

Intro to Java

Final Review- Spring 2010

Possible Exam Format

- True/False
- Code Tracing
 - Recursion Tracing
- Complete Code
- Write Code
- Writing Methods/Classes
- Drawing Arrays
- Finding Errors
- Is this statement valid (legal)?
- Javadocs
- Write overriding/overloading methods.
- General Multiple Choice
- Class/Subclass Relationship

Past Material

- Focus is on new material but you cannot forget material from before the midterm
- Know:
 - Data types
 - Loops
 - while
 - do/while
 - for
 - Decision statements (if, else, else if, switch)
 - Operators
 - Input/Output (keyboard)



Commonly Used Classes

Math Class

- `java.lang.Math`
- Common Methods-
 - `ceil(double a)`- rounds up
 - `floor(double a)`- rounds down
 - `min(a,b)`
 - `max(a,b)`
 - `pow(double a, double b)`- a^b

String Class

- Should know how to manipulate Strings
- Common Methods
 - `charAt(int index)`
 - `compareTo(String s)`
 - `equals(String s)/ equalsIgnoreCase(String s)`
 - `indexOf(String s)/ indexOf(char c)`
 - `length()`
 - `replace(String s, String newS)`
 - `substring(int start)/substring(int start, int end)`
 - `trim()`
 - `toUpperCase()/ toLowerCase()`

Random Class

- `java.util.Random`
- Know how to use Random class
 - ex. `Random rand = new Random();`
`int x=rand.nextInt();`
 - Can get random integer, double, float or boolean
- Look at `MyRandom.java`



Object Oriented Programming

Overview of OOP

- What is a class?
 - Definition or “blueprint” used to create objects
 - Set of related data making up the object’s current state (data members)
 - A set of behaviors (methods)
- What is an object?
 - Instance of a class
- What are the benefits to using classes and objects?
 - Organization, code reuse, encapsulation
- What is encapsulation?
 - Hiding or “protecting” the object’s data by providing limited access

Classes

- Consists of:
 - A list of variables or “data members”
 - A constructor(s)
 - A list of methods
 - (Can be either class or instance type)
- Also has constructor in order to create an instance of a class (an object)
- Essentially a blueprint for creating and modifying an object
- Examples to look at: Card.java, Course.java, Deck.java, Time.java, etc...

Access

- Different modifiers give variables and methods different amounts of access
 - Public- Anyone can access it
 - Protected- Package access = any file in same folder
 - Private- Only the class can access it
- What's the point?
 - Keeping data protected or private stops just anyone from modifying it (by accident or by choice)

Modifier	Class	Package	World
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

Class vs. Instance Variables

- Instance variables- data members that are associated with each instance of the class
 - Ex. Each car has it's own color or year
- Class Variables- data members that are associated with the entire class as a whole (applies to all instances of the class) = static
 - Ex. Let's say I want to keep track of the number of all the cars made so I have an integer variable called carCount. Every car has the same carCount since it is an attribute of the Car class and does not vary by each Car object.

Variable Scope

- Where can the variable be used and accessed
- A variable's scope depends on where it was declared (not where it is created/instantiated)
- When the variable is out of its scope, it is destroyed or unreachable
- Local Variable- Created and used locally inside of a method; once the method call is over, the variable “dies”
- Instance Variable- As long as the instance of that class is still alive, the instance variable inside lives; when the object is destroyed, so is the instance variable
- Class Variable- Class variables can be accessed anywhere in the class



Global Variables

- A variable that is accessible in every scope
- Provides too much access, discouraged from use due to the ease of modification (no encapsulation)

Default Constructor- automatically assigns all information to data members, takes no parameters

```
public class Car
{
    private String make, model, color;
    private int year;
```

Data members- private access encapsulates data

```
public Car(){
    make="Ford";
    model="F-150";
    color="red";
    year=2009;
}
```

```
public Car(String mk, String mo, String c, int y){
    make=mk;
    model=mo;
    color=c;
    year=y;
}
```

Mutator Method- Changes some data member of the object and alters its overall state; takes a parameter

```
public String toString(){
    return color + " " + year + " " + make + " " + model;
}
```

```
public void setColor(String newColor){
    color=newColor;
}
```

```
public String getColor(){
    return color;
}
```

Accessor Method- Changes nothing about the method but allows access to private data member and returns the information

Constructor

- Needed to create an instance of the class
- Used to instantiate data members when creating a new instance of the class
- Can have many constructors
- Format: `public class ClassName(){ ... }`
- No return type!
- Usually has at least default constructor which takes no parameters
- Constructors that take parameters use them to assign initial values to the data members

Methods

- Method Header
 - `public/private (static or not) returnType Name(parameters)`
- Can return up to one thing (void if returning nothing)
- Cannot nest method definitions
- Some kinds of methods are : accessors, mutators, helpers
- Be ready to write methods
- Constructors and methods are not the same
- Method Overloading- Multiple methods with the same name that differ in parameters (number of/type)

Accessor Methods

- Allows access to the private data encapsulated by the object and returns that information
- Used to find the state of an object, or what its data member is
- For example, an accessor in the Color program would be `.getBlue()`, a method to return the blue value of a Color object
- Usually has a non-void return type and takes no parameters

Mutator Methods

- Allows access and modification of private data encapsulated by an object
- Used to modify and change an objects data member
- For example, a mutator in the Color program would be `.setBlue(int b)`, a method to change the blue value of a Color object
- Usually has a void return type and takes some parameters (in order to change the state of the object)

Return Statement

- Used to get some sort of data from the method
- The statement comes at some point in the method and follows the pattern: return (some data type);
- Commonly used to return int, boolean, double, String and other objects
- Can have a return type of void meaning no return statement is needed
- Can have multiple return statements – control goes back to method call immediately!
- Can even call other methods in return statement (recursion)

Static Methods

- When to use static methods?
 - When accessing static (class) variables or calling other static methods
 - When no instance of the class is created
- Cannot access an instance variable from a static method; can only call another instance method with an explicit object
 - Error: Non-static ... cannot be referenced from a static context
- `ClassName.methodName(parameters)`
- `public/private static returnType name(parameters)`

```
public static int randomInt(Random rand, int low, int high)
{
    if(low > high)
    {
        int temp=low;
        low=high;
        high=temp;
    }
    return low + rand.nextInt(high - low + 1);
}
```

Keyword: this

- Using the this keyword refers to the current instance of the class
- Helps prevent confusion if:
 - There is another object of the same type being worked with
 - There is another variable with the same name (usually passed in as a parameter)
- Example:
 - ```
public void setGrowthRate(double growthRate)
{
 this.growthRate = growthRate;
}
```

# Reference vs. Value

- Primitive data type variables are **passed by value** meaning a copy of the variable is created when being passed. So any changes made to the variable in a method won't effect the original.
  - Examples include int, boolean, double, etc...
- When an object is created, it is a **reference variable**, because it stores the address of where the object is stored in memory. So if a reference variable is passed and modified in a method, it will be changed in the driver as well.
  - Examples include passing an array or any user defined class/object (anything requiring the =new objectName() to create)

# Driver

- A class that will create and use instances of other classes (objects)
- Usually in separate files than the classes they “drive”
- Creates objects and uses methods to modify them
- Needs to be in same directory of the classes it creates instances of
- Examples: CardTest.java, CustomerTest.java, testTime.java, etc...

```
public class carTest
```

```
{
```

```
 public static void main(String[] args)
```

```
 {
```

```
 Car c=new Car();
```

```
 Car c2=new Car("Honda", "Civic", "Blue", 1985);
```

```
 System.out.println("Car c is a " + c);
```

```
 System.out.println("Car c2's color is " + c2.getColor());
```

```
 String c2OrigColor=c2.getColor();
```

```
 c2.setColor("orange");
```

```
 System.out.println("Car c2's old color is " + c2OrigColor + " and new
color is " +c2.getColor());
```

```
 }
```

```
}
```

Creates a new instance of  
the Car class, a default  
Car object

Creates a new instance of  
the Car class, but with  
user defined parameters

Uses an accessor method  
to find the Car's color (a  
private data member)

Uses a mutator method  
to modify the color of the  
Car, c2

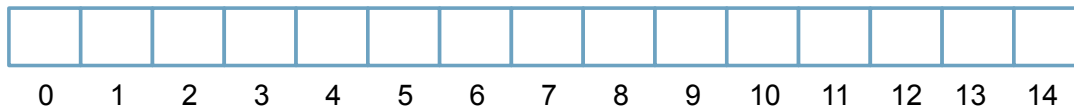


# Arrays

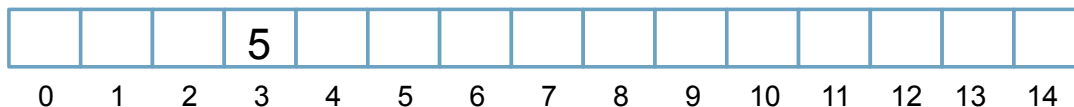
# Arrays

- A data structure used to hold a collection of elements of the same type
- Each array has a set size and type that is defined when it is created
- Access an element in an array by using [index]
- Declaration: `type [] name;`
- Creation: `name = new type[size];`

```
int [] numArray= new int[15];
```



```
numArray[3]= 5;
```



# Arrays Continued

- Can declare and create an array at the same time
  - `type [] name = new type[size];`
- In order to print a full array must manually call each element (loop)
- Can initialize all the elements in the array during its creation:
  - `double [] temp= {98.7, 67.5, 80.3, 74.3, 40.2};`
- To initialize (large) array to same value use loop
- To get size of array: `arrayName.length`
  - This only returns capacity of array not how many elements are filled
- Know how to copy and manipulate arrays (look at assignment with ColorCollection)

## 2 Dimensional Arrays

- Similar to arrays except can have any number of rows and columns
- Array of references to arrays
- Syntax is similar to array except with [][]
  - `type [][] name = new type[rowSize][colSize];`
- Access using `[row#][column#]`
- `arrayName.length` returns number of rows
- To get number of columns use: `arrayName[x].length`
- Initialize
  - `int [][] x= { {5, 2, 3}, {6, 7, 9} };    ⇒    

|   |   |   |
|---|---|---|
| 5 | 2 | 3 |
| 6 | 7 | 9 |`
- `x[0]` refers to the first row since it is really a reference to an array

# Arrays of Objects

- Array of objects = Array of references
- Each element in the array holds a reference to location of the object in memory, not the actual object
- Need to instantiate both the array and the objects in the array
- Initial type for each object is null (null pointer error)
- Still must treat objects normally (use methods to access/change data members)
- Example
  - `Car [] carArr = new Car[4];`  
`carArr[0]= new Car();`  
`carArr[1] = new Car("Nissan", "Sentra", "Green", 1992);`



# Searching and Sorting

# Searching

- **Sequential Search (SortSearch.java)**
  - Good for shorter searches
    - Time is proportional to number of elements/length of array-  $O(n)$
  - Goes through the each element until the search term is found or array is finished
  - ```
for(int i=0; i<arr.length; i++)  
{  
    if(searchTerm==arr[i])  
        Do something and break  
}
```
- **Binary Search**
 - Better for longer searches
 - Time is proportional to logarithm of array length (base 2)- $O(\log_2 n)$
 - Array must be sorted
 - Looks at the middle of the array, if search term is larger, searches the right side else if the term is smaller searches the left side

Searching Example

- Sequential Search

3	6	12	14	0	9	5	7	13	1	8	2	4	11	10
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑ Checks and sees that `arr[0] != 7`, continues ...

3	6	12	14	0	9	5	7	13	1	8	2	4	11	10
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑ Checks and sees that `arr[1] != 7`, continues ...

UNTIL it reaches 7 at `arr[7]` and then breaks out of the loop

3	6	12	14	0	9	5	7	13	1	8	2	4	11	10
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Searching for: 7

```
int searchTerm=7;
int found=-1;

for(int i=0; i<arr.length; i++)
{
    if(arr[i]==searchTerm)
    {
        found=i;
        break;
    }
}

if(found>-1)
    System.out.println(searchTerm +
        " found at element " + found);
else
    System.out.println(searchTerm +
        " was not found");
```

If instead searching for say 25 would reach end of array and loop without finding it...

Searching Example

- Binary Search

Searching for: 55

10	21	27	36	54	55	67	81	89	110	125	212	242	305	489
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Checks and sees that $55 < arr[7]$ (middle), checks left ...

10	21	27	36	54	55	67	81	89	110	125	212	242	305	489
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Checks and sees that $55 > arr[3]$ (middle), checks right ...

10	21	27	36	54	55	67	81	89	110	125	212	242	305	489
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Checks and sees that $55 > arr[3]$ (middle), checks right ...

If instead searching for say 65 would continue searching until subarray size was only 1 and would not find 65...

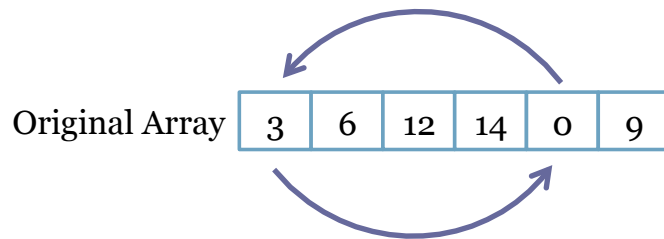
Look at the **CODE!** It can be found in `Sorter.java` or in section 10.7 of textbook.

Sorting

- Know how to implement a sort
- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Examples: `Sorter.java` (in `Arrays`: `Sort.java`, `Sort2.java`)

Selection Sort

- Look for the smallest element of the array and move it to the first position of the array (by swapping), then check the remaining array and find the next smallest element and move it to the 2nd position and etc... until the entire array is sorted.



After pass 1:

0	6	12	14	3	9
---	---	----	----	---	---

After pass 2:

0	3	12	14	6	9
---	---	----	----	---	---

After pass 3:

0	3	6	14	12	9
---	---	---	----	----	---

After pass 4:

0	3	6	9	12	14
---	---	---	---	----	----

After pass 5:

0	3	6	9	12	14
---	---	---	---	----	----

Final Array

0	3	6	9	12	14
---	---	---	---	----	----

```
int pass, index, min_index;  
String minvalue;
```

```
for (pass = 0; pass < max - 1; pass++)  
{
```

```
    min_index = pass;  
    minvalue = ra[pass];
```

```
    for (index = pass+1; index < max; index++)  
    {
```

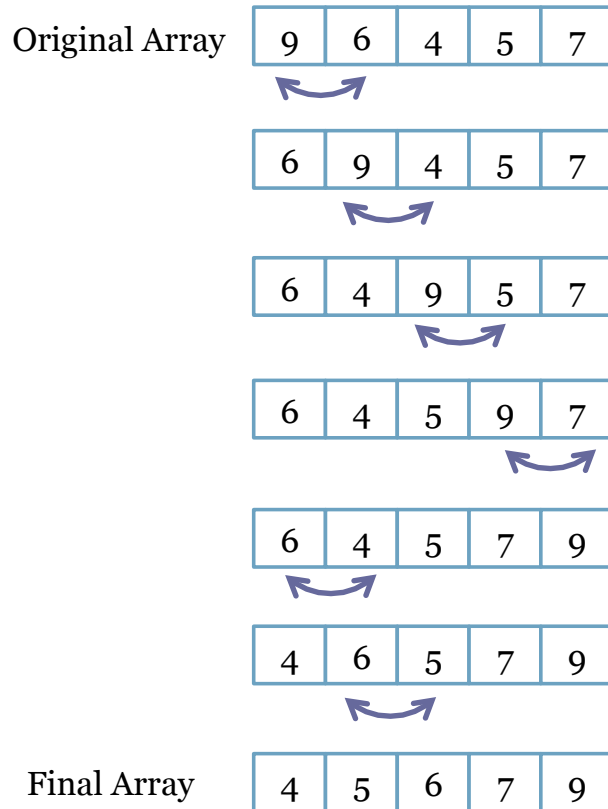
```
        if (ra[min_index].compareToIgnoreCase(ra[index]) > 0)  
        {  
            min_index = index;  
            minvalue = ra[min_index];  
        }  
    }
```

```
    ra[min_index] = ra[pass];  
    ra[pass] = minvalue;
```

```
}
```

Bubble Sort

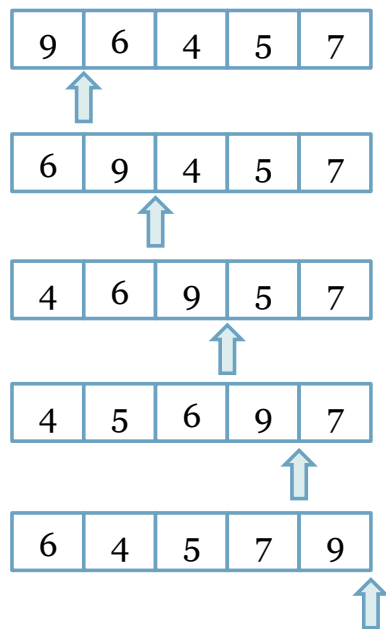
- Looks at each 2 elements in the array and sees if they are ordered, if they are not, will swap them and move to next 2 elements. It does this to the entire array until it is sorted.



```
String temp;
for (int size = max-1; size >= 1; size--)
{
    for (int i = 0; i < size; i++)
    {
        if (ra[i].compareToIgnoreCase(ra[i+1]) > 0)
        {
            temp = ra[i];
            ra[i] = ra[i+1];
            ra[i+1] = temp;
        }
    }
}
```

Insertion Sort

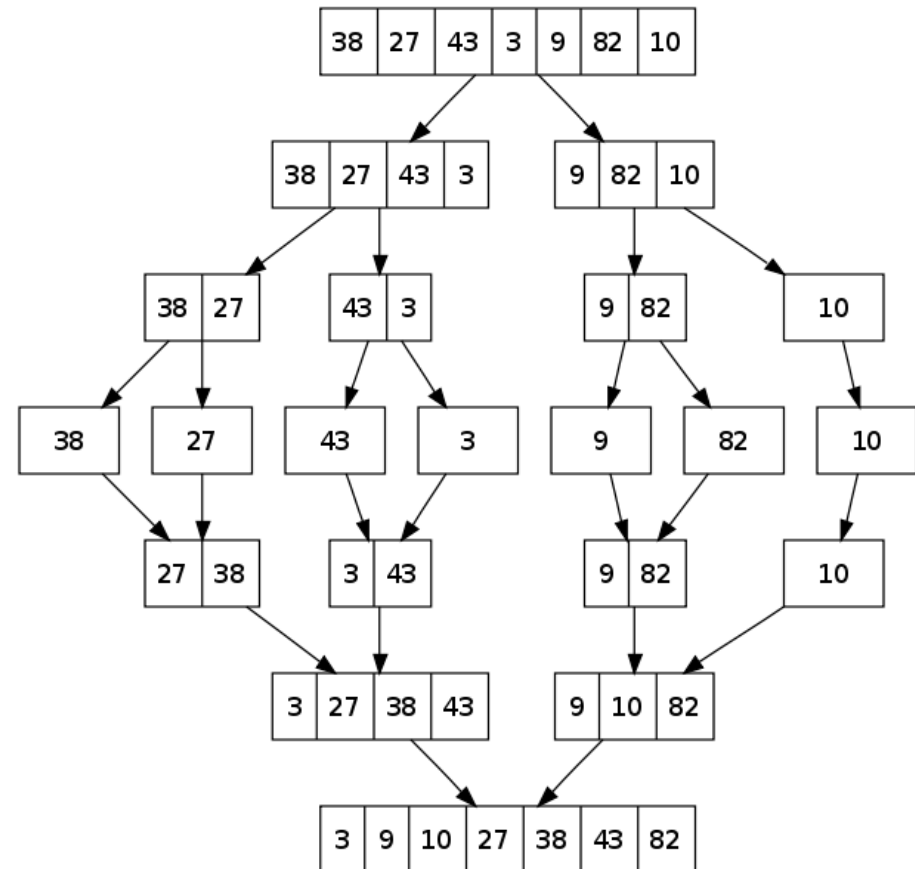
- Looks at each element in the array and inserts in somewhere to the left where the array is already sorted into its correct position.



- Everything to the left of the arrow should always stay sorted
- Look at element immediately following array and insert in its correct position to the left of the arrow and move all subsequent elements in the array down 1 spot

Merge Sort

- Splits the array continually until it is small enough to sort, sorts and merges back together, sorts and merges back together, etc... until array is full and sorted
- Code is given in Sorter.java
- Usually a recursive sort



Recursion

- When a method calls itself
- Indirect recursion uses several methods calling on each other
- Used to simplify a problem, need to identify a stopping condition or a “base case”
 - Base case- The case that will end the recursion
- Any problem that can be written recursively also can be written using loops
 - So why use recursion?
 - Can give a more straightforward and functional solution.

Recursion Continued

- Basic Structure
 - `if(something) //base case or stopping condition`
`return or end method`
 - `else //if not base, recurse`
`call the method again`
- Can be more complicated, such as the Meow hw problem that contained several cases
- Look over solution to the Meow program
- Know how to write a recursive method
- Know how to trace out a recursive method

Recursion Examples

- Factorial is a common example.

```
public int factorial(int n)
{
    if (n==1)           //base case
        return 1;
    else
        return (n*factorial(n-1));
}
```

- Power is also a common example but contains more conditions.

```
public double power(double x, int n)
{
    if (n > 1)
        return x * power(x, n - 1);
    else if (n < 0)
        return 1.0 / power(x, -n);
    else
        return x;           //base case
}
```

Inheritance

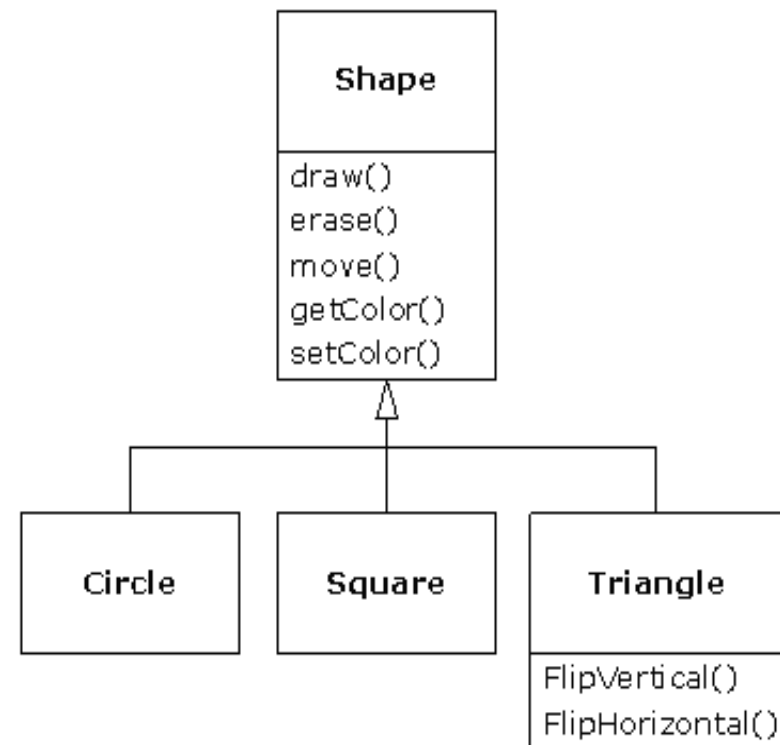
- Base Class (Superclass) and Subclass
 - Derived Class inherits all the data members and methods from the Superclass
- Know the terminology: Ancestor and Descendant
- The Subclass can do everything the Superclass can do and more... (through defining its own variables and methods)
- In order for a class to inherit from the Superclass use the keyword **extends**
 - `public class Teacher extends Employee{ ...}`
 - In this example, the Teacher class inherits everything from the Employee class.

Inheritance- super

- Can override or redefine methods that are defined in the Superclass in the Subclass
- Use `super()` to call the Base (superclass) constructor
- Also use `super.methodName()` to call the base class method
- Overriden methods must have same return type

Inheritance Example

- Shape is the superclass
- Circle, Square and Triangle are all types of shapes; therefore they extend Shape and gain all of its data members and methods
- They also define new methods and data members.
- Remember a Triangle is a Shape but a Shape is not always a Triangle.
 - So you can make a Shape object equal to a type Triangle but not the other way around (compile error)
- Warning: This is not the same as Professor Selinski's shape example



Polymorphism

- The ability for a particular method call to perform different operations at different times.
- What does this actually mean?
 - This occurs when you have a reference variable that refers to different types of objects during the programs execution.
 - Example:
 - If you create a Shape reference variable, you can make this a Circle, Square or Triangle. So depending on what subtype it is, it can call a different method.

Inheritance Cont.

- Often override toString(), compareTo() and equals()

- Example: (for a Square class)

- public String toString()
{
 return (side + " by " + side + " square");
}

- Example: (for a Circle class)

- public int compareTo(Circle c)
{
 return (this.area() - c.area());
}

Abstract Class (not covered)

- Methods that aren't appropriate for being declared in the superclass but are for the subclass. You are telling the compiler that the subclasses will define this method.
 - Abstract methods means the class must be abstract too
 - Means that you plan to overwrite the abstract classes methods in later descendant classes
- Abstract methods must have just a method header with the word abstract in it and ended by a semicolon.
 - `public abstract double area();`
 - This method was not appropriate to define in shape, since the areas are all calculated differently for each shape but should be defined for each type of shape.
- Since there are partially defined methods in an abstract class, you cannot create an instance of that class
- You use the keyword **extends** with an abstract class.

Abstract Class Example

```
public abstract class Player
{
    String name;
    int points;
    public abstract void move();

    public Player(String n)
    {
        name=n;
        points=0;
    }
    public void display()
    {
        System.out.println(name + " has " + points + " points.");
    }
}
```

```
public class Human extends Player
{
    public Human(String n)
    {
        super(n);
    }
    public void move()
    {
        .... update points
    }
}
```

Interface (covered – know this)

- Interfaces take abstract classes as step further in that every method is just a method header. In essence every method is abstract in an interface.
- What is an interface?
 - Basically a skeleton for other classes
 - Tell what methods and variables the class must implement
 - Can never have an instance of an interface.
- Uses the keyword **implements**
- Example: Professor Selinski's Shape is an interface.

Interface Example

```
public interface Animal
{
    String name, color;
    double weight, height;
    void grow();
    void display();
}
```

```
public class Giraffe implements Animal
{
    public Giraffe(String n, String c, double w, double h)
    {
        name=n;
        color=c;
        weight=w;
        height=h;
    }
    public void grow()
    {
        weight +=10.0;
        height+=5.0;
    }
    public void display ()
    {
        System.out.println(name + " is a " + color + " giraffe that is " height + " inches and " + weight + "
pounds.");
    }
}
```

Inheritance and Polymorphism

- Know how interfaces, abstract classes, inheritance and polymorphism work.
- Go over Professor Selinski's code examples as well as homework (Employee.java and etc..) and Chapter 13 of the textbook.
- Remember that it is possible to both extend a class and implement an interface (must be done in that order).
- Have a good understanding of the relationships between the super and sub class.
- Know how to write an interface and super/subclasses.
 - Including method overriding.

Exception Handling

- Know basic terminology (throw, try/catch)
- If you are performing an operation that can error, put the code in a try statement and catch the possible error
- Can have several catch statements for each type of error

```
try{
    some code...
}catch(...Exception)
{ ....}
catch(...Exception)
{.....}
```

- The some code... must be code that has possibility of throwing some exception.
- There are several exceptions built in into Java.
- You can also write your own exception class.

Exception Example

- Also look at lab website and Week 13 for example code on exception handling and her example code in folder Exceptions.

```
try{
    System.out.println("Enter name of file you want to read from:");
    filename=readKeyboard.nextLine();
    Scanner fromFile= new Scanner(new FileReader(filename));
    ....
}catch(FileNotFoundException e){
    System.out.println( "Error. That file does not exist.");
}
```