Introduction to Computational Thinking

Object-oriented programming

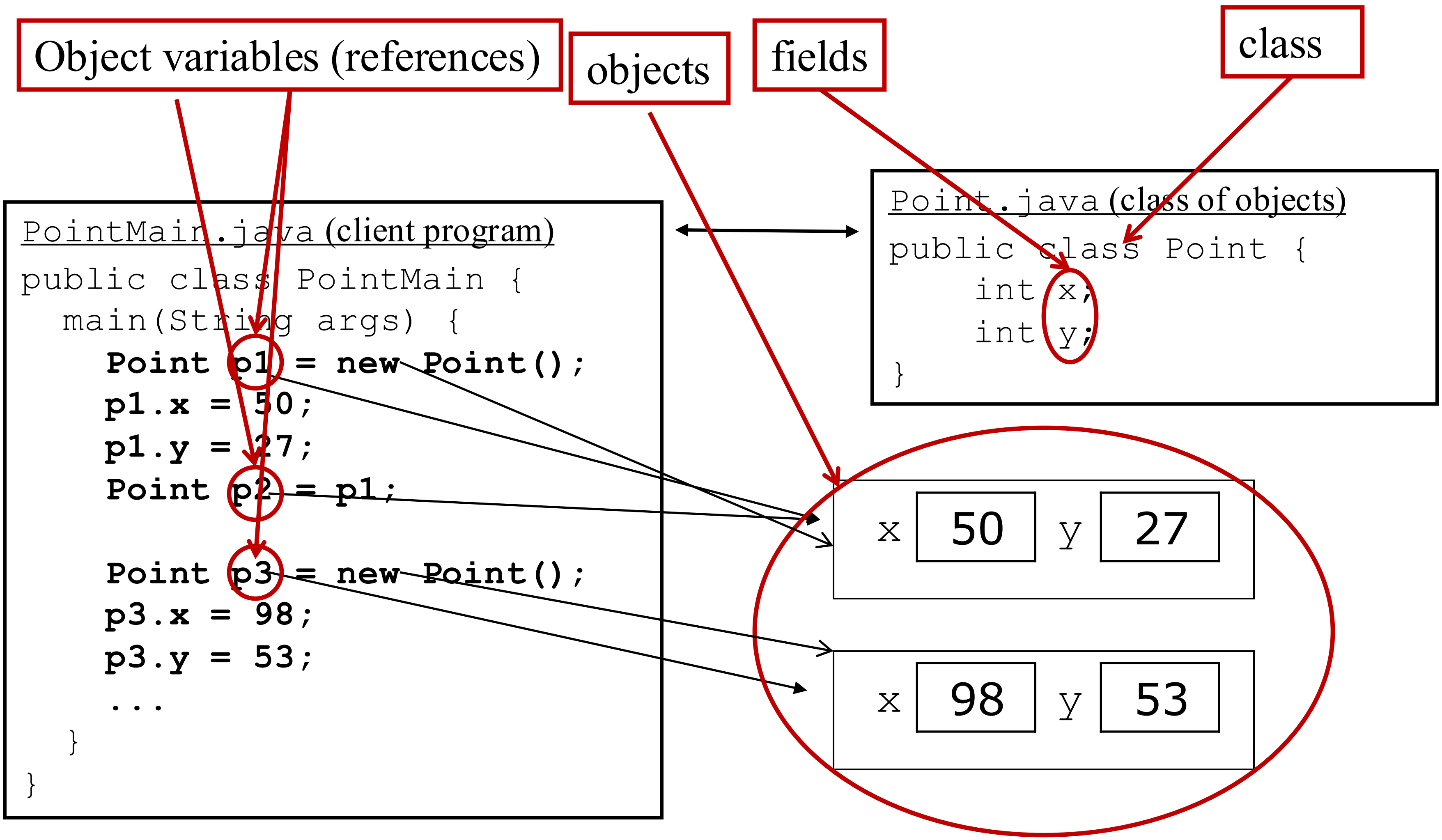
Qiao Xiang, Qingyu Song

https://sngroup.org.cn/courses/ct-xmuf25/index.shtml

12/10/2025

This deck of slides are heavily based on cs112 at Yale University and cs101 at UCAS, respectively, by courtesy of Dr. Y. Richard Yang and Dr. Zhiwei Xu.

Recap: Class, Object, Variable, Field



Recap: Static Method vs Instance Method

```java
public class Point {
    int x;
    int y;
    public static void draw(Point p) {
        StdDraw.filledCircle(p.x, p.y, 3, 3);
        StdDraw.textLeft(p.x, p.y, "(" + p.x + ", " + p.y + ")" );
    }
}
```

```java
public class Point {
    int x;
    int y;
    public static void draw(Point p) {
        StdDraw.filledCircle(p.x, p.y, 3, 3);
        StdDraw.textLeft(p.x, p.y, "(" + p.x + ", " + p.y + ")");
    }
}
```

Recap: Defining Related Method and Data in the Same Class: Instance Method

```
public class Point {
    int x;
    int y;

    public static void draw(Point p) {
        StdDraw.filledCircle(p.x, p.y, 3);
        StdDraw.textLeft(p.x, p.y,
                         "(" + p.x + ", " + p.y + ")");
    }
}
```

```
Point p1 = new Point();
p1.x = 7; p1.y = 2;
p1.draw(); // Point.draw(p1);

Point p2 = new Point();
p2.x = 4; p2.y = 3;
p2.draw(); // Point.draw(p2);
```

p1 provides the implicit parameter: The x and y in draw() are those of the object referenced by **p1**.

p2 provides the implicit parameter: The x and y in draw() are those of the object referenced by **p2**.

Outline

- Defining classes
  - Data encapsulation (struct)
  - Data+behavior encapsulation
    - instance methods
      - constructors

Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

```
Point p = new Point();
p.x = 3;
p.y = 8;                         // tedious(乏味)
```

- We'd rather specify the fields' initial values at the start:

```
Point p = new Point(3, 8);    // better!
```

  - We are able to do this with most types of objects in Java.

Constructors

- **constructor**: a special method to initialize the state of new objects.

```
public type(parameters) {
    statements;
}
```

1. runs when the client uses the `new` keyword
2. no return type should be specified;
   it implicitly "returns" the new object being created
3. If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to zero-equivalent values.

Constructor example

```
public class Point {
    int x;
    int y;

    // Constructs a Point at the given x/y location.
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }


    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    ...
}
```

Client (User) code

```
public class PointMain3 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
    }
}
```

Multiple Constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters (same rule of method overloading).

Common Constructor Issues

1. By accidentally giving the constructor a return type.

   Not a constructor, but a method named `Point`

```
public void Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}
```

2. Declare a local variable with the same name as a field.

   Rather than storing value into the field, the param is passed to local variable. The field remains 0.

```
public class Point {
    int x;
    int y;
    public Point(int initialX, int initialY) {
        int x = initialX;
        int y = initialY;
    }
}
```

Common Constructor Issues

- **"shadowing"**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field

```
public class Point {
    int x;
    int y;
    ...
    public Point(int x, int y) {
        System.out.println("x = " + x);// para x
    }
```

  - In most of the class, `x` and `y` refer to the fields.
  - In `Point(int x, int y)`, `x` and `y` refer to the method's parameters.

The `this` keyword: Access Fields/Methods within Class

- **`this`** : Refers to the implicit parameter inside your class.

  (a variable that stores the object on which a method is called)

  - Refer to a field: `this.`**field**
  - Call a method: `this.`**method**`(`**parameters**`);`

Fixing "Shadowing" with `this`

- To refer to the data field `x`: `this.x`
- To refer to the parameter `x`: `x`

```
public class Point {
    int x;
    int y;

    ...
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Calling another constructor

```
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);      // calls (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

Summary: Class Definition Components

- Variables
  - fields (instance variables per object)
  - static variables (shared by all objects)
- Methods
  - static methods (method usable with or without objects)
    - Can access only static variables
  - instance methods (can be used only on objects; can access both static and instance variables)
    - Constructors
    - Accessors (do not change object state)
    - Mutators (modify object state)

Outline

- Admin and recap
- Defining classes
  - Data encapsulation (struct)
  - Data+behavior encapsulation (OOP)
  - OOP design methodology

Example: Procedural vs OOP Design

Function-oriented

```
public class DrawRocket{

  public static void main(String args[]){
    for (int size = 3; size < 10; size++){
      drawRocket(size);
    }
  }

  public static void drawRocket(int scale){
    printCap(scale);
    ...
  }

  ...

  public static void printCap(int scale){
    ...
  }
}
```

Object-oriented

```
public class RocketDrawing{
  public static void main(String args[]){
    for (int size = 3; size < 10; size++){
      Rocket curRocket = new Rocket(size);
      curRocket.draw();
    }
  }
}

public class Rocket{
  public int rocketSize;

  public Rocket(int rocketSize){
    this.rocketSize = rocketSize;
  }

  public void draw(){
    printCap();
    ...
  }
  ...
  public void printCap(){
    ...
  }
}
```

Recap: Design and Implementation Methodology: Procedural Based

- Design (goal oriented)
  - top-down stepwise goal-driven method decomposition
  - methods designed are those needed for the current goal
  - verb driven

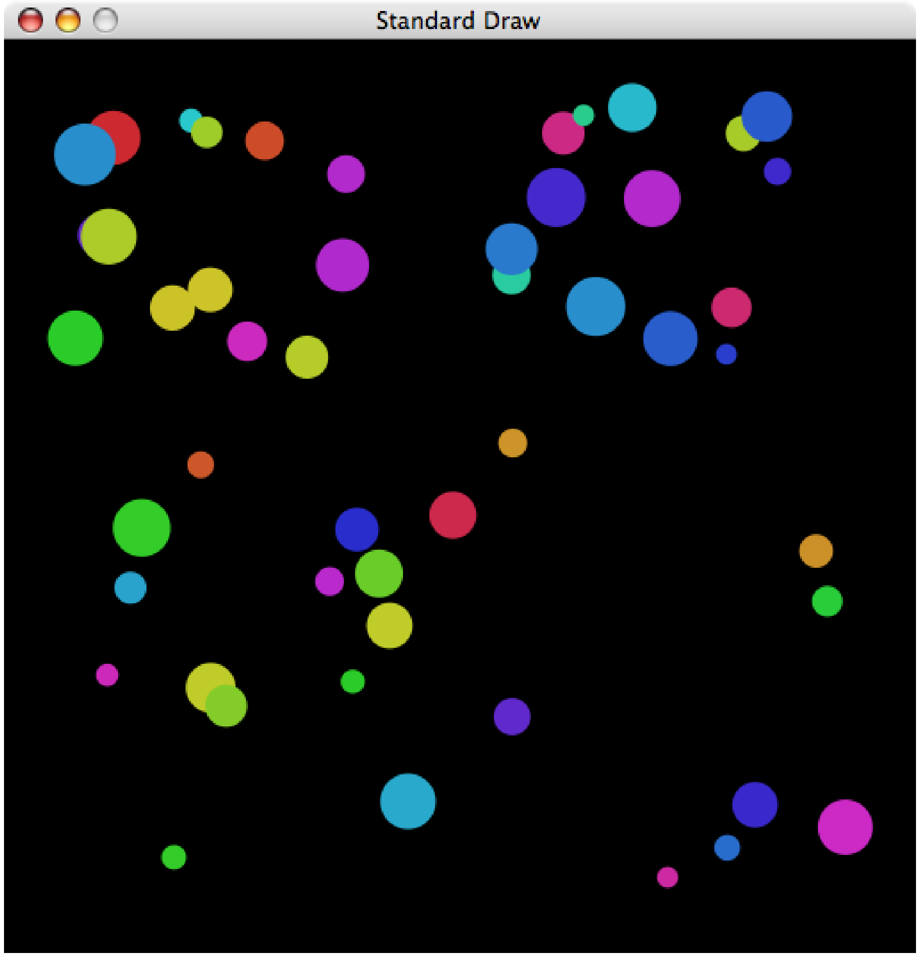
- Program implementation and testing
  - bottom-up

Design and Implementation Methodology: Object-Oriented

- Design
  - Identify objects that are part of the **problem domain** or solution
    - Each object has state (variables)
    - Each object has behaviors (methods)
  - Often do not consider one specific goal, but rather a context (**problem domain**), to lead to more reusable, extensible software
  - noun driven

Example: The `Ball` Class

- We define a `Ball` class to model self-bouncing balls



The `Ball` Class

- Design questions:
  - State: what field(s) do we need to represent the state of a self-bouncing ball?
    - `rx, ry,` current position
    - `radius`: radius
    - `vx, vy,` speed
    - `color,` current color
    - `left, right, bottom, top:` cage (boundaries)
  - Behaviors/operations: what are some common behaviors of a self-bouncing ball?
    - A default constructor, to set up a random ball in unit square
    - A constructor, to set up a ball with given parameters
    - A `move` method, to update position
    - A `draw` method, to display

See Ball.java, BouncingBalls.java

Bouncing Ball in Unit Square

Ball.java

```
public class Ball {
    double rx, ry;
    double vx, vy;
    double radius;

    public Ball() {
        rx = ry = 0.5;
        vx     = 0.015 - Math.random() * 0.03;
        vy     = 0.015 - Math.random() * 0.03;
        radius = 0.01  + Math.random() * 0.01;
    }

    public void move() {
        if ((rx + vx > 1.0) || (rx + vx < 0.0)) vx = -vx;
        if ((ry + vy > 1.0) || (ry + vy < 0.0)) vy = -vy;
        rx = rx + vx;
        ry = ry + vy;
    }

    public void draw() {
        StdDraw.filledCircle(rx, ry, radius);
    }
}
```

instance variables

constructor

Bounce反弹

methods

An Array of Objects

- It is common that we create an array of objects
  - Use `new` to invoke constructor and create each one.

```
public class BouncingBalls {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      Ball balls[] = new Ball[N];
      for (int i = 0; i < N; i++)
         balls[i] = new Ball();

      while(true) {
         StdDraw.clear();
         for (int i = 0; i < N; i++) {
            balls[i].move();
            balls[i].draw();
         }
         StdDraw.show(20);
      }
   }
}
```

create and initialize N objects

animation loop

Outline

- Admin and recap
- Defining classes
  - Data encapsulation (struct)
  - Data+behavior encapsulation (OOP)
  - OOP design methodology
  - Objects and reference semantics

Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

addr	value
100	0
101	0
102	0
103	0
104	0
105	0
106	0
107	0
108	0
109	0
110	0
111	0
112	0

main memory
(64-bit machine)

Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

b1
100

addr	value	
100	0.50	rx
101	0.50	ry
102	0.05	vx
103	0.01	vy
104	0.03	radiu
105	0	
106	0	
107	0	
108	0	
109	0	
110	0	
111	0	
112	0	

main memory
(64-bit machine)

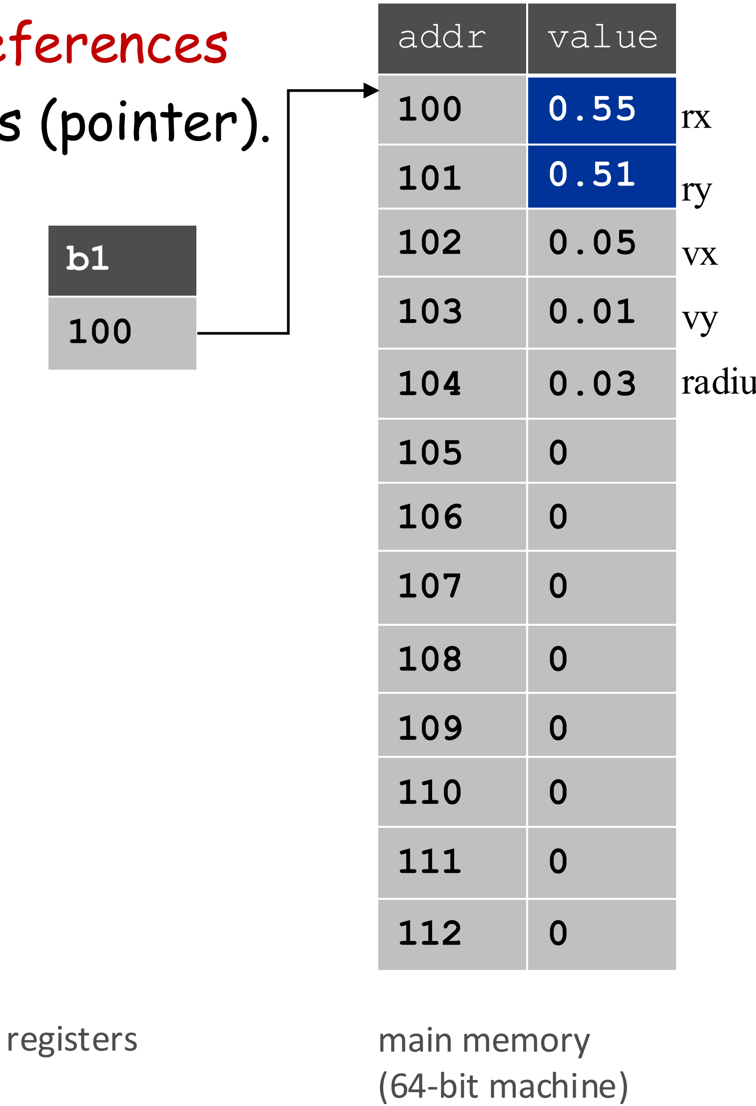
Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```



Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

b1
100

registers

addr	value	
100	0.60	rx
101	0.52	ry
102	0.05	vx
103	0.01	vy
104	0.03	radiu
105	0	
106	0	
107	0	
108	0	
109	0	
110	0	
111	0	
112	0	

main memory
(64-bit machine)

Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

b1
100

b2
107

registers

addr	value	
100	0.60	rx
101	0.52	ry
102	0.05	vx
103	0.01	vy
104	0.03	radiu
105	0	
106	0	
107	0.50	rx
108	0.50	ry
109	0.07	vx
110	0.04	vy
111	0.04	radiu
112	0	

main memory
(64-bit machine)

Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

b1
100

b2
107

registers

addr	value	
100	0.60	rx
101	0.52	ry
102	0.05	vx
103	0.01	vy
104	0.03	radiu
105	0	
106	0	
107	0.57	rx
108	0.54	ry
109	0.07	vx
110	0.04	vy
111	0.04	radiu
112	0	

main memory
(64-bit machine)

Object References

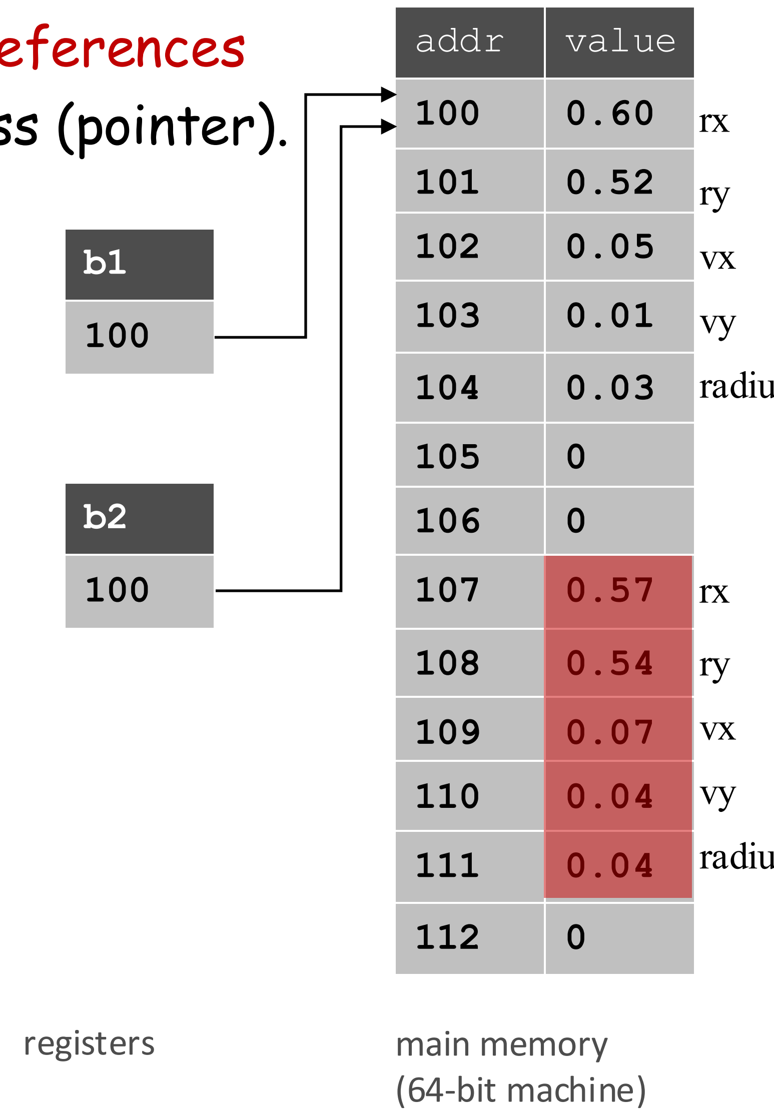
- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```

Data stored in 107 – 111 for bit recycler (garbage collection).



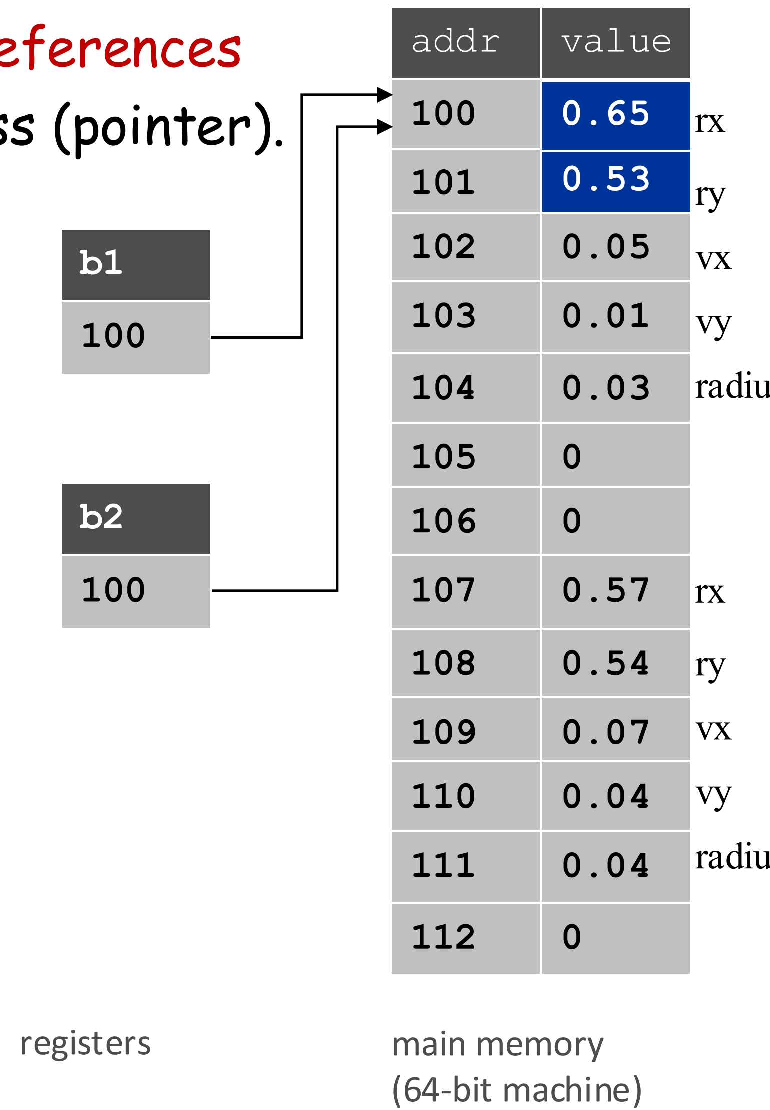
Object References

- Recall: non-primitive variables store references
- Reference: essentially machine address (pointer).

```
Ball b1 = new Ball();
b1.move();
b1.move();

Ball b2 = new Ball();
b2.move();

b2 = b1;
b2.move();
```



Moving `b2` also moves `b1` since they are aliases that reference the same object.

Outline

- Admin and recap
- Defining classes
  - Data encapsulation (struct)
  - Data+behavior encapsulation (OOP)
  - OOP design methodology
  - Objects and reference semantics
  - OOP design examples

Design and Implementation Methodology: Object-Oriented

- Design
  - Identify objects that are part of the problem domain or solution
    - Each object has state (variables)
    - Each object has behaviors (methods)
      - Constructors, accessors, mutators(修改器)
  - Often do not consider one specific goal, but rather a context
  - noun driven

Example: The `BankAccount` Class

- We define an `BankAccount` class to model a bank account
- Design questions:
  - State: what field(s) do we need to represent the state of a bank acct?
    - `acctNumber`, an integer
    - `acctName`, a string
    - `balance`, an integer
  - Behaviors/operations: what are some common behaviors of a bank account in a simple banking context?
    - A constructor, to set up the object
    - Accessors
      - a `getBalance` method, to return balance
      - a `toString` method, to return a string description of the current state
    - Mutators
      - a `withdraw` method, to withdraw from the account
      - a `deposit` method, to deposit into the account
      - a `addInterest` method, to add interest

See BankAccount.java, Transactions.java

Example: Account and Transactions

```
public class BankAccount {
    final double RATE = 0.035;
    long acctNumber;
    String acctName;
    double balance;

  public BankAccount (String owner, long
     account, double initial) {
    acctName = owner;
    acctNumber = account;
    balance = initial;
  }

  public double deposit (double amount) {
    if (amount > )
        balance = balance + amount;

    return balance;
  }
  ...
}
```

```
public static void main (String[] args) {
    BankAccount aliceAcct =
        new BankAccount ("Alice", 11111, 100.00);

    BankAccount bobAcct =
        new BankAccount ("Bob", 22222, 200.00);

    BankAccount charlesAcct =
       new BankAccount ("Charles", 33333, 300.00)

    bobAcct.deposit (30.00);

    ...
}
```

Example: The Three `BankAccount` Objects in Transactions

aliceAcct: BankAccount

acctNumber = 11111
acctName = "Alice"
balance = 100.00

bobAcct: BankAccount

acctNumber = 22222
acctName = "Bob"
balance = 200.00

charlesAcct: BankAccount

acctNumber = 33333
acctName = "Charles"
balance = 300.00

aliceAcct: BankAccount

acctNumber = 11111
acctName = "Alice"
balance = 100.00

bobAcct: BankAccount

acctNumber = 22222
acctName = "Bob"
balance = **230.00**

charlesAcct: BankAccount

acctNumber = 33333
acctName = "Charles"
balance = 300.00

After bobAcct.deposit (30.00);

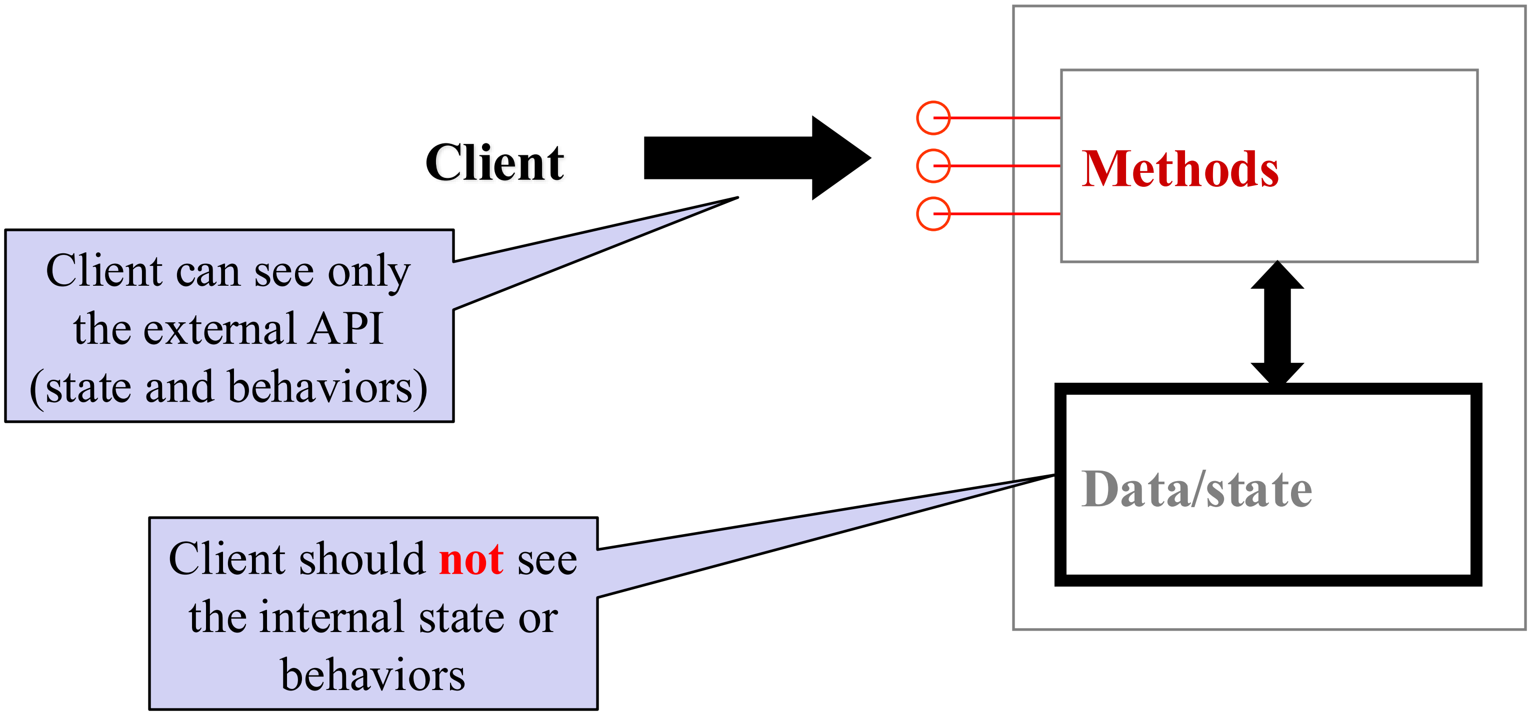
Outline

- Admin and recap
- Defining classes
  - Data encapsulation (struct)
  - Data+behavior encapsulation (OOP)
  - OOP design methodology
    - Objects and reference semantics
    - Simple examples
  - The encapsulation(封装) principle

Two Views of an Object

- You can take one of two views of an object:
  - external (API) - the interaction of the object with its users
  - internal (implementation) - the structure of its data, the algorithms used by its methods

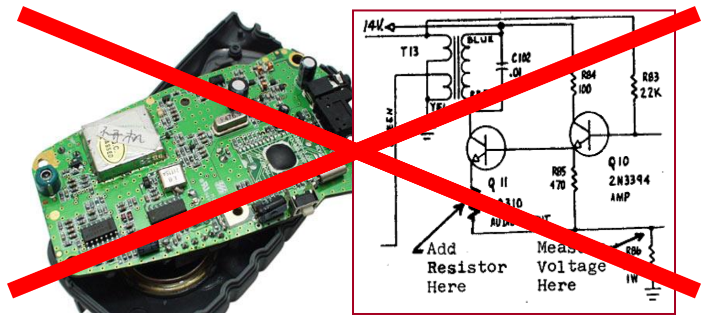
The Encapsulation Principle



Encapsulation Analogy

Client

API



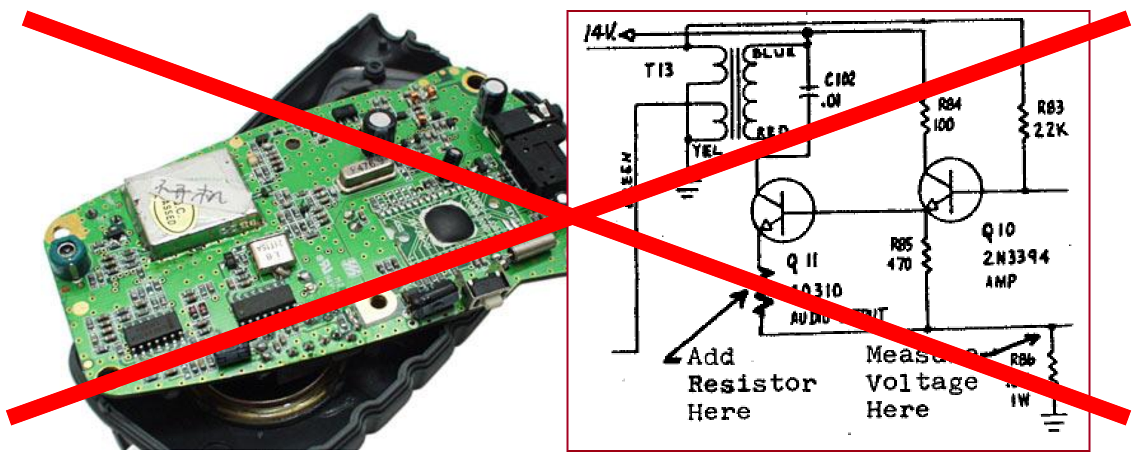
Implementation

client needs to know
how to use API

implementation needs to know
what API to implement

Encapsulation Analogy

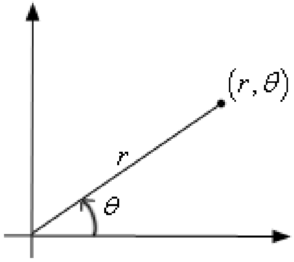
- As a client, you don't understand the inner details of iPhone, and you don't need to.
- Apple does not want to commit to any internal details so that Apple can continuously update the internal



Why Encapsulating Data

- **Consistency**: prevent "reach in" and directly alter object's state
  - Protect object from unwanted access
    - Example: `BankAccount` balance.
  - Maintain state **invariants**
    - Example: Only allow `BankAccount`s with non-negative balance.
    - Example: Only allow `Date`s with a month from 1-12.

- **Flexibility**: internally modify state without worrying about breaking others' code
  - Example: `Point` could be rewritten in polar, <u>clients will not see difference</u>.



Accomplish Encapsulation: Access Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *access modifiers*(修饰符)

- An access modifier is a Java reserved word that specifies the accessibility of a method, data field, or class
  - we will discuss two access modifiers: `public`, `private`
  - we will discuss the other modifier (`protected`) later

The `public` and `private` Access Modifiers

- access modifiers enforce encapsulation
  - **`public`** members (data and methods): can be accessed from **anywhere**
  - **`private`** members: can be accessed from a method defined in the **same class**
  - Members without an access modifier: default **private** accessibility,

Using Access Modifiers to Implement Encapsulation: Methods

- Only service methods should be made `public`

- Support or helper methods created simply to assist service methods should be declared `private`

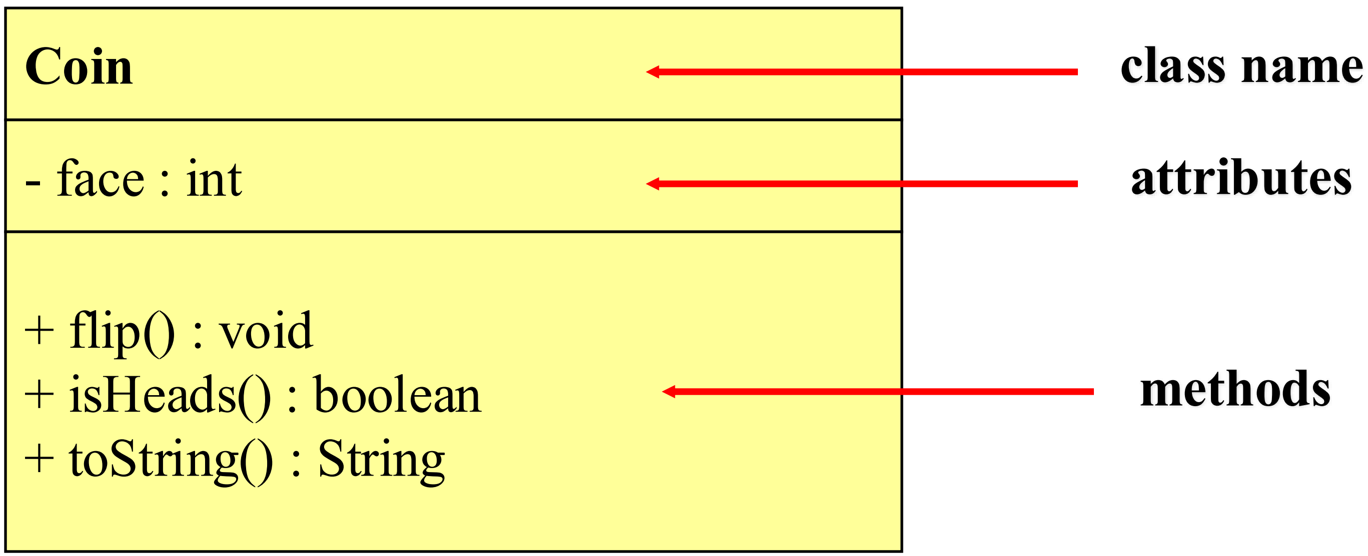
The Effects of Public and Private Accessibility

	public	private
variables	violate Encapsulation Use Caution	enforce encapsulation
methods	provide services to clients	support other methods in the class

Examples: Set the Access Modifiers

- Coin
- Ball
- BankAccount
- Point

Class Diagram



Above is a class diagram representing the Coin class.
"-" indicates private data or method
"+" indicates public data or method

Outline

- Defining classes
  - Motivation and basic syntax
  - Simple examples
  - The encapsulation principle
  - OOP analysis examples
    - Random objects vs Math.random

Static `Math.random()` method vs `Random` Objects

Recall: `Math.random()`

- `public static double random()`
  - Returns a random number between 0 and 1

- Since computer is deterministic (given the same input parameter, gives the same output), how can `Math.random()` return a different number each time?

A Little Peek into Random Number Generation

- The random numbers generated by Java are actually pseudo-random numbers

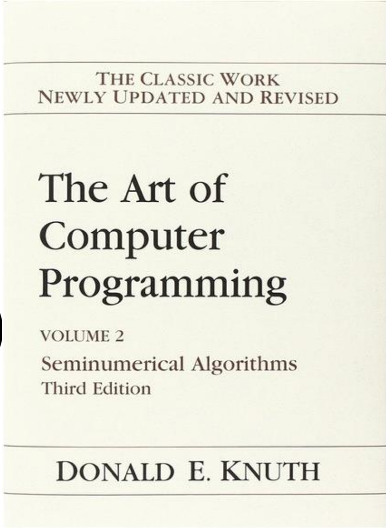
- Suppose you get a random number R_n, the next time you call it to get R_{n+1}, it returns:

  $$R_{n+1} = (R_n * 25214903917 + 11) \pmod{m}$$

  it then converts to the right range to you !

- This method is proposed by D. H. Lehmer
  - in mathematical jargon, Java uses a type of linear congruential pseudorandom number generator

- Implication: the previously returned random number must be remembered
  - random method need to have memory (state)



The `Random` class

- OOP design is perfect for implementing random numbers: a `Random` object has
  - a state variable
  - a next method: computes next state based on current state, returns the new state
- Class `Random` is found in the `java.util` package.

```
import java.util.Random;
```

Method name	**Description**
`Random(long seed)`	Create a random number using a seed (R_0)
`Random()`	Create a random number using a seed derived from time
`setSeed(seed)`	Initialize the random number generator
`nextInt(`***<max>***`)`	Returns a random integer in the range [0, *max*) in other words, 0 to *max*-1 inclusive
`nextDouble()`	Returns a random real number in [0.0, 1.0)

Using Random Number Objects

```
Random rand = new Random(); // Default, seed by time
```

get a random number from 0 to *9 inclusive:*

```
int n = rand.nextInt(10);   // 0-9
```

get a random number from 1 to *20 inclusive*

```
int n = rand.nextInt(20) + 1;   // 1-20 inclusive
```

get a random number in arbitrary range [*min*, *max*] inclusive:

```
int n = rand.nextInt(<size of range>) + <min>
```

- where ***<size of range>*** is (***<max>*** - ***<min>*** + 1)

How May `Math.random()` be Implemented Directly?

```
public class Math {
   private static int R = 0;
   public static double random() {
      R = R * 25214903917 + 11;
      // result = convert to the right range
      return result;
   }
..
}
```

How May `Math.random()` be Implemented using `Random` class?

A static variable, also called a singleton(单例).

```
public class Math {
   private static Random rand;
   public static double random() {
      if (rand == null)
         rand = new Random();
      return rand.nextDouble();
   }
..
}
```

A delegation (委托) implementation pattern.

Advantage and Issue of Using `Math.random()`

- Advantage
  - Hide object-oriented programming, simplifying programming (shorter program, no need to know seeds)
- Disadvantage
  - Without the ability to realize and control the state of the random variable
    - In testing, we want to repeat the same sequence of random numbers, but the Math.random design cannot provide this capability.

Outline

- Admin and recap
- Defining classes
  - Motivation and basic syntax
  - Simple examples
  - The encapsulation principle
  - Object examples
    - Random objects vs Math.random
  - Object-oriented analysis

Discussion

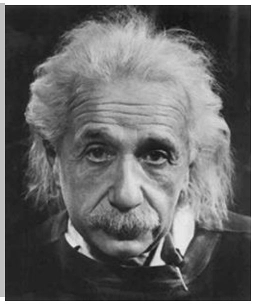
- A quite helpful tool in OO analysis is object relationship analysis. What are some basic object relationships?

Outline

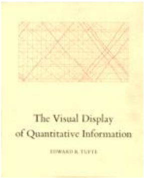
- Admin and recap
- Defining classes
  - Motivation and basic syntax
  - Simple examples
  - The encapsulation principle
  - Object examples
    - Random objects vs Math.random
- Object-oriented analysis
  - *Composition (has)/association relationship*

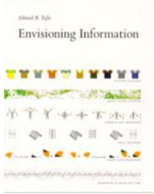
Domain: Data Visualization

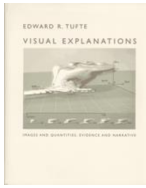
> *" If I can't picture it, I can't understand it. "*
>
> *— Albert Einstein*

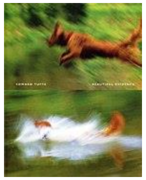


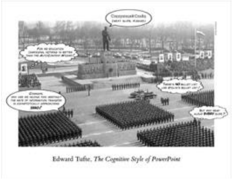
Edward Tufte (美国统计学家) Create charts with high data density that tell the truth.

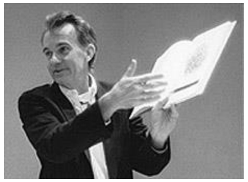




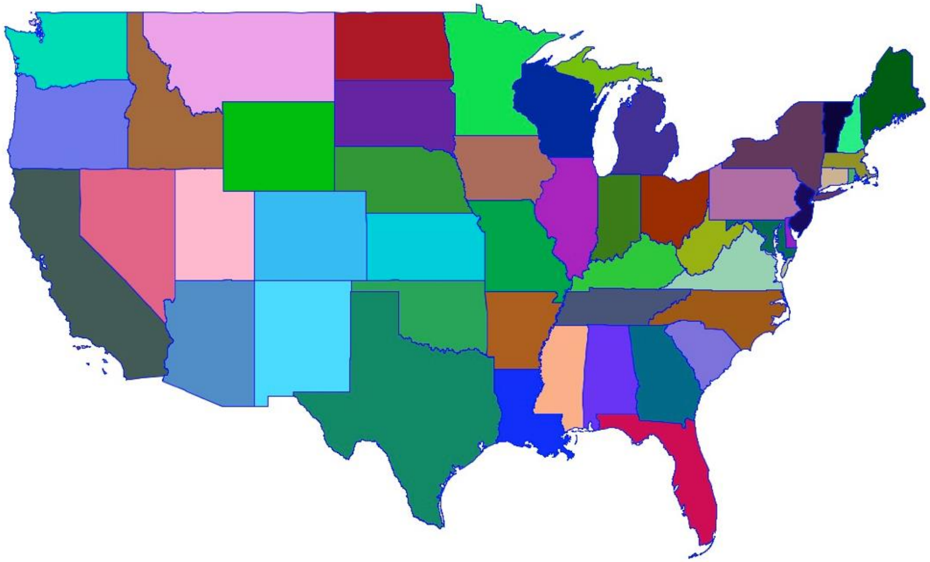


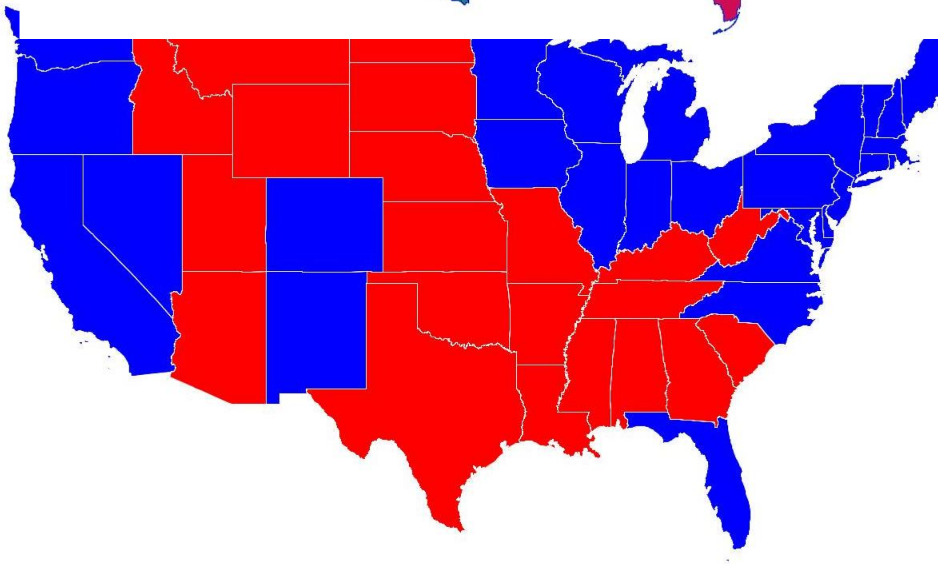


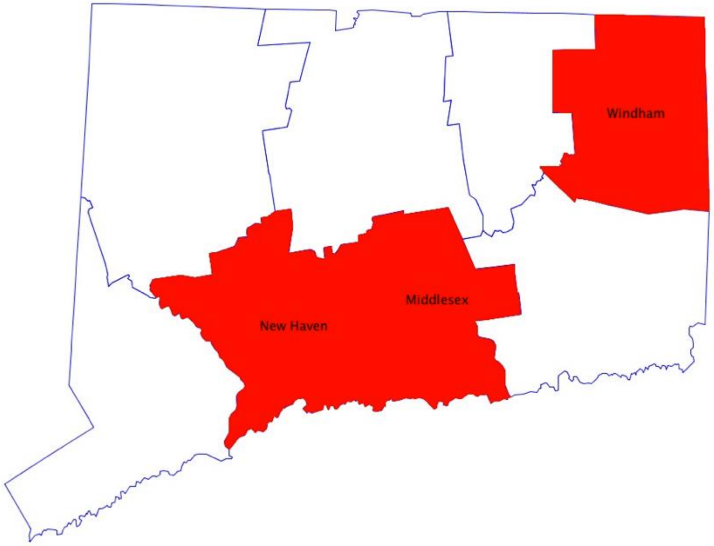




Domain: Visualization of Geographical Regions



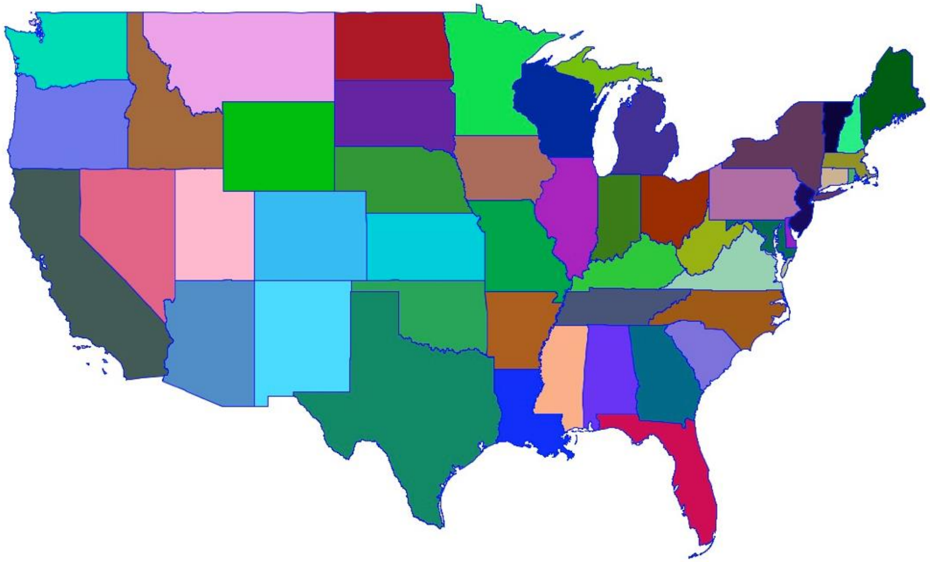


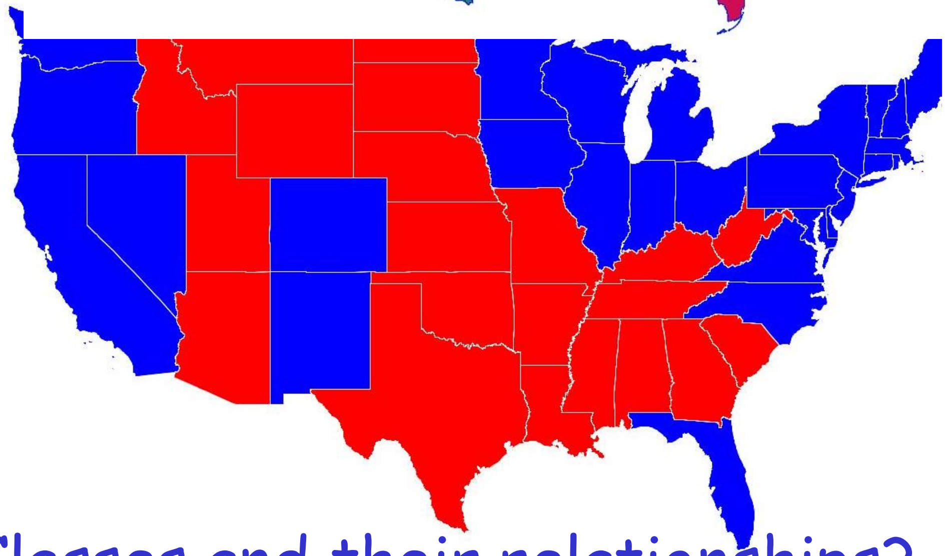


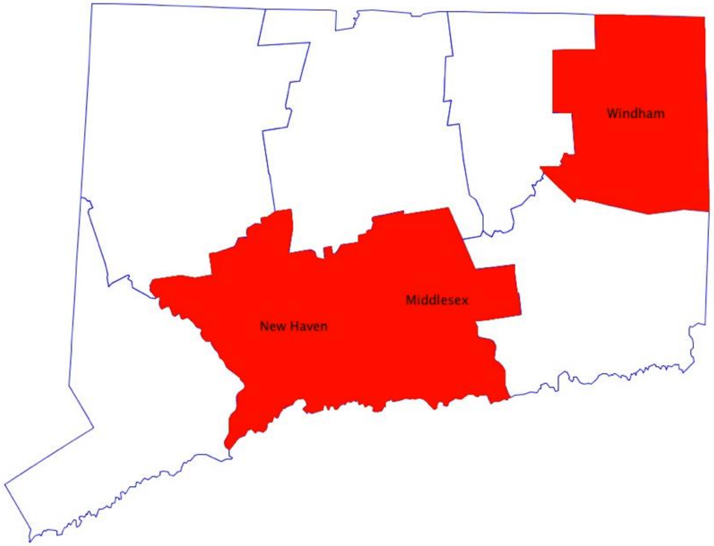
Example Use Cases

- GeoMap.java
- RandomColorMap.java
- RedBlueMap.java
- ClickColorMap.java

Example Domain: Visualization of Geographical Regions

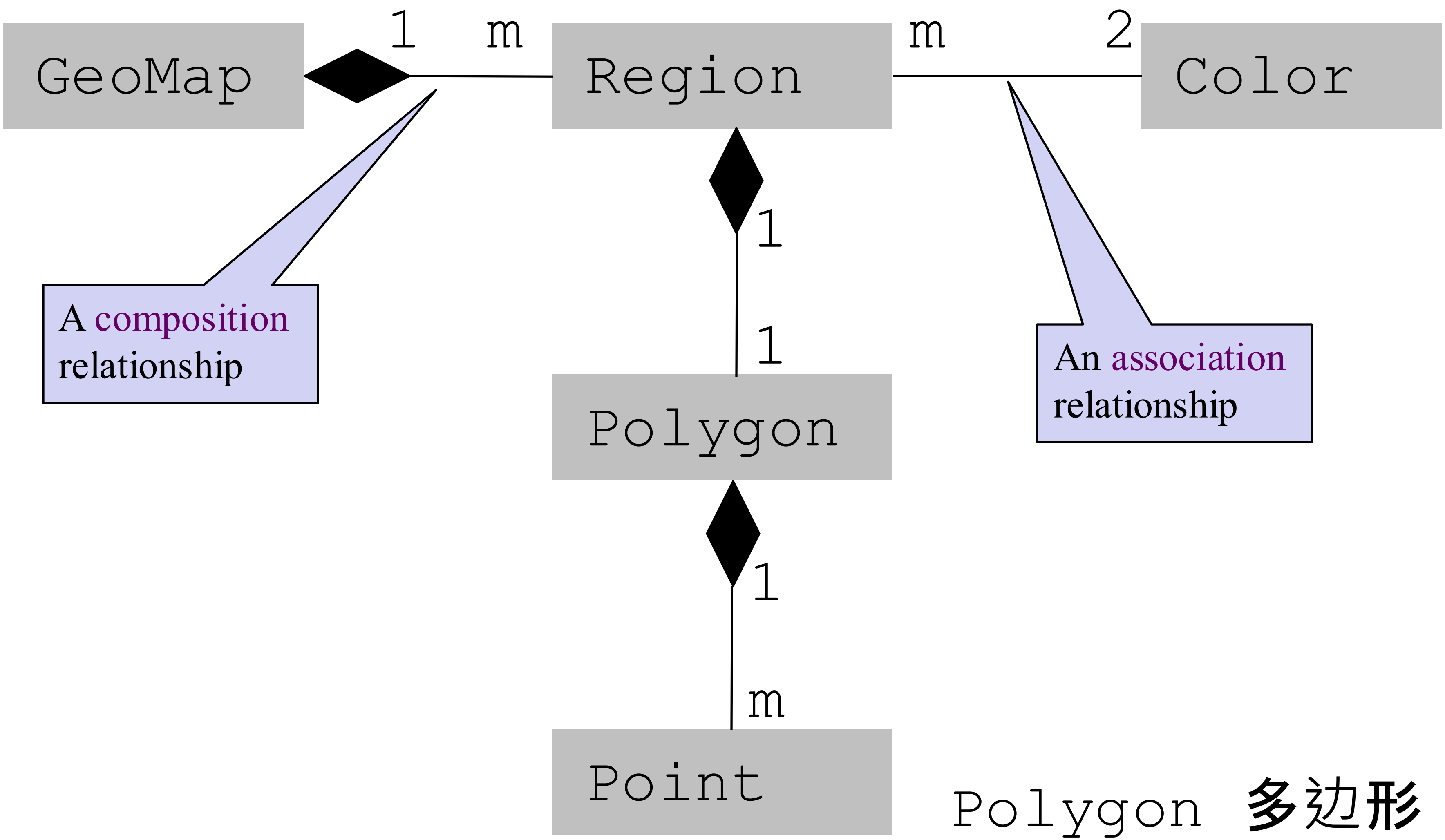






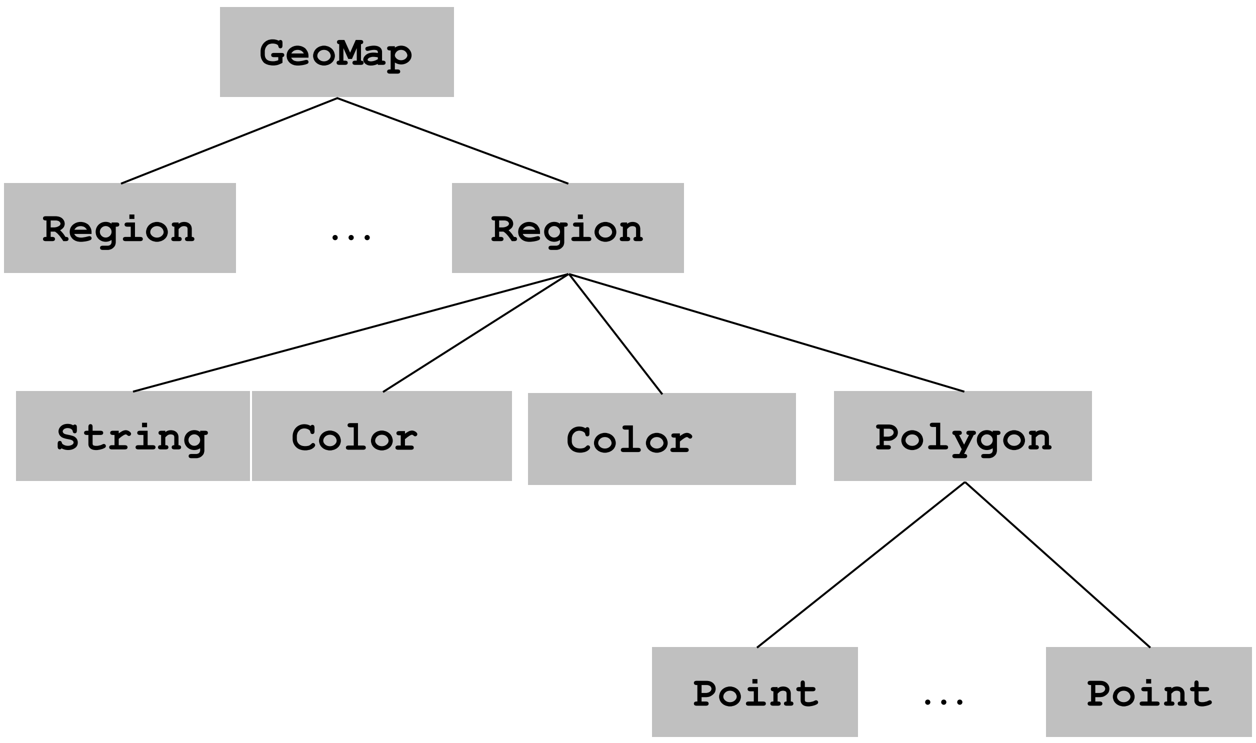
Classes and their relationships?

Major Classes and Relationship

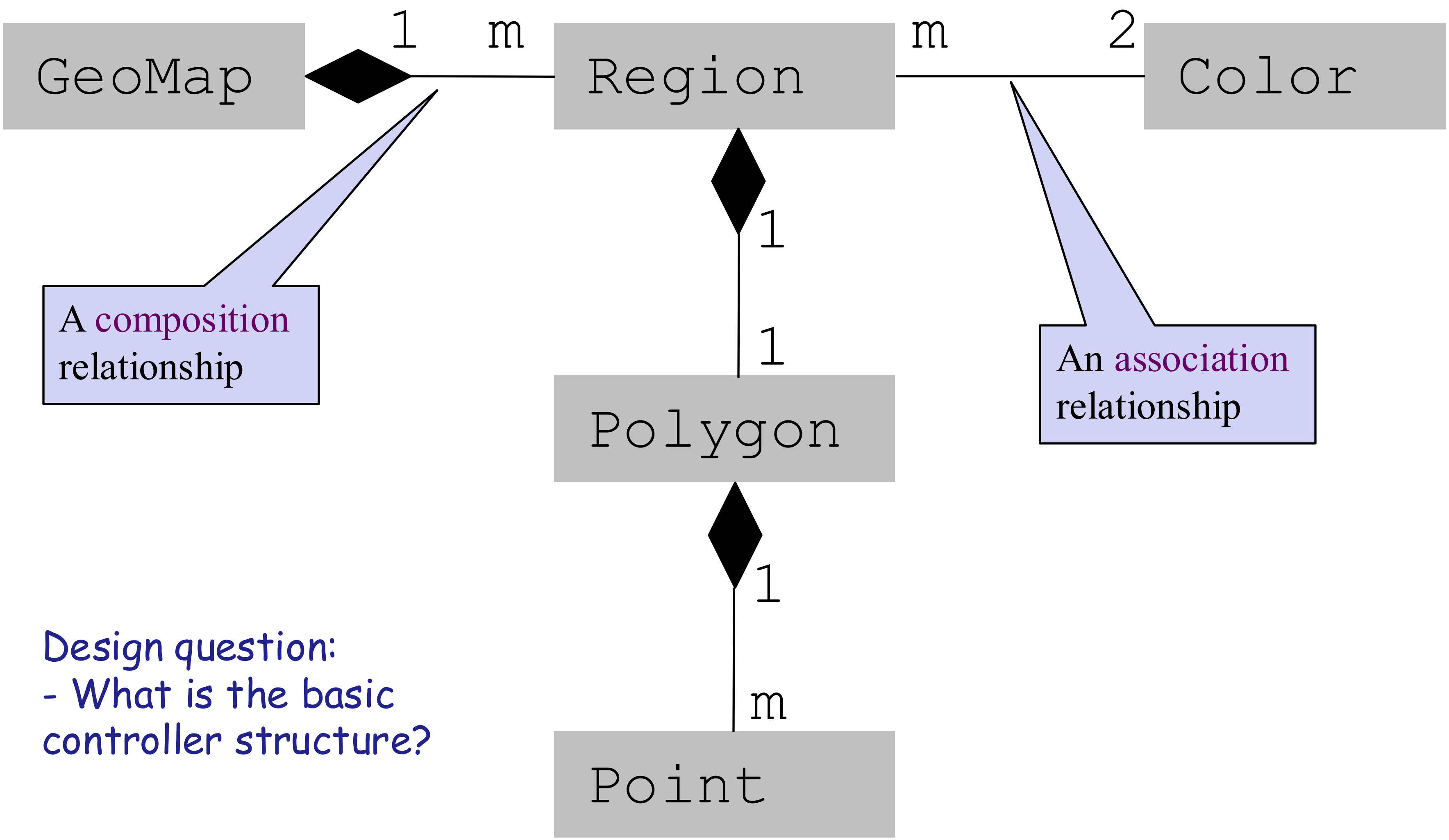


Polygon 多辺形

Major Classes and Relationship



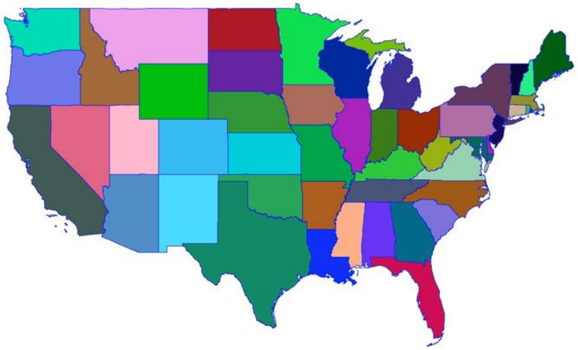
Major Classes and Relationship

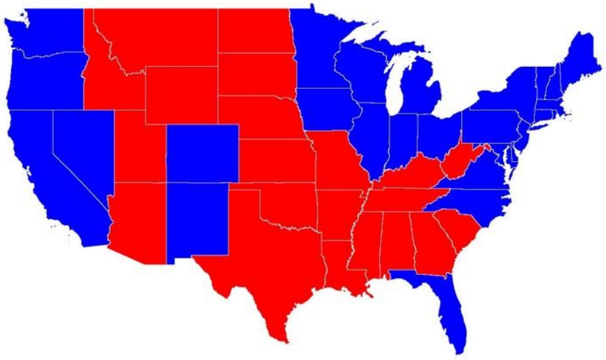


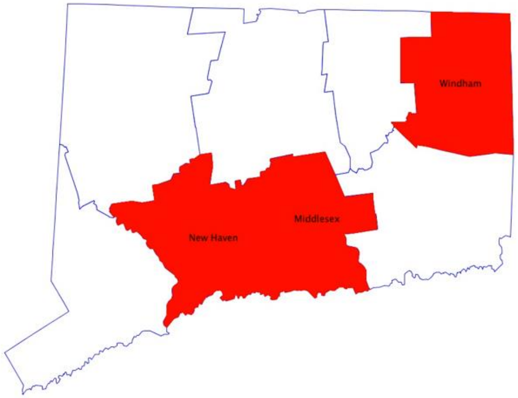
Design question:
- What is the basic controller structure?

Coloring Controller Structure

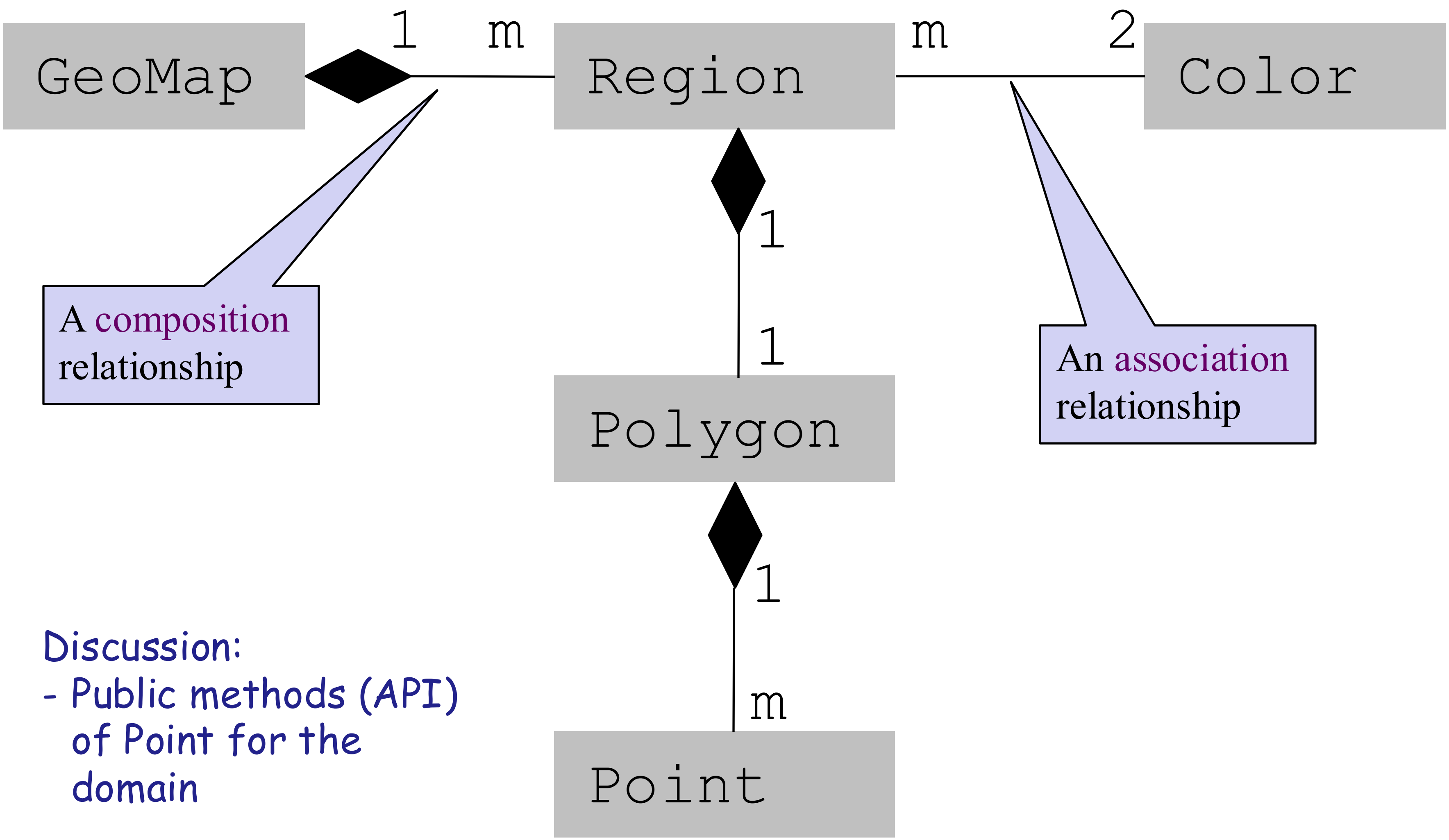
- Retrieve(检索) region (standard)
  - Batch: retrieve a list containing all regions
  - Specific: retrieve one specific region (e.g., the one being clicked)
- Coloring (customized)
  - Map properties of each region to a color





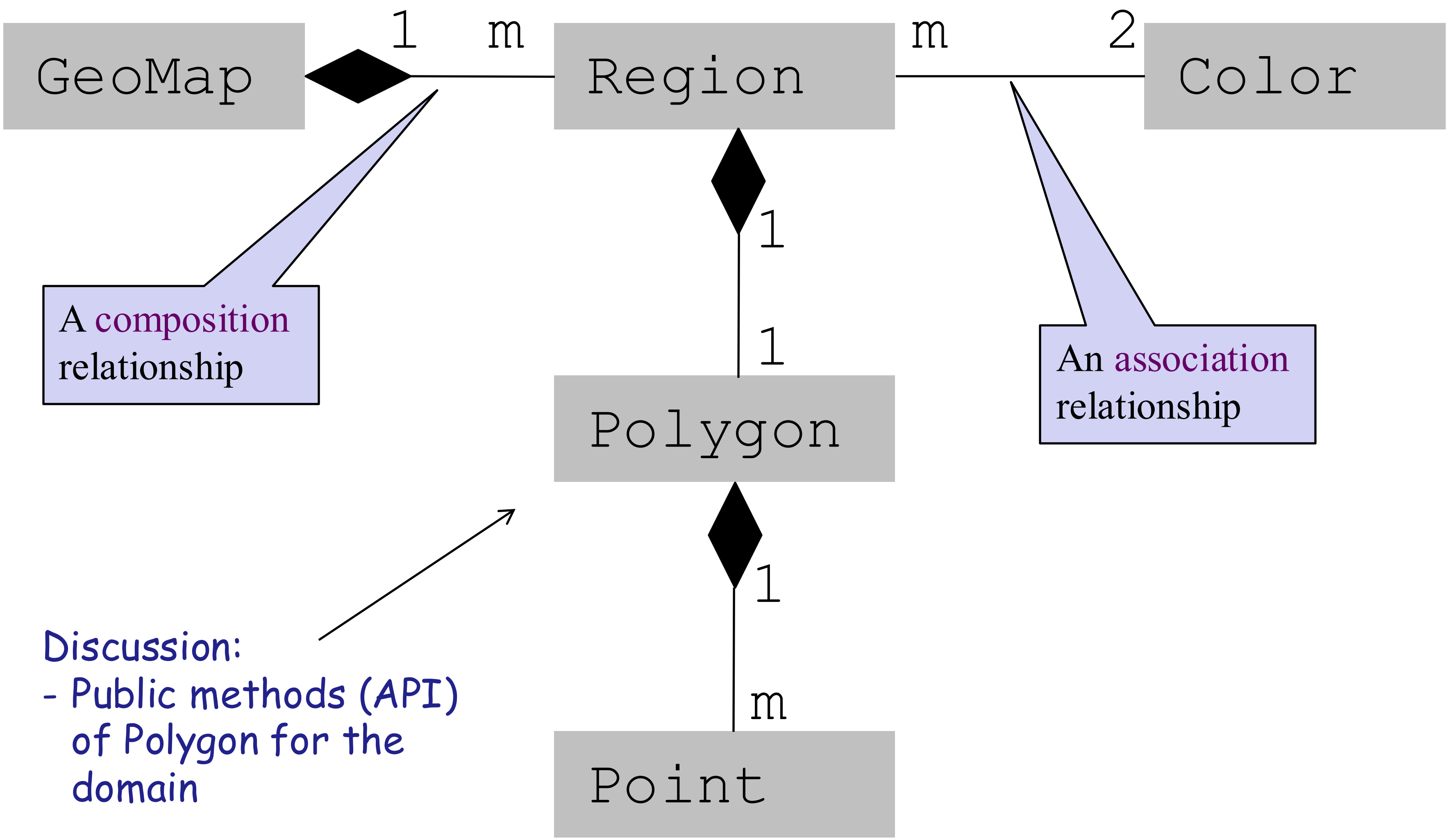


Major Classes and Relationship



Discussion:
- Public methods (API) of Point for the domain

Major Classes and Relationship



Discussion:
- Public methods (API) of Polygon for the domain

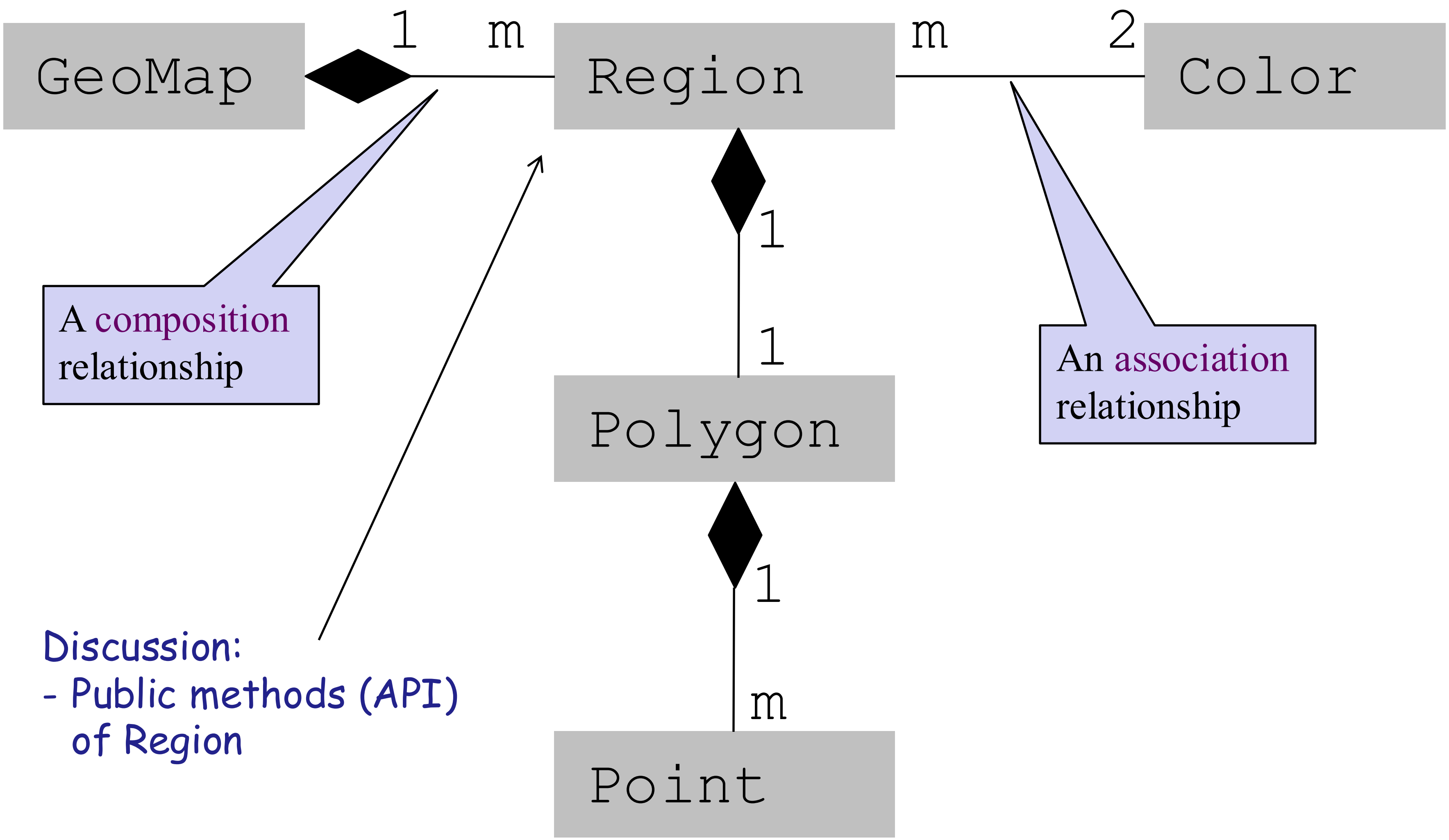
Polygon

```
public class Polygon {
   private final int N;            // number of boundary points
   private final Point[] points;   // the points

   // read from input stream
   public Polygon(Scanner input) {
        N = input.nextInt();
        points = new Point[N+1];
        for (int i = 0; i < N; i++) {
           points[i] = new Point ( input );
        }
        points[N] = points[0];
   }

   …
   public void draw() { … }
   public void fill() { … }
   public boolean contains(Point p) { … }
   public Point centroid() { … }
   …
}
```

Major Classes and Relationship



Discussion:
- Public methods (API) of Region

Region

Q: Should Region have a method that returns its internal Polygon?

```
public class Region {
   private final String  regionName;   // name of region
   private final String  mapName;
   private final Polygon poly;         // polygonal boundary
   private Color fillColor, drawColor;

   public Region(String mName, String rName, Polygon poly) {
      regionName = rName;
      mapName = mName;
      this.poly = poly;
      setDefaultColor();
   }
   public void setDrawColor (Color c) { drawColor = c; }
   public void draw() { setDrawColor(); poly.draw (); }
   public void fill() { … }
   public boolean contains(Point p) {
      return poly.contains(p);
   }
   public Point centroid() { return poly.centroid() }
   …
}
```

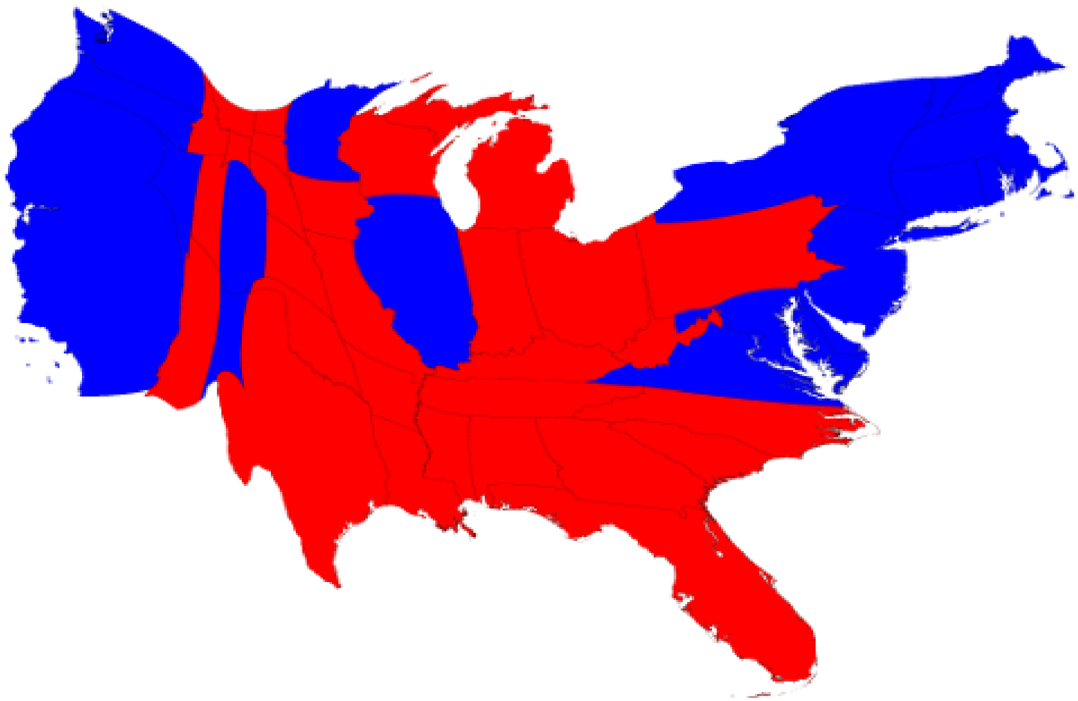
Even though most complexity is in Polygon, Polygon is not exposed. Region **delegates (委托)** tasks internally to Polygon.

Example Controllers

- GeoMap.java
- RandomColorMap.java
- ClickColorMap.java
- RedBlueMap.java

Cartograms

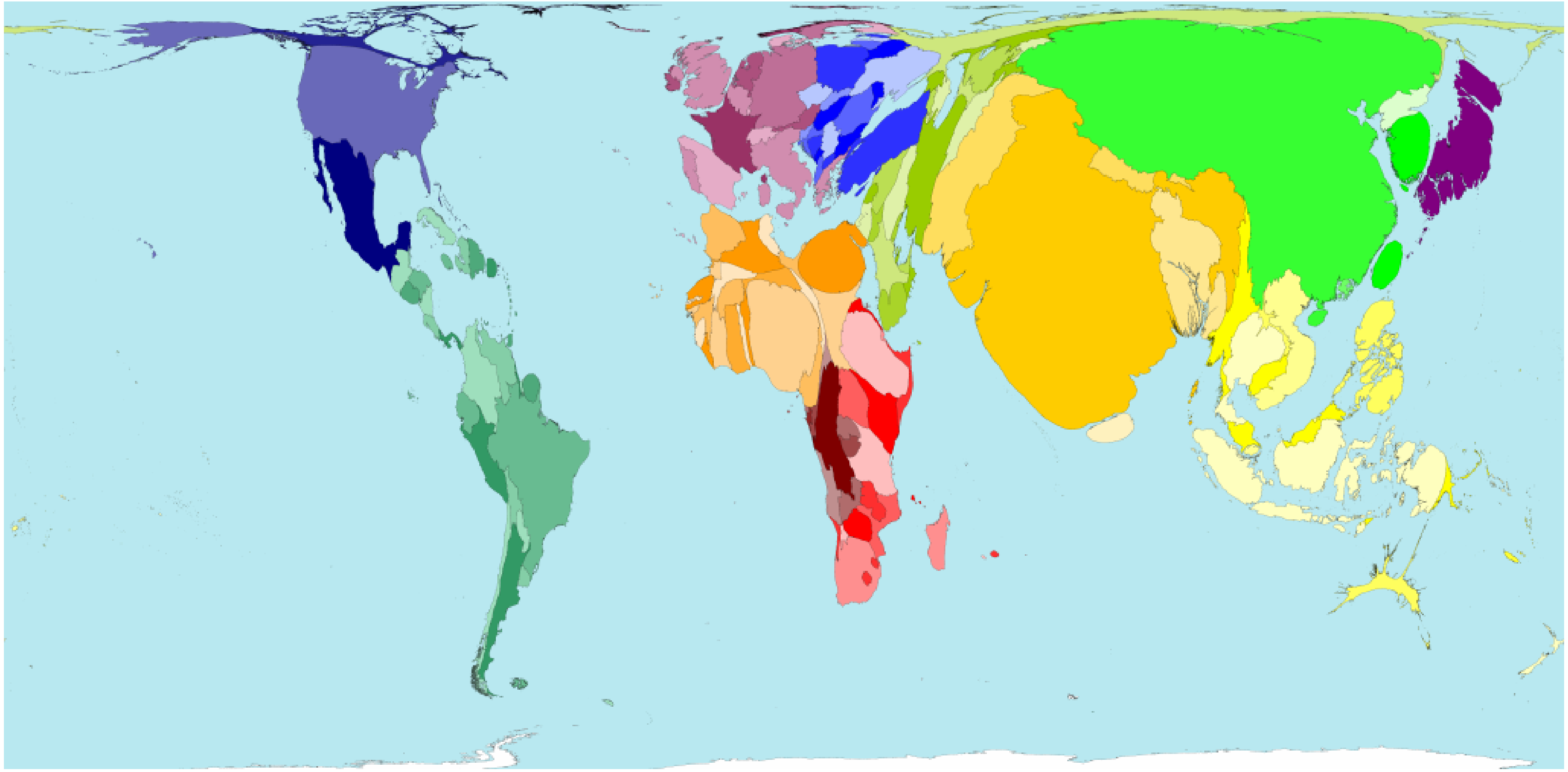
- Cartogram. Area of state proportional to number of electoral votes.



Michael Gastner, Cosma Shalizi, and Mark Newman
http://www-personal.umich.edu/~mejn/election/2016/

Cartograms

- Cartogram. Area of country proportional to population.



Outline

- Admin and recap
- Defining classes
- Object-oriented design
  - Composition (has)/association relationship and geo visualization
  - *Inheritance(继承) relationship*

Inheritance

- **Inheritance**: Reuse classes by deriving a new class from an existing one
  - The existing class is called the parent class, or superclass, or base class
  - The derived class is called the *child class* or *subclass*.

- As the name implies, the child inherits characteristics of the parent
  - The child class *inherits* every method and every data field defined for the parent class

Visualize Inheritance

- The child class *inherits* *all* methods and data defined for the parent class