passed to the method and the parameter in the method header are *aliases.*

```
.
.
String[] names = {"John","Sam"};

obj.printGreeting (names);
System.out.println (names[0]);
.
```

```
.
.
public void printGreeting(String[] n)

{
   n[0] = "Hello " + n[0];
   System.out.println (n[0]);
}
```

Output:

Hello John
Hello John.



Primitive Type parameters - When primitive type variables are passed as parameters, a *copy* is made of the variable. Changes made to the passed copy DO NOT affect the variable passed.

```
.
int x = 3;
obj.setGrade (x);

System.out.println(x);
.
```

```
public void setGrade( int a)
{
   System.out.println (a);
   a = 10;
   System.out.println(a);
}
```

Output:

3
10
3

Method Composition – A method invocation can also be passed as a parameter. For example, assume `x` is a `double` variable and assigned a value. Assume `half` is a function which receives a `double` and returns the value divided by `2.0`. (`half(4.0) -> 2.0`). Write an expression whose value is 1/8 the value of `x` by only calling the `half` method in a *single* expression.

(half (half (half (x)))

$f(x) = x/2$

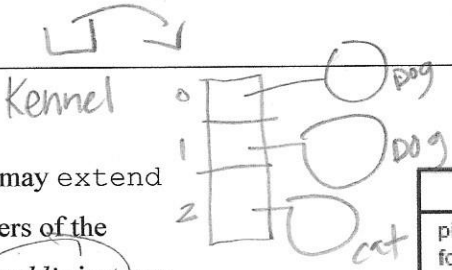
$f(4) \rightarrow 2$

$f(f(4)) \rightarrow 1$

$f(f(f(4))) \rightarrow \frac{1}{2}$

$4 \cdot \frac{1}{8} = \frac{1}{2}$

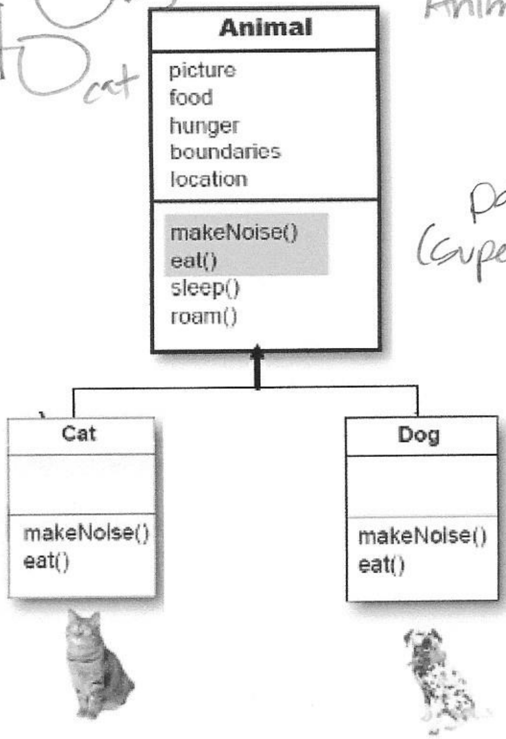
5.3 Inheritance



Animal[] Kennel = new Animal[3];

Subclasses and superclasses – Subclasses may `extend` superclasses. The subclass will inherit the members of the superclass. "Members of the class" mean all the *public* instance variables and methods.

The subclass can add new methods and instance variables of its own and it can *override* the methods it inherits from the superclass.



parent (superclass)

child Subclass

```
public class Dog extends Animal {

  public void makeNoise() {
      roam();
      S.o.p (picture + " Bow wow");
  }
  :
}
```

- Dog & Cats override some methods from parent class to do something different.
- Each have diff. implementations of inherited methods.

5.4 `Object` Class

EVERY object is a subclass of the `Object` class without explicitly using the `extends Object` on the class heading. Classes that extend another class have a *IS-A* relationship.

The `Object` class contains the `equals` and `toString()` methods:

```
public boolean equals(Object obj)
{
   // returns true if this object and obj are the
   // same object (if the references are the same)
}


public String toString()
{
   return getClass().getName() + '@' + Integer.toHexString(hashCode())
}
```

- Each class you write inherits these methods from Object.
- Override to provide implementation specific to your class.

We will want to provide our own versions of the `equals` and `toString` methods for the classes we write. This is called *method overloading.*

Show example toString().

`equals()` Method– Override the `equals()` method provided in the `Object` class to determine if two objects are equal. You specifiy which *instance variables* make the objects equal. You must *cast* the passed object to the class you are using. For example, a social security number in an `Employee` class, a name and address in a `Student` class, a title and author in a `Book` class.

```
public class Employee
{
   private int idNum;

   public int getIdNum()
   {
      return idNum;
   }

   @Override
   public boolean equals (Object o)
   {
      if (o instanceof Employee)
         return (this.idNum == ((Employee)o).getIdNum());
      else
         return false;
   }
}
```

instruction to compiler to give error if your header not correct.

is o created from Employee class?

- Cast o to look like Employee.
- Object doesn't have getIdnum().

- Define the `equals()` method for the `Box` class. `Box` objects are considered equal if they have the same `volume`, which is returned as a `double` value (HINT!!) from the `getVolume()` method in the `Box` class.

```
public boolean equals (Object o) {
  if (o instanceof Box)
    return math.abs(((Box)o).getVolume() - gtVolume()) <.0001;
  else
    return false;
}
```

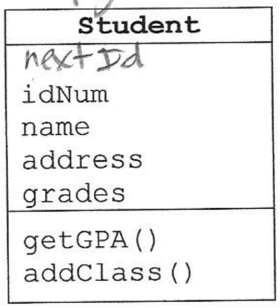
`toString()` Method – Every class should override the `Object` class `toString()` method which returns the object in a string form. When a call is made to print the object, decide what information is important and return a formatted string to be printed.

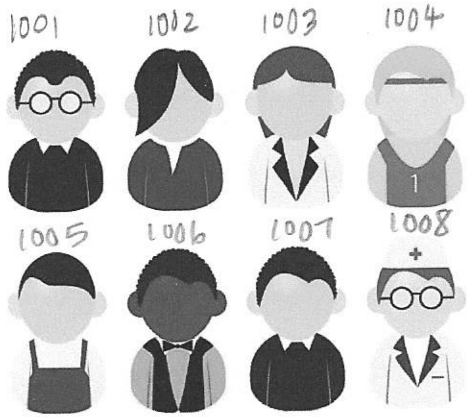
```
public class Student {

  // instance variables
  private String name;
  private int gpa;
  .
  .
  public String toString()
  {
     return "Student Name: " + name + "\nGPA: " + gpa;
  }

}
```

5.5 Static Class Members

Big Picture – In the Java programming language, we can write classes to represent multiple objects of the same type. For example, if we are designing a program to hold student information for a school, one of the classes we would create would be `Student`, because all students will have the same basic information, name, address, class list, grades, emergency contact information, etc. We only have to write the `Student` class once, and then create many `Student` objects from the class. Each `Student` object will keep track of it's own unique data.

* each object gets it's own copy of var and methods. defined in this class

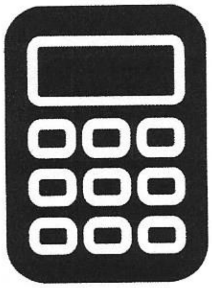




1009
nextId

private static int nextId = 1000;

Now, think about the `Math` class. This class acts like a calculator, in that we only need one calculator, not different instances of the calculator. The `Math` class contains mathematical methods (functions) which accept data and return data. These methods are *utility* methods, they are not dependent on information stored in a particular object (*instance data*). Whenever we need to perform a mathematical computation, we can invoke the stand alone method. These *utility* methods are `static` methods, meaning they are not invoked through an instance of an object. Only one copy is needed for all of our classes to use.



Math.abs(x);
→ class name

There are scenarios when designing our software where we want to keep one copy of a variable for all objects to share, like the one copy of the `Math` methods for all objects to share. For example, think about your student id number. Each time a new `Student` object is created, it is assigned a unique id number. How will our software know which number comes next? Where can we store this information? We will store this number in a `static` variable defined within the `Student` class.

`static` variables – Only one copy of the variable is created for all objects of the class to share. These variables are called *class variable*. All instances of the class (objects) share the static variables.

instance variables = 1 per object

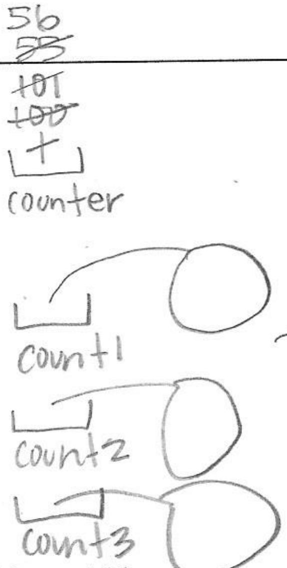
static variables = 1 per class

static variables

```
public class MyCounter
{
   private static int counter;

   public MyCounter (int startVal)
   {
      counter = startVal;
   }
   public MyCounter()
   {
      counter = 1;
   }

/**
*  Method to return and increment static variable counter
*/
   public int nextValue()
   {
        return counter++;
   }
}
```

2 constructors.

```
public class CounterTest
   {
        public static void main (String[] args)
        {
           MyCounter count1 = new MyCounter();
           MyCounter count2 = new MyCounter(100);

           System.out.println(count1.nextValue());
           MyCounter count3 = new MyCounter(55);
           System.out.println(count1.nextValue());

        }
   }
```



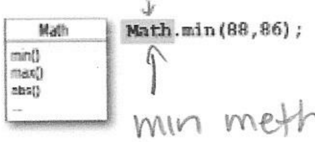
Output
100
55

3 objects have own copy of nextValue() but share 1 copy counter.

Suppose 5 objects of type `MyCounter` are instantiated. How many instances of the variable `counter` are there? 1

`static` methods – Methods marked `static` perform an operation for the entire class, not its individual object. Static methods cannot access *non-static instance data* (since only the object can access its own instance data) or *non-static methods*. Any client (separate program) can call a static method without having to create an object of the class in which the methods are defined. To call a static method, type the class name followed by the dot operator and method name.

Call a static method using a class name



`Math.min(88,86);`

min method invoked through class name.

Call a non-static method using a reference variable name

```
Song t2 = new Song();
t2.play();
```

constants – A variable that cannot be changed once initialized. Constants are marked `static final` and are all UPPERCASE. Since they are marked `static`, only one copy is created for all objects of the class type to share.

```
public static final CURRENT_YEAR = 2009;
```

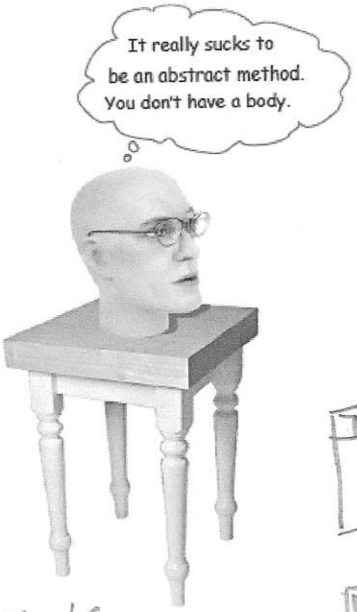
— 1 copy created.

5.6 abstract classes and interfaces

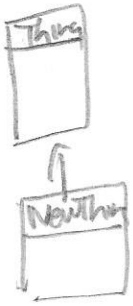
`abstract` – A method marked `abstract` is a method that does not have an *implementation*. There is no body of code defined for an abstract method. The header of the method, including its parameter list, is simply followed by a semi-colon.

```
public abstract void eat();
```

< ; no { }
no implementation



abstract class - If you declare an abstract method, you MUST mark the class abstract as well. You can't have an abstract method in a non-abstract class. Abstract classes cannot be *instantiated.* Classes which `extend` abstract classes, MUST provide the implementation for all of the inherited abstract methods.



```
public abstract class Thing {
    // contains abstract and non-abstract methods.
}
```

* no Thing objects.

```
public class NewThing extends Thing{
    // must provide implementation for all inherited
    // abstract methods.
}
```

new Thing() < error.

interface – A collection of constants and abstract methods. An interface cannot be *instantiated.* Classes that `implement` an interface, provide a "contract" between users and the programmer that the abstract methods will be written according to the interface.

* no implementation at all.

A class that implements an interface uses the reserved word `implements` followed by the interface name. A class can *implement* several different interfaces but can *extend* only one class.

When a class implements an interface, it can use all of the *constants* defined in it. This lets several classes share a set of constants. The interface construct formally tells us how we can interact with a class and is the basis for a powerful programming technique called *polymorphism.*

```
public class OneThing extends Thing implements T1, T2, T3 {
    // must provide code for All defined abstract methods.
}
```

5.7 Comparable Interface

`compareTo()` – In order to utilize the sorting and collections classes provided with Java, you will need to implement the `Comparable` interface with one abstract method `compareTo()`.

Listed below is the entire interface `Comparable`, provided in the Java API.

```
public interface Comparable
{
   \**
    *  Compares the executing object to the parameter to determine their
    *  relative ordering.  Returns an integer that is less than, equal to, or
    *  greater than zero if the executing object is less than, equal to or
    *  greater than the parameter, respectively.
    *\
   public int compareTo(Object o);
}
```

`compareTo()` – Accepts a parameter of type `Object`. You must *cast* the passed object to the class you are using.

```
public class Student implements Comparable
{
   private double gpa;

   public int compareTo(Object other)
   {
      Student s = (Student)other;
      if (this.getGpa() = s.getGpa())
            return 0;
      else if (this.getGpa() > s.getGpa())
               return 1;
      else
            return -1;
   }
}
```

↖ must define implementation for methods defined in this interface.

Generally, the same variables used in `equals()` method are used in the `compareTo()` method to compare objects. The "contract" for `compareTo()` is that the method will return 0 if the objects are *equal* a negative number if the executing object is *less* than the other object or a positive number if it is *greater*.

– must implement Comparable to use sort method in Arrays class.

import java.util.Arrays;

Box[] boxes = new Box[3];
// Fill array with Box objects.
Arrays.sort(boxes);

Write an entire `Box` class which implements the `Comparable` interface. Each `Box` object will have double instance variables for the `length`, `width` and `height`, an integer instance variable for the box `id` and share a class variable to hold the next id number (start at 1000); a constructor which accepts three double parameters for `length`, `width`, `height` and initializes the box id to the next unique number; and methods `getVolume()`, `toString()`, `equals()` and `compareTo()`. The `equals()` and `compareTo()` methods will use the box's volume for comparison.

```
public class Box implements Comparable {
   private double length, width, height;
   private int idNum;
   private static int nextId =1000;

   public Box (double l, double w, double h) {
      length=l;
      width=w;
      height=h;
      idnum = nextId++;
   }
   public double getVolume {
      return length * width * height;
   }
   public boolean equals (Object o) {
      Box b = (Box) o;
      return math.abs (getVolume() - b.getVolume())<.0001;
   }
   public int compareTo (Objecto) {
      Box b = (Box) o;
      if (this.equals (b))
         return 0;
      else if (getVolume() > b.getVolume())
         return 1;
      else
         return -1;
   }
```

```
public String toString() {
   return length + " x " + width + " x " + height
      + " Vol: " + getVolume();
}
}
```

5.8 Exceptions

exceptions – An exception is an *error* condition that occurs during the execution of a Java program. Common exceptions are:

null is empty.

- `NullPointerException` – uninitialized object variable.

  null
  b

  `BankAccount b;`

  `b.withdraw(100);` // throws a `NullPointerException`. Must create `b` object with `new`.

- `ArithmeticException` – dividing by zero
- `IllegalArgumentException` – If a parameter does not meet the precondition you can have the program terminate with this error.
- `ClassCastException` — when type isnt in hierarchy
- `ArrayIndexOutOfBoundsException`

5.9 Javadocs

– used to generate documentation.

Javadocs - `@param`, `@return`

```
/**
 * This method accepts the birth year and returns the age
 * @param y year born
 * @return the age in years
 */
public int computeAge(int y)
{
   age = CURRENT_YEAR - y;
   return age;
}
```

@param for each parameter - 1 per line

@param x . . .
@param y . . .
@param z . . .
example.

computeAge

`public int computeAge(int y)`

This method accepts the birth year and returns the age

Parameters:
`y` - year born

Returns:
the age in years

look at Java API.