CSCI 2320 Java Commands

JavaDoc Comments

```java
/**
 * Short program to demonstrate JavaDoc comments.
 *
 * @author Amy Smith
 * @version 02 Feb 2020
 *
 * <p>Longer descriptions go at bottom inside HTML-style
 * paragraph tags.</p>
 */

public class Demo {

   /**
    * Calls calcSum method and displays returned result.
    *
    * @param args unused
    */
   public static void main(String[] args) {
      System.out.println(calcSum(1.0,2.1));
   }

   /**
    * Given to double values returns their sum.
    *
    * @param a first value to be used in sum
    * @param b second value to be used in sum
    * @return sum of the two parameters
    */
   public static double calcSum(double a, double b) {
      return a+b;
   }
}
```

Expected output:

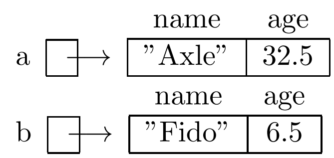
```
3.1
```

JavaDoc comments are specially formatted comments that can be parsed by an external tool (called `javadoc`) that can automatically generate a web-friendly representation of your documentation. The rules for JavaDoc comments of interest to us are:

- Start the comment with `/**` (rather than simply `/*`).
- Begin with a single sentence description of the class or method followed with a period.
- When documenting a class specify the author and version using `@author` and `@version`.
- When documenting a method use `@param paramname description` to document the name and purpose of each parameter. If applicable, use `@return` to describe the value being returned.
- If needed, longer descriptions appear at the bottom of the comment block inside <p></p> tags.

Classes as Objects (non-OOP)

```
1  public class Animal {
2     String name;
3     double age;
4  }
5
6  public class Driver {
7     public static void main(String[] args) {
8        Animal a,b;
9        a= new Animal();
10       a.name= "Axle";
11       a.age= 32.5;
12       System.out.println(a);
13       System.out.println(a.name);
14       System.out.println(a.age);
15
16       b= new Animal();
17       b.name= "Fido";
18       b.age= 6.5;
19       System.out.println(a.age+b.age);
20    }
21 }
```



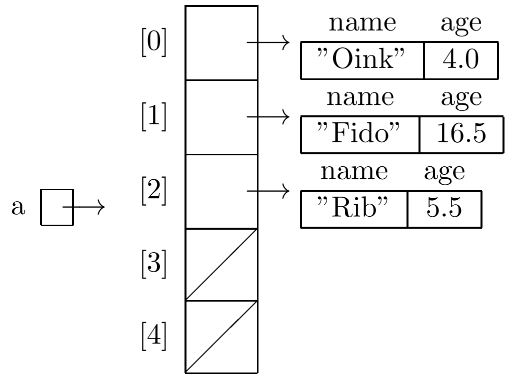
Expected output:

```
Animal@423252d6
Axle
32.5
39.0
```

Classes can be used to create a brand-new type of variable that can combine multiple variables of various types in a single unit. The individual pieces are distinguished from one another by adding a period after the variable name followed by the attribute name. Declaring variables of type `Animal` (line 8) give us the two small boxes labeled `a` and `b` that are depicted containing arrows. We say "`a` is a *reference* to an object" or we might say "`a` contains an *address*". To create an actual object (the double-box that holds a name and a value for age) we have to use the *new* operator (lines 9 and 16). Once an object is created and its address in stored in the appropriate variable we use the access the pieces of object as individual variable using the dot notation. NOTE: In this example, the variable `a.age` is a variable of type `double` and can be used like any other variable of type double. NOTE: On line 12, when we print the contents of `a` we see the arrow/address/reference rather than the values inside the object that `a` is pointing to. When you put a period after `a` it means "go to the object whose address is stored in `a`." NOTE: In order to use a class to instantiate a variable you will need a `main()` method. This is often done in a separate class which we call a "driver" (i.e., it drives the program).

Arrays of Objects (non-OOP)

```java
1  public class Animal {
2     String name;
3     double age;
4  }
5
6  public class Driver {
7     public static void main(String[] args) {
8        int i;
9        Animal [] a;
10       a= new Animal[5];
11
12       a[0]= new Animal();
13       a[0].name= "Oink";
14       a[0].age= 4.0;
15       a[1]= new Animal();
16       a[1].name= "Fido";
17       a[1].age= 16.5;
18       a[2]= new Animal();
19       a[2].name= "Rib";
20       a[2].age= 5.5;
21
22       for (i=0; i<3; i++) {
23          System.out.printf("%-8s %4.1f\n",
24             a[i].name,a[i].age);
25       }
26    }
27 }
```



Expected output:

```
Oink      4.0
Fido     16.5
Rib       5.5
```

It is common to create an array of objects. The variable `a` is declared to be an array of animals on line 9. As with any array we start out with a small box. We create the actual array on line 10. Each of the five array elements is a variable of type `Animal`. That is to say, each of them is a reference. Initially they are all set to the special value `null` which means "not pointing anywhere" (and is represented in the diagram with a slash). So, the variable `a[0]` is an Animal reference. To create an actual `Animal` object we use `new` (line 12, 15, and 18). Once the array is created we can use a loop to conveniently access each part.

```
1 public class Animal {
2    private String name;
3    private double age;
4
5    public void setName(String newName) {
6       name= newName;
7    }
8
9    public String getName() {
10      return name;
11   }
12 }
13
14 public class Driver {
15   public static void main(String[] args) {
16      Animal a;
17      a= new Animal();
18      //a.name= "Fido"; // won't compile b/c name is private
19      a.setName("Fido");
20      System.out.println(a.getName());
21   }
22 }
```

Expected output:

```
Fido
```

Object-Oriented Programming is an approach to writing program that builds programs as a collection of objects that know how to take care of themselves. Each object has *attributes* (characteristics) that are represented using class-wide variables. Each object also has a series of actions is knows how to perform. Objects can see/use their attributes but they often keep those attributes hidden from others. In this example the `Animal` class has two private attributes: `name` and `age`. Because they are private we cannot directly access them from the driver. One way to provide access is through public getter (lines 12–12) and setter (lines 13–13) methods. Notice that methods in a class are accessed using the same dotted notation as attributes. NOTE: In this example we provide no way to set or use the `age` attribute.

Constructors

```java
1  public class Animal {
2     private String name;
3     private double age;
4
5     public Animal() {
6        name= "Not sure";
7        age= 0.0;
8     }
9
10    public Animal(String name, double howOld) {
11       this.name= name;
12       age= howOld;
13    }
14
15    public void show() {
16       System.out.println(name+" is "+age+" years old");
17    }
18 }
19
20 public class Driver {
21    public static void main(String[] args) {
22       Animal a,b;
23       a= new Animal();
24       b= new Animal("Fido",8.5);
25       a.show();
26       b.show();
27    }
28 }
```

Expected output:

```
Not sure is 0.0 years old
Fido is 8.5 years old
```

A constructor is a method that:

- has the same name as the class
- has no return type (not even `void`)
- is called automatically when a new instance of the class is created
- typically for the purpose of initializing the attributes of the class

You can have more than one constructor as long as the signature of the parameter lists differ. In the driver (on line 23) we call the constructor defined in lines 5–8, which causes the attributes of the newly created object to be set. Likewise, on line 24 we call the constructor defined in lines 10–13. When we invoke the `show()` method for each object we find the attributes have been set as expected. NOTE: Line 11 demonstrates use of the keyword `this` which refers to the current object. Normally, when we are working in the method of a class and we want to refers to one of the attributes of the class we simply use the name of the attribute (e.g., lines 6, 7, and 12). In the case of line 11 we prepend `this.` to the name of the attribute because we happen to have a parameter of exactly the same name. In this case `name` refers to the parameter and `this.name` refers to the class-wide attribute.

toString()

```
1  public class Animal {
2     private String name;
3     private double age;
4
5     public Animal(String name, double age) {
6        this.name= name;
7        this.age= age;
8     }
9
10    @Override
11    public String toString() {
12       return name+" is "+age+" years old";
13    }
14 }
15
16 public class Driver {
17    public static void main(String[] args) {
18       Animal a;
19       a= new Animal("Fido",8.5);
20       System.out.println(a);
21    }
22 }
```

Expected output:

```
Fido is 8.5 years old
```

It is conventional to define a `toString()` method for a class which serves the purpose of providing a string representation of an object. Characteristics of a `toString()` method:

- it is named `toString`, takes no parameters, and returns a `String`
- is called automatically when printing an object

Recall that the expected result of printing the reference to an object (as is done on line 20) is for the address of the object to be displayed. Technically, what happens is that the default `toString()` method is invoked (which displays the address). So, when we give an updated/customized version of `toString()` the result of line 20 is that it calls our new method which returns a more helpful value. NOTE: Since we are *overriding* (i.e., providing a new definition for) an existing method it is good form to include the `@Override` directive in front of it.

Inheritance

New classes can be created by *inheriting* from existing classes. On line 24 we create a new `Pet` class that inherits from `Animal`. This causes the new `Pet` class to take on all of the attributes and methods are are found in `Animal`. Besides the obvious benefit of being able to reuse code we get a powerful ability: *polymorphism*. Polymorphism is demonstrated on lines 40 and 45. Notice the variable `b` is declared to be of type `Animal` (line 40), but we are storing a `Pet` object in it (line 45)! One design question when reusing code is whether to inherit or to *aggregate*. Aggregation simply means to use an existing class by making an attribute. In this example we could have declared `Pet` to be it's own class and given it an attribute of type `Animal`. If two entities share an IS-A relationship we typically inherit, however. In this case, it is accurate to say "a Pet IS-A Animal". If the relationship is HAS-A then we typically use aggregation. In this example, if we had used aggregation then we would not have been able to leverage polymorphism. Line 28 shows Java's unusual notation for invoking a parent class' constructor. In this case we let the constructor in `Animal` initialize the name and age attributes for us.

```
1  public class Animal {
2     private String name;
3     private double age;
4  
5     public Animal(String name, double age) {
6        this.name= name;
7        this.age= age;
8     }
9  
10    public double calcEstimatedLifeSpan() {
11       return age*2.0;
12    }
13 
14    public double calcEstimatedHeartRate() {
15       return -20.0*calcEstimatedLifeSpan()+460;
16    }
17 
18    @Override
19    public String toString() {
20       return name+" is "+age+" years old";
21    }
22 }
23 
24 public class Pet extends Animal {
25    private String owner;
26 
27    public Pet(String name, double age, String owner) {
28       super(name,age);
29       this.owner= owner;
30    }
31 
32    @Override
33    public String toString() {
34       return super.toString()+" (Owner: "+owner+")";
35    }
36 }
37 
38 public class Driver {
39    public static void main(String[] args) {
40       Animal a,b;
41       a= new Animal("Algernon",2.2);
42       System.out.println(a+" w/ guessed heart rate of "+
43          a.calcEstimatedHeartRate());
44 
45       b= new Pet("Fido",8.5,"Mary");
46       System.out.println(b+" w/ guessed heart rate of "+
47          b.calcEstimatedHeartRate());
48    }
49 }
```

Expected output:

```
Algernon is 2.2 years old w/ guessed heart rate of 372.0
Fido is 8.5 years old (Owner: Mary) w/ guessed heart rate of 120.0
```

Abstract Classes

```java
1 public abstract class Animal {
2    private String name;
3    private double age;
4    public Animal(String name, double age) {
5       this.name= name;
6       this.age= age;
7    }
8    public abstract void makeSound();
9 
10    @Override
11    public String toString() {
12       return name+" is "+age+" years old";
13    }
14 }
15 public abstract class Pet extends Animal {
16    private String owner;
17    public Pet(String name, double age, String owner) {
18       super(name,age);
19       this.owner= owner;
20    }
21    @Override
22    public String toString() {
23       return super.toString()+" (Owner: "+owner+")";
24    }
25 }
26 public class Dog extends Pet {
27    public Dog(String name, double age, String owner) {
28       super(name,age,owner);
29    }
30    public void makeSound() {
31       System.out.println("Woof, woof!");
32    }
33 }
34 public class Cat extends Pet {
35    public Cat(String name, double age, String owner) {
36       super(name,age,owner);
37    }
38    public void makeSound() {
39       System.out.println("Meow");
40    }
41 }
42 public class Driver {
43    public static void main(String[] args) {
44       Animal a,b;
45       //a= new Animal(); // no can do
46       //a= new Pet();    // no can do
47       a= new Dog("Fido",8.5,"Mary");
48       b= new Cat("Tom",4.0,"Mary");
49       System.out.println(a);
50       a.makeSound();
51       System.out.println(b);
52       b.makeSound();
53    }
54 }
```

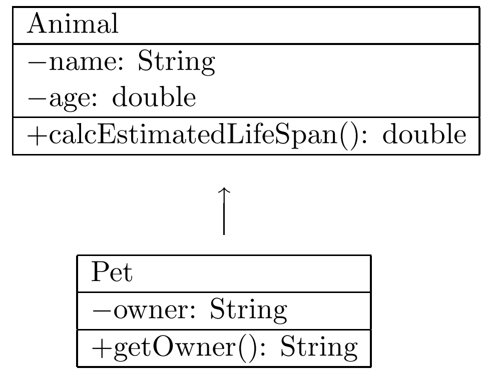
Expected output:

```
Fido is 8.5 years old (Owner: Mary)
Woof, woof!
Tom is 4.0 years old (Owner: Mary)
Meow
```

An *abstract class* is one that has not been fully defined. A class is made abstract by including the `abstract` keyword in the header (line 1). Typically an abstract class has one or more abstract methods (line 8). In this case we want every Animal to define a `makeSound()` method ... however we don't know what sound needs to be filled in by each kind of animal. On line 15 we inherit from `Animal` to create `Pet`. Since we still don't know how to define `makeSound()` for a `Pet` we must make `Pet` abstract as well. The `Dog` class inherits from the abstract class `Pet` but it provides an actual definition for the previously abstract `makeSound()` method. Therefore, `Dog` does not need to be abstract. Abstract classes cannot be instantiated which is why lines 45-46 will not compile if included. Although you cannot create instances of abstract classes they are useful as base classes for building an inheritance hierarchy. NOTE: The variables `a` and `b` are declared to be `Animal` on line 44 (which is abstract), but thanks to polymorphism we can instantiate them with non-abstract `Dog` and `Cat` objects (lines 47–48).

```java
public class Animal {
   private String name;
   private double age;
   public Animal(String name, double age) {
      this.name= name;
      this.age= age;
   }
   public double calcEstimatedLifeSpan() {
      return age*2.0;
   }
   public double calcEstimatedHeartRate() {
      return -20.0*calcEstimatedLifeSpan()+460;
   }
   @Override
   public String toString() {
      return name+" is "+age+" years old";
   }
}
public class Pet extends Animal {
   private String owner;
   public Pet(String name, double age, String owner) {
      super(name,age);
      this.owner= owner;
   }
   public String getOwner() {
      return owner;
   }
   @Override
   public String toString() {
      return super.toString()+" (Owner: "+owner+")";
   }
}
```

Here is a UML representation of these classes.



UML (Unified Modeling Language) is a widely used notation for representing class hierarchies in object-oriented programs. The purpose of the diagrams is to succinctly represent the structure of a program. These are the conventions we will follow in this course:

- Each class is represented as a rectangle with three parts: (1) the name of the class, (2) a list of attributes, and (3) a list of methods. Each of these parts is separated from other parts by horizontal lines.
- We use a leading − to mean "private", a leading + to mean "public", and a leading # to mean "protected".
- Attributes include an access modifier followed by the name of the attribute followed by a colon followed by its type.
- Methods include an access modifier followed by the name of the method followed by it parameter list (including types) followed by a colon followed by its return type.
- Classes that are related via inheritance have an arrow pointing from the inheritING class to its parent.

Access Modifiers

```
1  public class Animal {
2     private String name;
3     protected String owner;
4     double age;
5     public double lifeSpan;
6
7     public Animal(String name, double age, String owner) {
8        this.name= name;
9        this.age= age;
10       this.owner= owner;
11       lifeSpan= age*2.0;
12    }
13 }
14 public class Pet extends Animal {
15    public Pet(String name, double age, String owner) {
16       super(name,age,owner);
17    }
18
19    public void show() {
20       //System.out.println("Name : "+name); // nope!
21       System.out.println("Owner: "+owner);
22       System.out.println("Age : "+age);
23       System.out.println("Span : "+lifeSpan);
24    }
25 }
26 public class Driver {
27    public static void main(String[] args) {
28       Pet a;
29       a= new Pet("Fido",8.5,"Mary");
30       a.show();
31       //System.out.println(a.name); // nope
32       System.out.println(a.owner);
33       System.out.println(a.age);
34       System.out.println(a.lifeSpan);
35    }
36 }
```

Expected output:

```
Owner: Mary
Age  : 8.5
Span : 17.0
Mary
8.5
17.0
```

NOTE: Lines 2–5 demonstrate the four possible access modifiers: private, protected, no-modifier, and public.

	this class	any class in same package	subclass in another package	any class
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
private	✓	✗	✗	✗

Suppose an attribute (or method) `x` has an access modifier "protected". Then `x` can be accessed from within the class in which it is defined. It can also be accessed by any class in the same package and by subclasses that are defined in other packages. It would not, however, be accessible by just any class in another package. Java, by default, considers classes in the same directory to be in the same package. So, in this example, the attributes `owner`, `age`, and `lifeSpan` are all accessible both in the driver and in the inherited class `Pet`. General guidance for access modifiers: (1)make methods public and (2) make attributes private (unless you need to directly access them in inherited classes in which case make them protected).

Comparable

```java
1 public class Animal implements Comparable<Animal>{
2    private int id;
3    private String name;
4    private double age;
5
6    public Animal(String name, double age, int id) {
7       this.name= name;
8       this.age= age;
9       this.id= id;
10   }
11
12   @Override
13   public int compareTo(Animal other) {
14      if (this.age > other.age) {
15         return 1;
16      }
17      else if (this.age < other.age) {
18         return -1;
19      }
20      return 0;
21      // return this.name.compareTo(other.name);
22      // return this.id - other.id;
23   }
24
25   @Override
26   public String toString() {
27      return name;
28   }
29 }
30 public class Driver {
31    public static void main(String[] args) {
32       int ans;
33       String x,y;
34       x= "hello";
35       y= "there";
36       ans= x.compareTo(y);
37       System.out.println(ans);
38       if (y.compareTo(x) > 0) {
39          System.out.println(y+" is greater than "+x);
40       }
41
42       Animal a,b;
43       a= new Animal("Max",4.0,20);
44       b= new Animal("Fido",8.5,10);
45       ans= a.compareTo(b);
46       System.out.println(ans);
47       if (a.compareTo(b) < 0) {
48          System.out.println(a+ " is less than "+b);
49       }
50    }
51 }
```

Expected output:

```
-12
there is greater than hello
-1
Max is less than Fido
```

The built-in `String` class is an example of a class that *implements the Comparable interface* by defining a `compareTo()` method that returns an integer. In the example line 36 compares `x` with `y` and the answer comes back as a negative 12. The negative result suggests $x < y$. A positive result would mean $x > y$ and a zero result would mean they are equal. If we need to compare objects of a custom-built class it is good form to follow these same conventions. To formally announce our class implements the Comparable interface we use the notation shown on line 1. Once we've done that we have to write a method called `compareTo` that accepts an object of the specified type as a parameter and returns an int. In the case of our `Animal` object it is not obvious what it may mean to "compare two animals". The example assumes we compare based on age. Lines 21 and 21 provide examples of how the method might have been written if we based comparison on the `name` attribute or `id` attribute, respectively.

Equals

```java
public class Animal {
   private int id;
   private String name;
   private double age;

   public Animal(String name, double age, int id) {
      this.name= name;
      this.age= age;
      this.id= id;
   }

   @Override
   public boolean equals(Object other) {
      if (this == other) {
         return true;
      }
      if (!(other instanceof Animal)) {
         return false;
      }
      Animal a= (Animal) other;
      return this.name.equals(a.name);
   }

   @Override
   public int hashCode() {
      return name.hashCode();
   }

   @Override
   public String toString() {
      return name+"("+age+"/"+id+")";
   }
}

public class Driver {
   public static void main(String[] args) {
      Animal a,b,c;
      a= new Animal("Max",4.0,20);
      b= new Animal("Fido",8.5,10);
      c= new Animal("Max",1.0,30);
      if (a.equals(b)) {
         System.out.println(a+ " equals "+b);
      }
      if (a.equals(c)) {
         System.out.println(a+ " equals "+c);
      }
      if (a == c) {
         System.out.println(a+ " == "+c);
      }
   }
}
```

Expected output:

```
Max(4.0/20) equals Max(1.0/30)
```

Since every class in Java implicitly inherits from the `Object` class, each class is endowed with a `.equals()` method that return true or false if the parameter is “equal to” the current object. The default behavior of this method is to compare the addresses of the objects rather than their contents. The `String` class automatically overrides this behavior to do a meaningful lexicographic comparison of the characters in the string. When we write custom classes we need to override this method to give it useful behavior if we need to compare objects for equality. Java has an unusual requirement that when overriding `.equals()` we must also override `hashCode()` (so that “equal” elements will return the same hashCode value). In our example we define `Animal` objects to be “equal” if they have the same name. So, we relegate our definitions of `equals()` and `hashCode()` to the `String` class.

Recursion

```java
public class Recursion
{
   public static void main(String [] args)
   {
      System.out.println("4-factorial is: "+nfact(4));

      System.out.println("Moving 4 discs from L to R");
      tower("L","R","M",4);
   }

   public static void tower(String from, String to, String using, int n)
   {
      if (n==1)
         System.out.println("Move from "+from+" to "+to);
      else {
         tower(from,using,to,n-1);
         tower(from,to,using,1);
         tower(using,to,from,n-1);
      }
   }

   public static int nfact(int n) {
      if (n==0) {
         return 1;
      }
      return n*nfact(n-1);
   }
}
```

Expected output:

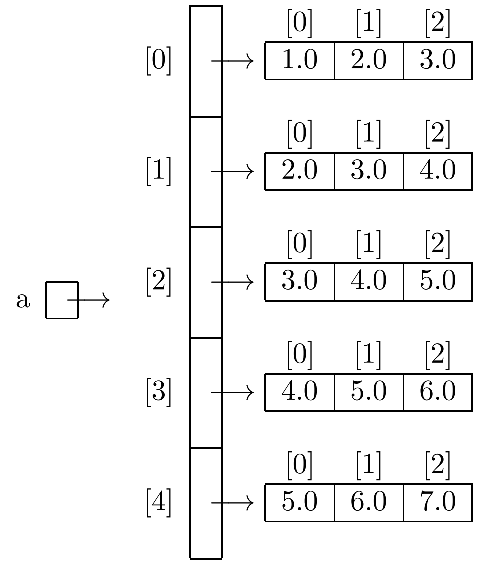
```
4-factorial is: 24
Moving 4 discs from L to R
Move from L to M
Move from L to R
Move from M to R
Move from L to M
Move from R to L
Move from R to M
Move from L to M
Move from L to R
Move from M to R
Move from M to L
Move from R to L
Move from M to R
Move from L to M
Move from L to R
Move from M to R
```

An *recursive* method is one that calls itself. Recursion is a form of looping and is useful for solving a variety of problems. When writing a recursive method you need to:

- Using words, provide a recursive solution to problem. (e.g., "n-factorial is n multiplied by $n-1$-factorial).
- Identify a base condition, which when met, ends the recursion. (e.g., if $n = 0$ then the answer is 1).
- As you write your code, in faith, call the already-working method as part of your solution.

This example demonstrates two unrelated recursive method (one that returns a value and one that displays a result).

```java
1  public class TwoD
2  {
3     public static void main(String [] args)
4     {
5        int i,j;
6        double [][] a;
7
8        a= new double[5][3];
9
10       for (i=0; i<5; i++) {
11          for (j=0; j<3; j++) {
12             a[i][j]= i+j+1.0;
13          }
14       }
15
16       display(a);
17    }
18
19    public static void display(double [][] a)
20    {
21       int i,j;
22
23       for (i=0; i<a.length; i++)
24       {
25          for (j=0; j<a[i].length; j++) {
26             System.out.printf("%6.2f",a[i][j]);
27          }
28          System.out.println();
29       }
30    }
31 }
```



Expected output:

```
  1.00  2.00  3.00
  2.00  3.00  4.00
  3.00  4.00  5.00
  4.00  5.00  6.00
  5.00  6.00  7.00
```

An *two-dimensional array*, is perhaps most accurately visualized in Java as an array of arrays. The variable `a` declared on line 6 is specified as a reference to a two-dimensional array of `double` values. In the diagram line 6 creates the small box labeled `a`. To actually get the values we must use `new` (demonstrated on line 8). From the diagram we can see that `a[0]` is a reference to a one-dimensional array have three elements. To get to a single piece of the one dimensional array pointed to by `a[0]` we need to index it. So, `a[0][2]` is a double value (3.0). To access every double value in a two dimensional array we use nested loops. NOTE: Java supports even high dimensions than two.

Static

```
1 public class Animal {
2    private static String name;
3    private double age;
4 
5    public Animal(String name, double age) {
6       this.name= name;
7       this.age= age;
8    }
9 
10   public static String getName() {
11      return name;
12   }
13 
14   @Override
15   public String toString() {
16      return name+" is "+age+" years old";
17   }
18 }
19 
20 public class Driver {
21   public static void main(String[] args) {
22      Animal a,b;
23      a= new Animal("Fido",8.5);
24      b= new Animal("Leon",2.5);
25      System.out.println(a);
26      System.out.println(b);
27      System.out.println(a.getName());
28      System.out.println(Animal.getName());
29   }
30 }
```

Expected output:

```
Leon is 8.5 years old
Leon is 2.5 years old
Leon
Leon
```

When declaring a variable or a method in a class you have the option of making it static or not. A *static* method or variable is one that is shared among all the instances of a class. If a method or variable is *not* static then each instance of the class gets it own copy. In this example, the `name` attribute is declared to be static and we have two instances of the `Animal` class: `a` and `b`. When `a` is created on line 23 its name is set to “Fido” and age to 8.5. When `b` is created on line 24 its name is set to “Leon” and age to 2.5. Since the `name` attribute is static (i.e., one copy shared among all instances) the name “Leon” overwrites “Fido”. Since age is not static, each instance gets its own copy of age. This is reflected in the output. The `getName()` method is declared to be static. Therefore it can be invoked as expected on line 27 and also without using an particular instance as on line 28. NOTE: A static method cannot refer to a non-static variable. When deciding whether something should be static or not the rule of thumb is: “If it is possible to make it static then do so.” In our example we really need every instance of the `Animal` class to have its own name so the name attribute should not be static.

Reading Text Files: BufferedReader

```java
   // requires some imports:
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public static double sumFile(String filename) throws IOException {
   BufferedReader numFile;
   double total=0.0;
   String str;

   numFile= new BufferedReader(new FileReader(filename));
   while ((str= numFile.readLine())!=null) {
      total= total + Double.parseDouble(str);
   }
   numFile.close();
   return total;
}
```

Suppose datafile looks like this:

```
1.5
10.2
2.3
5.2
```

Expected return value: 19.2

The `BufferedReader` class can be used to read data from files. It runs much faster than using `Scanner` but has the disadvantage that it can only read a single line at a time as a string. So, lines from the file may need to be broken apart and converted to various types. The open command can generate an `IOException` which must be handled. This example accepts the name of a text file that presumably contains one number per line of the file. We then read one line at a time and put it into the string `str`. The value comes back a `null` we know the read failed and it is time to stop reading. We convert the string to a double value and then add it to our running total.

```java
   // requires some imports:
import java.io.FileNotFoundException;
import java.io.PrintStream;

public static void saveData(String filename) throws
    FileNotFoundException
{
   PrintStream f= new PrintStream(filename);
   double y= 3.8234;

   f.print("This is so fun!");
   f.print("How are you?\n");
   f.println("What do we do now?");
   f.printf("I am am %8.1f feet tall\n",y);
   f.close();
}
```

Expected output (in junk.txt):

```
This is so fun!How are you?
What do we do now?
I am    3.8 feet tall
```

To write to a text file use the `PrintStream` class which is the same type of object as `System.out`. So, you can use `.print()`, `.println()`, and `.printf()` to write to a file just as if you were displaying it to the screen. When you open the file by creating the `PrintStream` object it will create a new file if it doesn't exist; if the file did exist it is destroyed!

Try-Catch

```java

int num;
Scanner kb= new Scanner(System.in);

try {
   System.out.print("Enter number: ");
   num= kb.nextInt();
   System.out.println("You entered: "+num);
}
catch (InputMismatchException e) {
   System.out.println("Must enter an integer");
   System.out.println(e);
}
```

Expected output if user enters 7.

```
Enter number: 7
You entered: 7
```

Expected output if user enters “frog”.

```
Enter number: frog
Must be an integer
java.util.InputMismatchException
```

A *try-catch block* can be used to prevent a program from crashing on an error condition. In this example, if the user types a `double` or `String` instead of an `int` the program would normally crash with an `InputMismatchException`. With the try-catch block we ask the program to try the statements listed. Then we catch exceptions that occur using one or more catch blocks. If the listed exception occurs control is transferred to the catch block rather than crashing. NOTE: This is not a looping structure! If you want your program to go back and let the user try entering the number again then you need to put the try-catch block inside a loop. NOTE: try-catch can be used to handle code that works with files so you no longer need to announce `throws FileNotFoundException` in the header.

```
public class ListNode {
   double data;  // this will be a list of double values
   ListNode next; // reference to the next element
}

ListNode head,p;

head= new ListNode();  // create first element
head.data= 1.5;
p= head;     // head and both referring to first element
p.next= new ListNode(); // create second element
p= p.next;   // head stays put and p moves to second elem
p.data= 2.6;
p.next= new ListNode(); // create third element
p= p.next;            // p moves to third elem
p.data= 5.0;
p.next= null;         // mark this as last element

// To traverse a list we use a temporary pointer to
// start at head and then move through the list to the end.
p= head;
while (p!=null) {
   System.out.println(p.data);
   p= p.next;
}
```

Expected output:

```
1.5
2.6
5.0
```

Working with a linked list doesn't require any special commands beyond those already documented in this command sheet. A *linked list* is a series of data elements in which each element keeps track of the location of the next element. There are several consequences of this approach:

- Linked lists are sequential in nature
- Each element needs to store data *and* a pointer/arrow/reference to the next element in the list.
- We need to keep track of where the beginning of the list is. We will use a special variable called `head` for this purpose.
- The last element needs to contain a marker indicating that there is no next element. We will store the special value `null` in the last node's next pointer.

In this example we create three elements that are chained together and then print them out. The variable `head` stays fixed at the beginning of the list and the variable `p` is used to move through the list.