Recap

- A class is a template from which objects can be created.
  - Can have a number of instances
- An object contains
  - State (attributes, instance variables)
  - Behavior (methods, messages)
- A program is a series of messages and responses that are send and received between objects
  - behavior is shared amongst objects

Constructor methods

- `private` limits direct access (via '.') to the class and its subclasses.
- A special method that has the same name as the class, used to create and initialize objects
- public methods provide read and write access to private members
  - getter methods
  - setter methods

```
public class Triangle {
  private int sideA;
  private int sideB;
  private int sideC;

  Triangle(int v1, int v2, int v3){
    this.sideA = v1;
    this.sideB = v2;
    this.sideC = v3;
  }
  public int getSideA(){
    return sideA;
  }
  public int getSideB(){
    return sideB;
  }
  public int getSideC(){
    return sideC;
  }
  public void setSideA(int newVal){
    sideA = newVal;
  }
  public void setSideB(int newVal){
    sideB = newVal;
  }
  public void setSideC(int newVal){
    sideC = newVal;
  }
}
```

Constructor methods (cont)

- What happens when a Java object is initialized:
  - data fields are set to `0`, `false` or `null`
  - data fields with initializers are set in the order they appear in the class definition
  - the constructor method body is executed
- `this` is a special keyword that denotes the current executing object.

```
public class Triangle {
  private int sideA;
  private int sideB;
  private int sideC;

  Triangle(int v1, int v2, int v3){
    this.sideA = v1;
    this.sideB = v2;
    this.sideC = v3;
  }

    ....
 }
```

Omitting the modifier defaults to `private`

The anatomy of a method

<modifiers> *<return-type>* *<method-name>*(*<arguments>*)

{

    <method-body>

}

- modifiers
  - `public`, `private`, `protected`, `final`, `static`
- return-type
  - `void`, `int`, String, Triangle... any Java Type
- method-name
  - getX, setY, toString, compareWith ... name starting with lower case
- arguments (comma separated list of <*type*> <*name*>)

The anatomy of a method (cont)

- method-body
  - a list of Java statements

```
public int area(){
    int result = 0;
    result = (sideA*sideB)/2;
    return result;
  }
```

- All different paths of execution inside the method should return a value of the type specified as the methods return type !

```
public int area(){
    int result = 0;
    if (sideA != 0 && sideB != 0){
       result = (sideA*sideB)/2;
       return result;
    } else {
       return result;
    }
 }
```

The anatomy of a method (cont)

- Local method variables have to be initialized. The following piece of code gives a compilation error

```
public int area(){
    int result;
    if (sideA != 0 && sideB != 0){
       result = (sideA*sideB)/2;
       return result;
    } else {
       return result;
    }
  }
```

`result` is not initialized in the `else` branch

The anatomy of a method (cont)

- Field shadowing

```
public class Rectangle{
    int sideA;
    int sideB;

    Rectangle(int value1, int value2){
        int sideA = value1;
        int sideB = value2;
        System.out.println("SideA Constructor = " + sideA +"\n");
        System.out.println("SideB Constructor = " + sideB +"\n");
    }
    public void showInfo(){
        System.out.println("SideA = " + sideA +"\n");
        System.out.println("SideB = " + sideB +"\n");
}
```

- `int sideA = value1;`
  - inside the constructor creates a new local variable also called `sideA` and gets the `value1`. On method exit this variable goes away

Categorizing methods

- Accessors
  - methods that are used to obtain information from an object without affecting its state.
    - e.g. getAge(), getArea()
- Mutators
  - methods that alter the state of the object
    - e.g. incrementAgeByOne(), setAge(int)
- The "features" of an object refer to both its state and behavior.

Control flow (if)

if (<*boolean-expression*>){

<*then-block*>

Optional

}else{

<*else-block*>

}

```
if (age < 21){
    underAge = true;
}else{
    underAge = false;
}
```

if (<*boolean-expression*>){

<*then-block*>

Repeat

}else if(<*boolean-expression*>){

<*elseif-block*>

} else{

<*else-block*>

```
if (age < 19 && age >= 13){
    teenager = true;
}else if(age >= 19 && age< 21){
    underAge = true;
}else if (age >= 21 && age<60){
    adult = true;
}else{
    seniorCitizen = true;
}
```

Control flow (while)

while(<*test*>){

    <*while-block*>

}

```
int i=0;
while(i <= 10){
   System.out.println(“Hi!”);
   i++;
}
```

do{

    <*do-while-block*>

}while(<*test*>)

```
int i=0;
do{
   System.out.println(“Hi!”);
   i++;
}while(i <= 10)
```

Control flow (for)

for(<*init*>;<*termination*>;<*incr*>){

    <*for-block*>

}

```
for(int  i=0; i <= 10; i++){
    System.out.println("Hi!");
}
```

```
int i=0;
while(i <= 10){
    System.out.println("Hi!");
    i++;
}
```

```
for(int  i=0; i <= 10; i++){
    System.out.println("Hi!");
}
```

Control flow (switch)

switch(<*integer-expre*>):

Repeat

case <*int-val*>: <*statements*>

default:<*statements*>

}

- `break` forces control to move to the first statement after the whole switch block

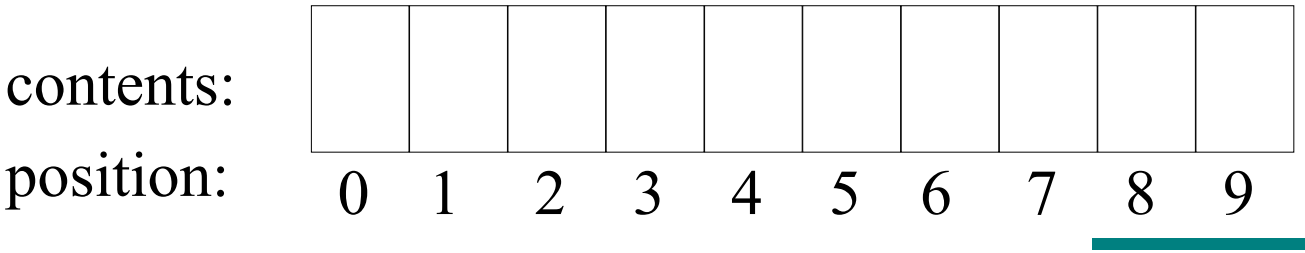
- Without `break` execution falls through to the next switch case.

```
int age = 12;
switch(month){
    case 10:
        teacher="Smith";
        break;
    case 11:
        teacher="Jones";
        break;
    default:
     if (age < 10 || age > 12 ){
        System.out.println("Wrong
          age group);
    }
}
// rest of the program
```

Java Arrays

- An ordered collection, or numbered list of values.
  - values can be primitive, objects or other arrays
  - All elements of the array must be of the same type

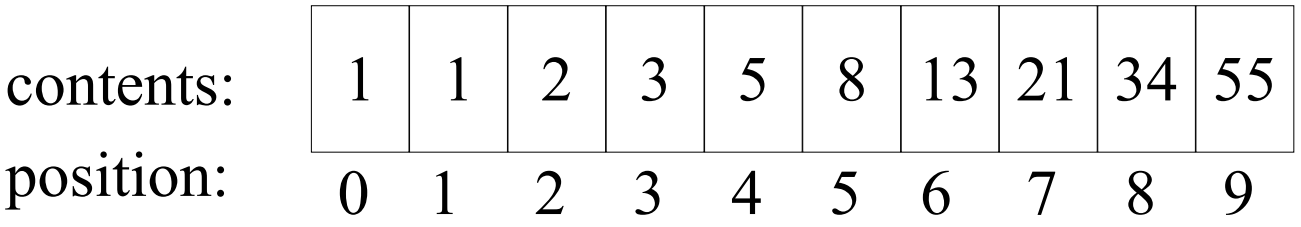
```
// defining arrays
int a;
int[] arrayofIntegers;
Triangle[] arrayofTriangles;

//creating
String[] lines = new String[9];
int[] sequence = new int[10];
```



Java Arrays(cont)

- Using [] and the number corresponding to the index allows you to read/write to that array location

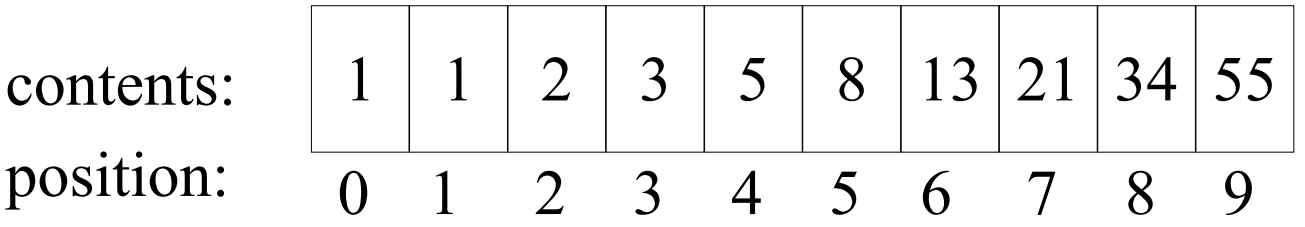
```
//creating
int[]  sequence = new int[10];

//assigning to an array
sequence[0] = 1;
sequence[1] = 1;
sequence[2] = 2;
sequence[3] = 3;
sequence[4] = 5;
sequence[5] = 8;
sequence[6] = 13;
sequence[7] = 21;
sequence[8] = 34;
sequence[9] = 55;
//or
sequence ={1,1,2,3,58,13,21,34,55};
```



Java Arrays(cont)

- You can get the length of the array
  - special call on arrays
- You cannot change the length of an array once you have defined it.
  - we are again using '.' but it is specially made for array types in Java.

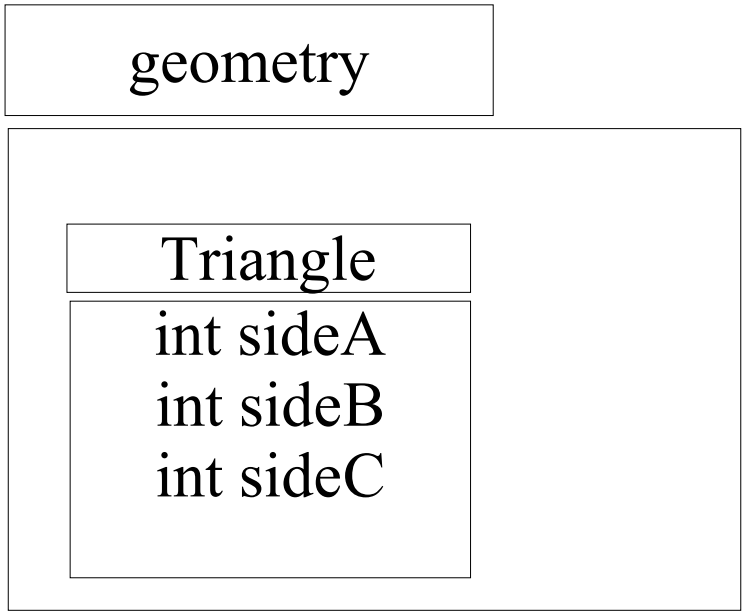
```
//assigning to an array
sequence ={1,1,2,3,58,13,21,34,55};

for (int i=0; i < sequence.length;i++)
{
    System.out.prinln("Index "+i+
                          " holds "+
                           sequence[i]);
}
```



Categorizing classes

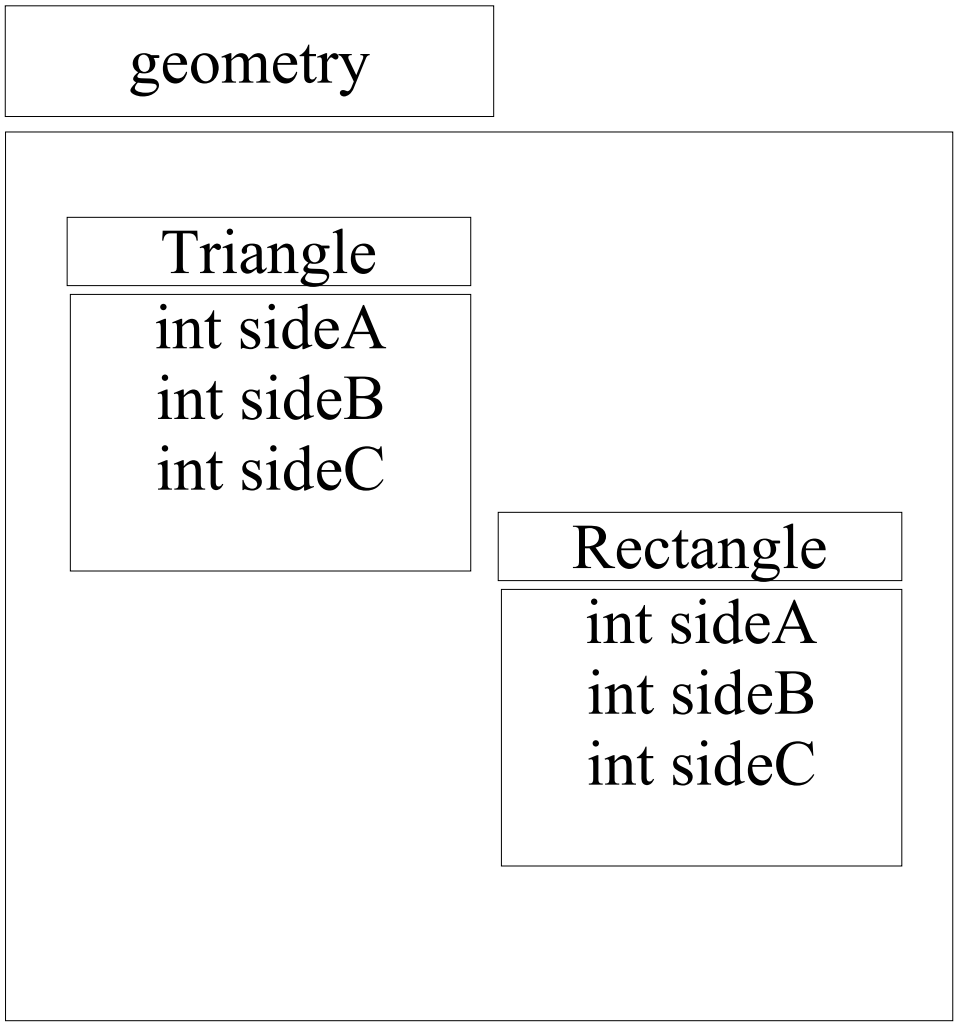
- Packages
  - bundles that can be created using Java to group together classes (library).

```
package geometry;

public class Triangle{
   int sideA;
   int sideB;
   int sideC;

// same as before
```



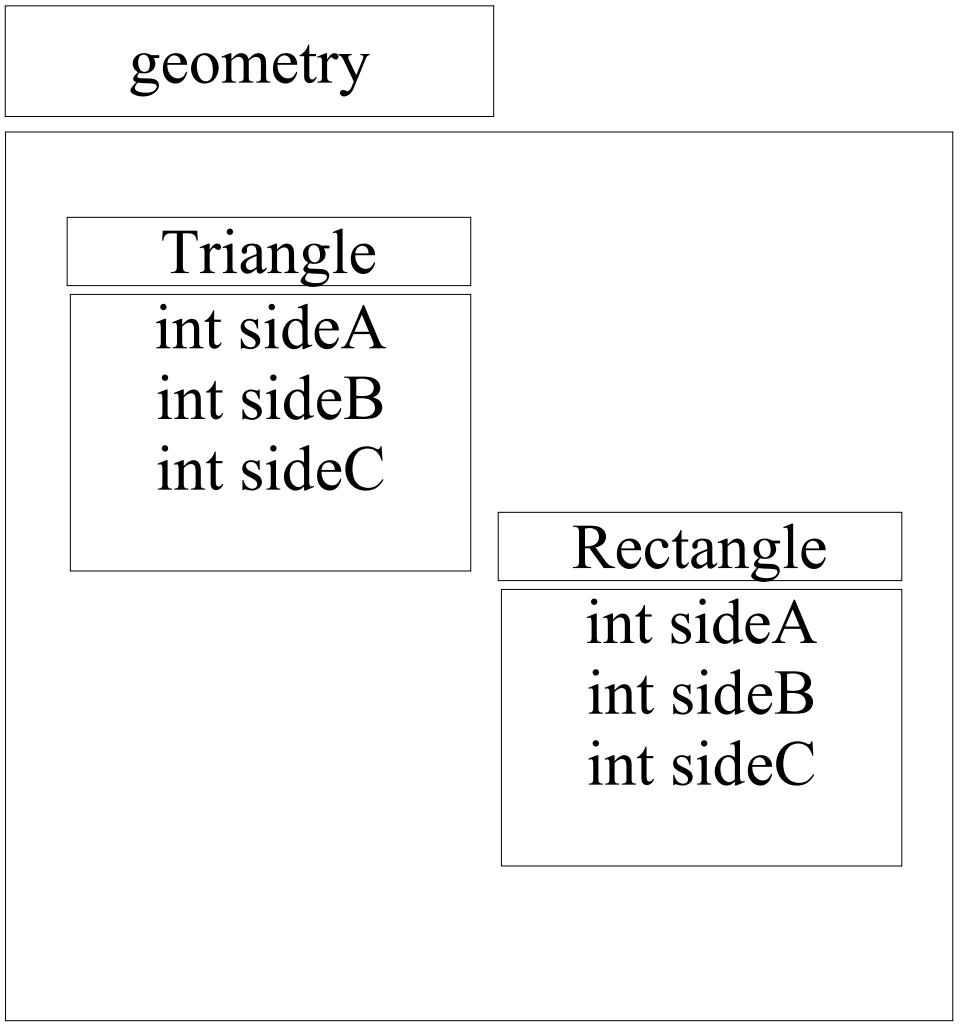
Categorizing classes (cont)

```
package geometry;

public class Triangle{
   int sideA;
   int sideB;
   int sideC;

// same as before
}

package geometry;

public class Rectangle{
   int sideA;
   int sideB;

// same as before
}
```



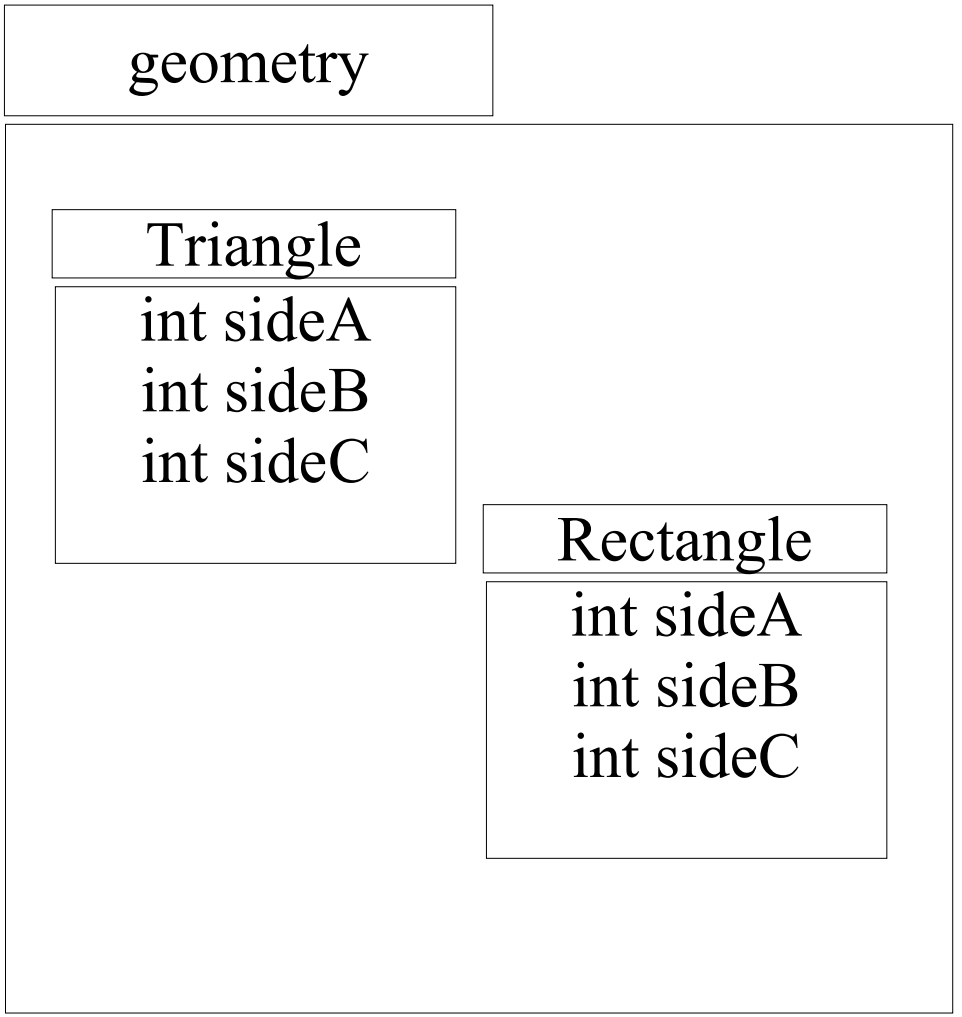
Categorizing classes (cont)

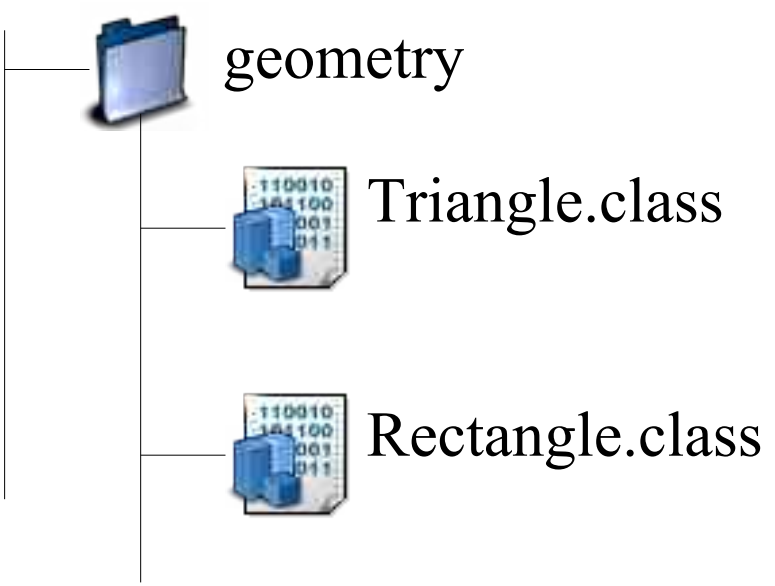
- `public class Triangle`
  - allows for objects in different packages to create and use `Triangle`
- `private int sideA`
  - doesn't allow `Rectangle` to directly access `sideA` (using '.' notation)
  - accessor methods have to be used.
- API
  - provides information as to which features are available inside a library



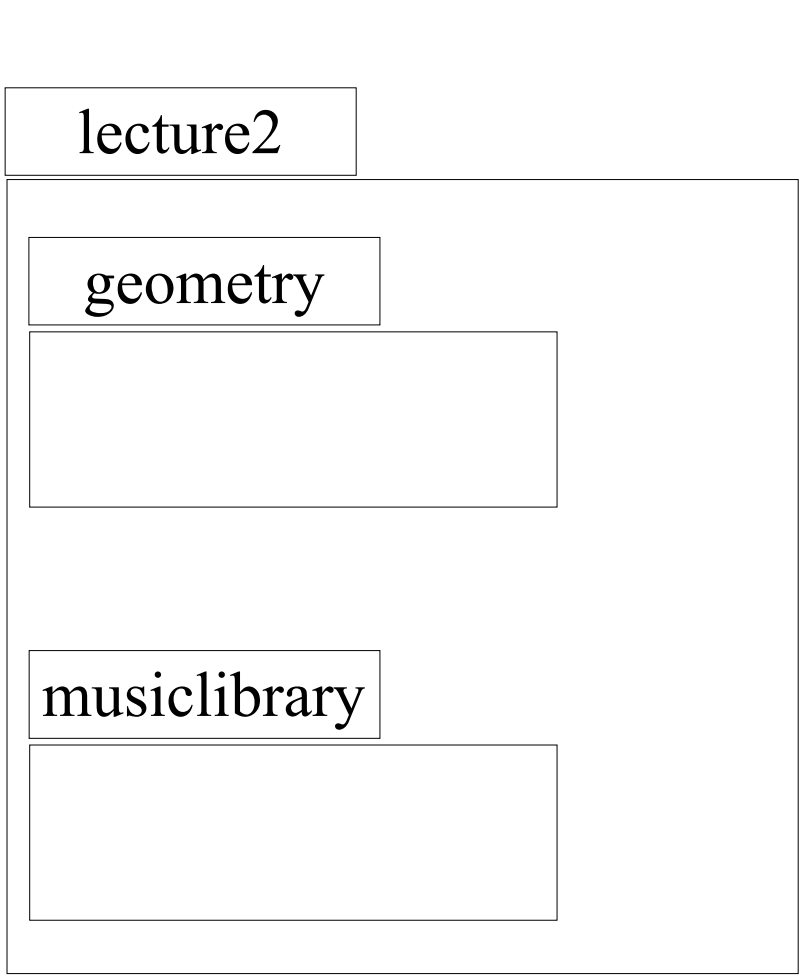
Categorizing classes (cont)

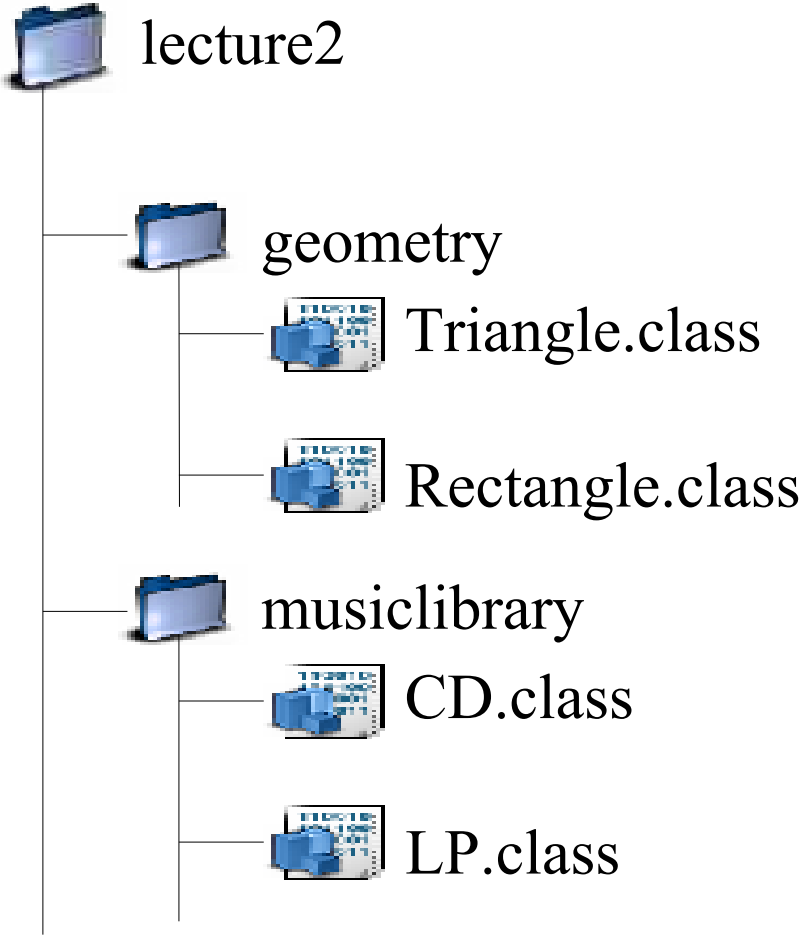
- Java imposes a restriction when creating packages



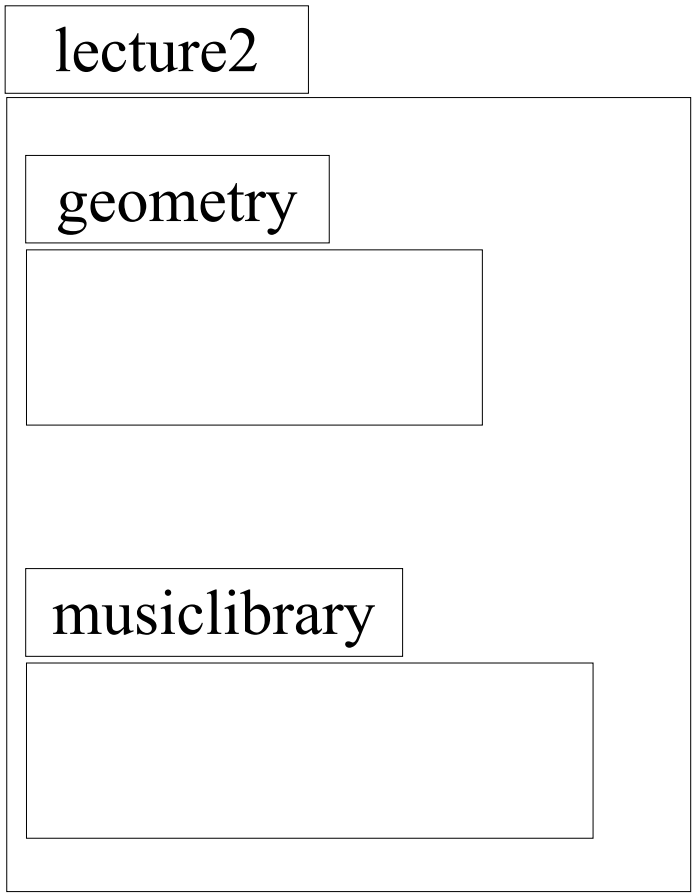


Categorizing classes (cont)





Categorizing classes (cont)



Make available the classes found in the package that follows

```
import lecture2.geometry.Triangle;

public class Main{
  public static void main(String[] args){
    Triangle test = new Triangle(3,4,5);
  }
}
```

`Triangle` is declared `public` and can be used by objects of a different package (i.e. `Main`)

A recipe for coding

- Give your classes "good" names
  - Triangle, Rectangle, CDLibrary
  - **NOT**: MyClass, ThisClass, AnotherObject
- Hide unnecessary information from other objects
  - internal memory of the object should always be declared `private`
  - use accessors and mutators to read and write
- Comment each method with
  - expected values as input (pre-condition)
  - expected results given correct inputs (post-conditions)
  - write example inputs and outputs inside your comments

A recipe for coding(cont)

```
/**
* Class: Triangle
* Author: Theo
* Goal: represents a right-angle
*       triangle. Calculates area
*       and perimeter
*/

public class Triangle {
  // Sides are integers
  int sideA;
  int sideB;
  int sideC;

//Accessors
  public int getSideA(){
    return sideA;
  }
  public int getSideB(){
    return sideB;
  }
  public int getSideC(){
    return sideC;
  }
```

```
// mutators
  public void setSideA(int value){
    sideA=value;
  }
 public void setSideB(int value){
    sideB=value;
  }
 public void setSideC(int value){
    sideC=value;
  }

/**
* area():int
* calculate the area of a rTriangle
* pre: true
* post: result = (sideA*sideB)/2
*/
  public int area(){
    int result = 0;
    result = (sideA*sideB)/2;
    return result;
  }
 ...
}
```

A recipe for coding(cont)

- After you compile with no errors **YOU ARE NOT DONE**.
- You should stress test your code
  - test for different values as input
    - try also wrong values to see how your program behaves
  - test for the different execution flows
    - provide test cases that exercise control flow constructs with **ALL** their branching.
  - make sure that all the test cases provide the "correct" results.
    - correct results = same output as the one described by your customer (homework, design, text description)
  - if a test case fails
    - detect, fix and run again