CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

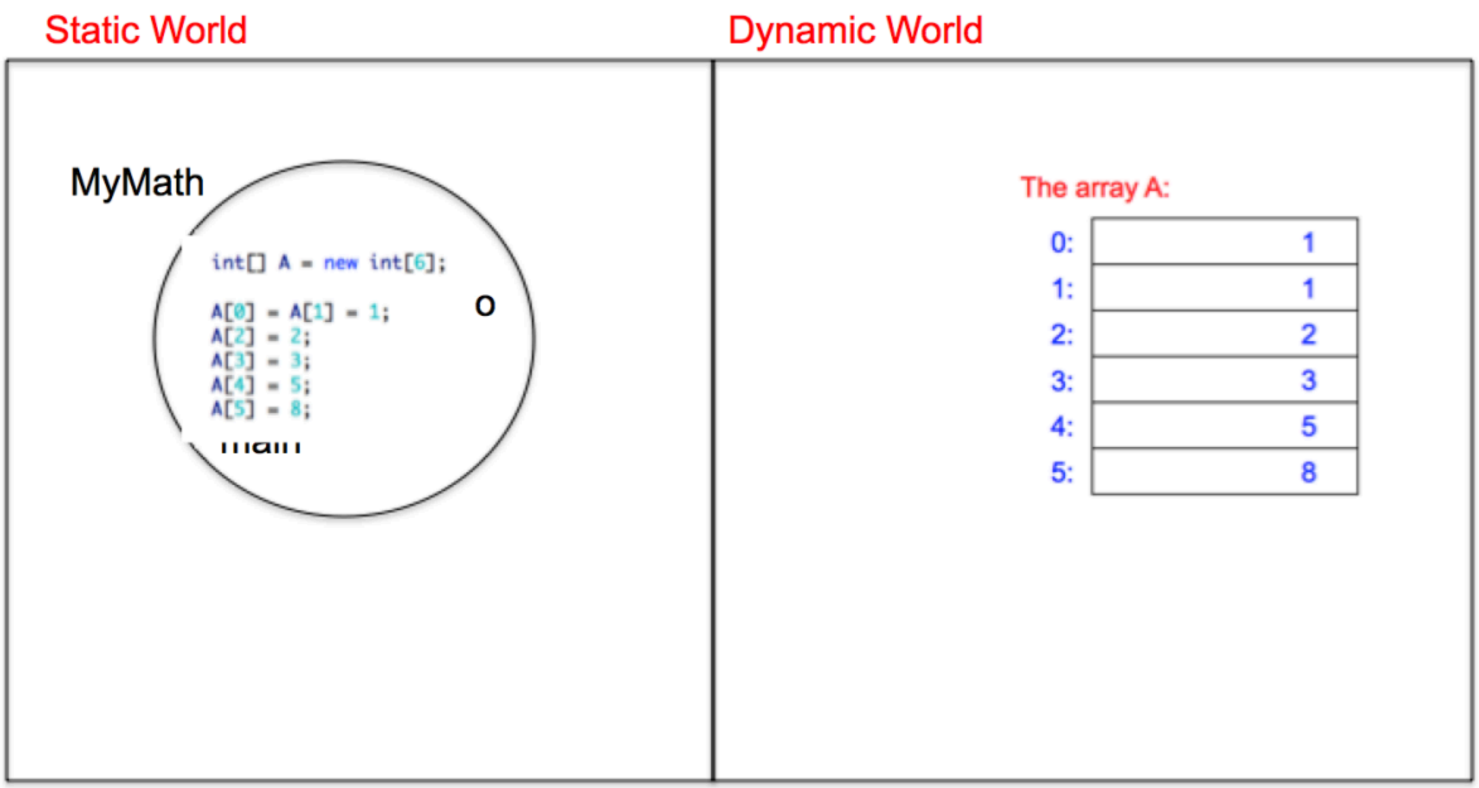
Today:

- Creating and Using Objects;
- public vs private;
- Object-Oriented Programming;
- Abstract data types

Creating and Using Objects

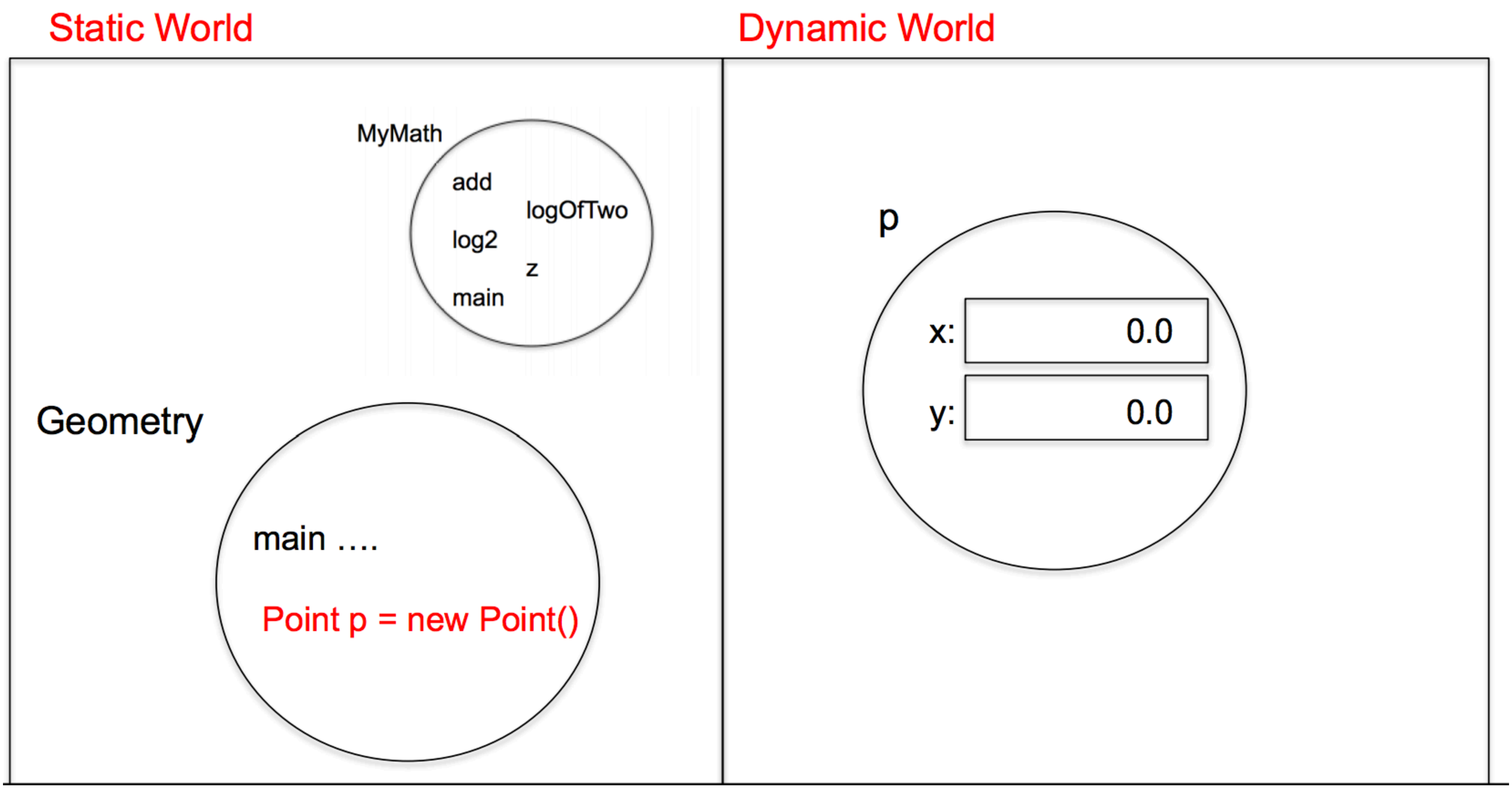
Computer Science

Creating Objects: Recall that when we declare an array in a method, we are creating a new object that lives in dynamic memory:



Creating and Using Objects

But, we can create new "classes" which contain members just like static classes…..



Creating and Using Objects

But, we can create new "classes" which contain members just

like static classes.....

```
> run Geometry
p =  (2.3,4.5)
```

CS112Homework

Geometry.java

```java
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point();
        p.x = 2.3;
        p.y = 4.5;

       System.out.println("p =  (" + p.x + "," + p.y + ")");

    }
}
```

MyMath.java

```java
public class MyMath {

    static double add(double x, double y) {
        return (x + y);
    }

    static final double logOfTwo = Math.log(2.0);

    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }

    static double  z = 8.0;    // just an example

    public static void main(String[] args) {

       System.out.println("add(2,3) => " + add(2,3));
       System.out.println("log2(8.0) => " + log2(z));

    }
}
```

ClientOfMyMath.java

```java
public class ClientOfMyMath {

    public static void main(String[] args) {

        System.out.println("MyMath.add(2,3) => "
                             + MyMath.add(2,3));
        System.out.println("MyMath.log2(8.0) => "
                             + MyMath.log2(MyMath.z));
        System.out.println("MyMath.logOfTwo => "
                             + MyMath.logOfTwo);

    }
}
```

Point.java

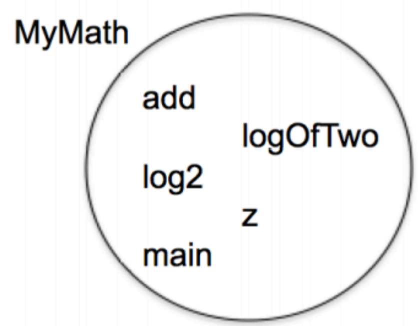
Note: no `static` keyword

```java
public class Point {

    double x = 0.0;          // create two doubles
    double y = 0.0;          // and initialize to 0.0
}
```

Creating and Using Objects

But, we can create new “classes” which contain members just like static classes…..

Static World



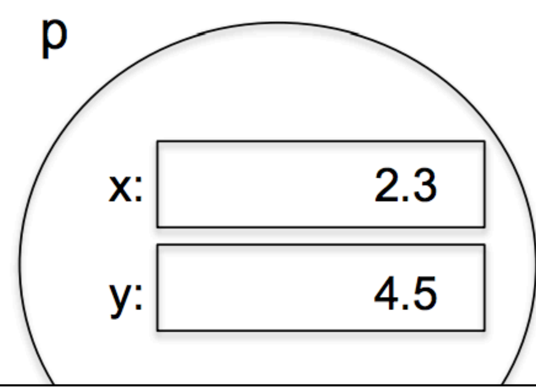
Remember: To access the members of a static class, use

name-of-class . name-of-member

for example:

MyMath.logOfTwo

Dynamic World



Remember: To access the members of an object, use the variable you assigned it to:

variable-name . name-of-member

for example: p.x

Boston University Computer Science

We can initialize the fields in an object using declarations:

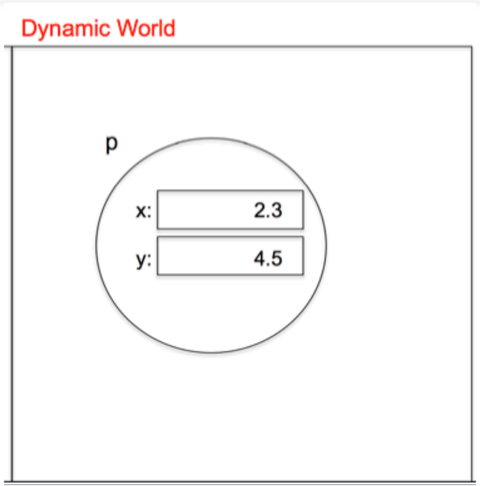
CS112Homework

Point.java

```java
public class Point {

    double x = 2.3;
    double y = 4.5;

}
```

Geometry.java

```java
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point();

        System.out.println("(" + p.x + "," + p.y + ")");

    }
}
```




```
> run Geometry
(2.3,4.5)
>
```

Creating and Using Objects

But then all objects created will have the SAME initial values; it is better to use a constructor, which allows you to create different initial values in client program:

CS112Homework

Point.java

```
public class Point {

    double x;
    double y;

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

}
```

Geometry.java

```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(2.3,4.5);

        System.out.println("(" + p.x + "," + p.y + ")");

    }
}
```

```
> run Geometry
(2.3,4.5)
>
```

Creating and Using Objects

But then all objects created will have the SAME initial values; it is better to use a constructor, which allows you to create different initial values in client program:

CS112Homework

Point.java

```
public class Point {

    double x;
    double y;

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

}
```

Geometry.java

```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(-1.9, 10.3);

        System.out.println("(" + p.x + "," + p.y + ")");

    }
}
```

```
> run Geometry
(-1.9,10.3)
>
```

Creating and Using Objects

Computer Science

But then all objects created will have the SAME initial values; it is better to use a constructor, which allows you to create different initial values:

keyword new to create new object

CS112Homework

Point.java

NO result type

name of class

```
public class Point {

    double x;
    double y;

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

}
```

Client calls constructor with initial values

Geometry.java

```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(2.3,4.5);

        System.out.println("(" + p.x + "," + p.y + ")");

    }
}
```

keyword `public`

refer to field using `this.`

```
> run Geometry
(2.3,4.5)
>
```

You can also create multiple instances of the object, with different initial values:

CS112Homework

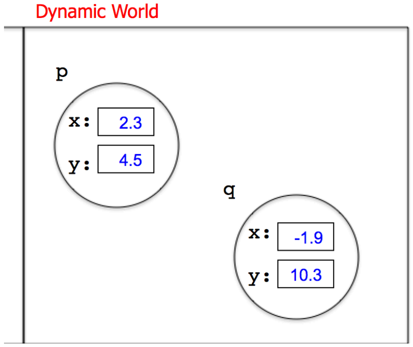
Point.java

```java
public class Point {

    double x;
    double y;

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

}
```

Geometry.java

```java
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(2.3, 4.5);

        System.out.println("(" + p.x + "," + p.y + ")");

        Point q = new Point(-1.9, 10.3);

        System.out.println("(" + q.x + "," + q.y + ")");

    }
}
```

Dynamic World




```
> run Geometry
(2.3,4.5)
(-1.9,10.3)
```

Creating and Using Objects

You can even create an array of objects:

CS112Homework

Point.java

```
public class Point {

    double x;
    double y;

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

}
```

Geometry.java

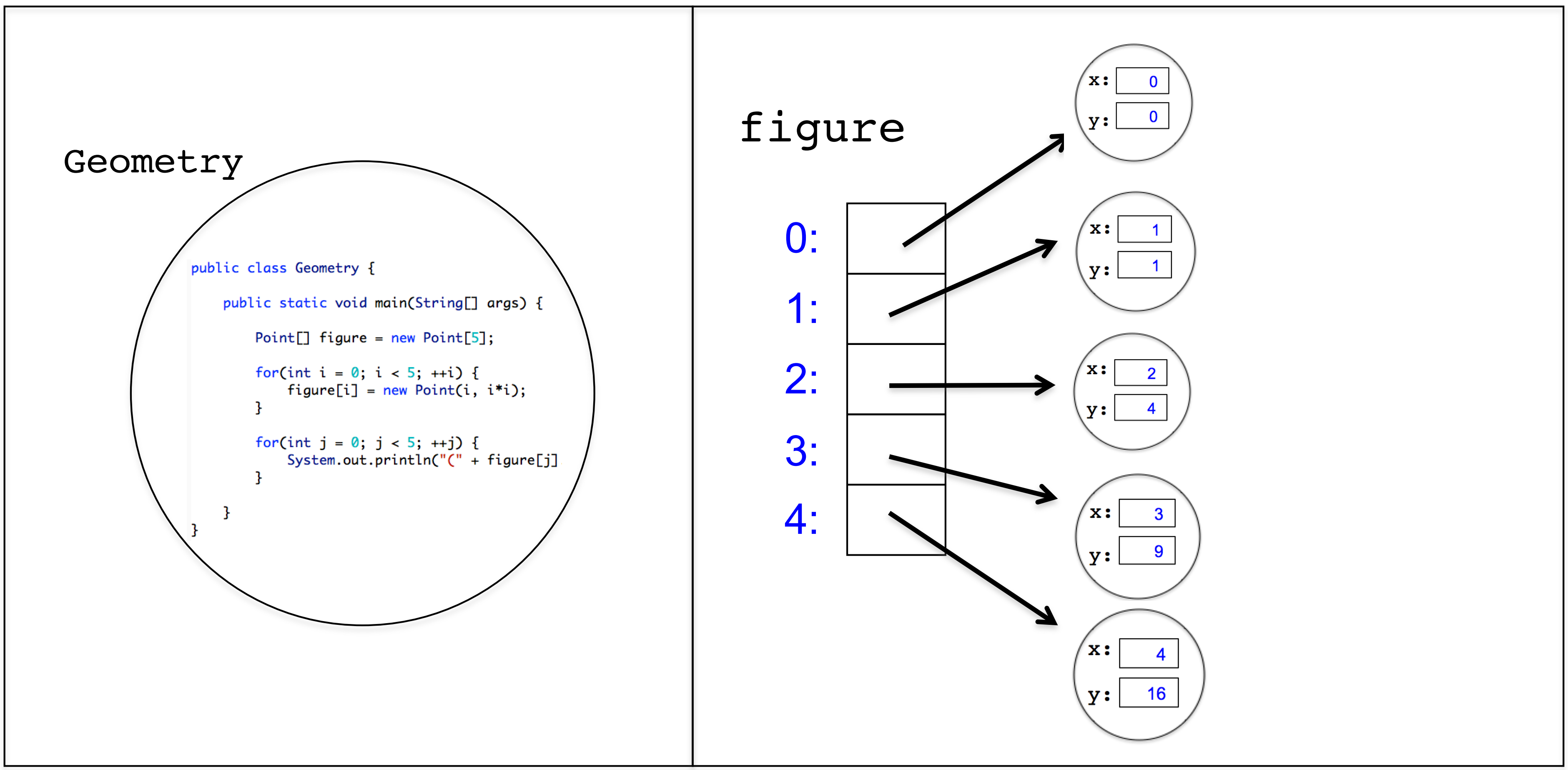
```
public class Geometry {

    public static void main(String[] args) {

        Point[] figure = new Point[5];

        for(int i = 0; i < 5; ++i) {
            figure[i] = new Point(i, i*i);
        }

        for(int j = 0; j < 5; ++j) {
            System.out.println("(" + figure[j].x + "," + figure[j].y + ")");
        }

    }
}
```

```
> run Geometry
(0.0,0.0)
(1.0,1.0)
(2.0,4.0)
(3.0,9.0)
(4.0,16.0)
```

Creating and Using Objects

Static World

Dynamic World



Creating and Using Objects

Computer Science

You can overload the constructor method, providing different versions that behave in different ways – all of which create a new object.

CS112Homework

Point.java

```
public class Point {

    public double x;
    public double y;

    public Point() {
        this.x = 0.0;
        this.y = 0.0;
    }

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

    public Point(double y) {
        this.x = 0.0;
        this.y = y;
    }

}
```

Geometry.java

```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point();
        System.out.println("p = (" + p.x + "," + p.y + ")" );

        Point q = new Point(2.3, 4.5);
        System.out.println("q = (" + q.x + "," + q.y + ")" );

        Point r = new Point(10.3);
        System.out.println("r = (" + r.x + "," + r.y + ")" );

    }
}
```

```
> run Geometry
p = (0.0,0.0)
q = (2.3,4.5)
r = (0.0,10.3)
>
```

You can overload the constructor method, providing different versions that behave in different ways – all of which create a new object.

If you do not provide a constructor, a default constructor is provided which simply initializes the fields to their default values.

CS112Homework

Point.java

```
public class Point {

    public double x;
    public double y;

}
```

Geometry.java

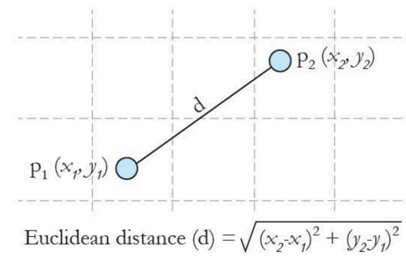
```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point();
        System.out.println("p = (" + p.x + "," + p.y + ")" );

    }
}
```

```
> run Geometry
p = (0.0,0.0)
>
```

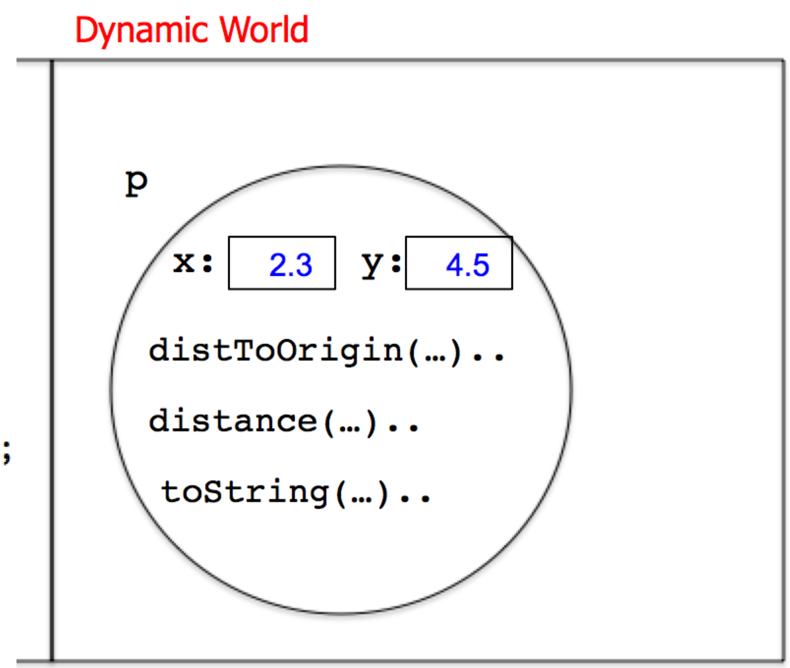
Creating and Using Objects

Computer Science

Finally, you can include (local) methods inside objects, and these methods can access fields in the objects:

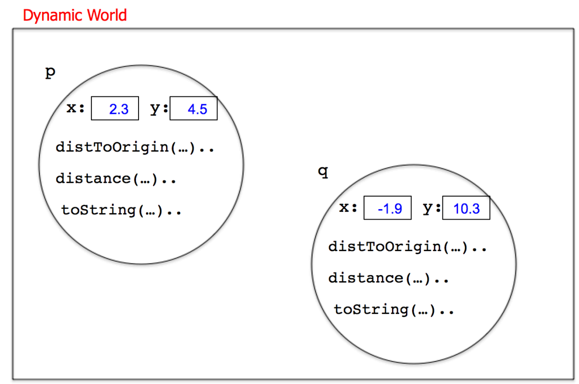



```
public class Point {

    public double x;
    public double y;

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

    public double distToOrigin() {
        return Math.sqrt((x * x) + (y * y));
    }

    public double distance(Point q) {
        double xDist = x - q.x;
        double yDist = y - q.y;
        return Math.sqrt((xDist * xDist) + (yDist * yDist));
    }

    public String toString() {
        return "(" + x + "," + y + ")";
    }

}
```



Creating and Using Objects

Finally, you can include (local) methods inside objects, and these methods can access fields in the objects:




```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(2.3,4.5);
        Point q = new Point(-1.9,10.3);

        System.out.println("Distance from p to origin = " + p.distToOrigin() );
        System.out.println("Distance from p to q = " + p.distance( q ) );
        System.out.println(p);
        System.out.println("Distance from " + p + " to " + q + " = " + p.distance( q ) );

    }
}
```

```
public class Point {

    public double x;
    public double y;

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

    public double distToOrigin() {
        return Math.sqrt((x * x) + (y * y));
    }

    public double distance(Point q) {
        double xDist = x - q.x;
        double yDist = y - q.y;
        return Math.sqrt((xDist * xDist) + (yDist * yDist));
    }

    public String toString() {
        return "(" + x + "," + y + ")";
    }

}
```

```
> run Geometry
Distance from p to origin = 3.7802116342871597
Distance from p to q = 6.479197481170026
(2.3,4.5)
Distance from (2.3,4.5) to (-1.9,10.3) = 6.479197481170026
```

NOTE in particular the method `toString()`, which can be included in any object. When the name of the object is used in a context where a String is expected, it will use the String you return in the method. This is how you display a String representation of the object, especially useful for debugging!

```
public class Point {

    double x;
    double y;

    .......

    public String toString() {
        return "(" + x + "," + y + ")";
    }

}
```

```
public class Geometry {

    public static void main(String[] args) {

        Point p = new Point(2.3,4.5);
        Point q = new Point(-1.9,10.3);
        ......

        System.out.println(p);
        System.out.println("Distance from " + p + " to " + q + " = " + p.distance( q ) );

    }
}
```

```
> run Geometry
Distance from p to origin = 3.7802116342871597
Distance from p to q = 6.479197481170026
(2.3,4.5)
Distance from (2.3,4.5) to (-1.9,10.3) = 6.479197481170026
```

Scope revisited: public and private members of classes

Recall that the scope of a declaration is the region of the program where the declaration has meaning; we have seen two different rules for scope:

```
public class Point {

    public double x;
    public double y;

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

    public double distToOrigin() {
        return Math.sqrt((x * x) + (y * y));
    }

    public double distance(Point q) {
        double xDist = x - q.x;
        double yDist = y - q.y;
        return Math.sqrt((xDist * xDist) + (yDist * yDist));
    }

    public String toString() {
        return "(" + x + "," + y + ")";
    }

}
```

Scope of field x is whole class

Scope of local variable xDist is until end of closest enclosing block { … }

Scope revisited: public and private members of classes

Computer Science

But scope also applies in the larger context of files and directories and the whole computer memory! There are two keywords we will use to define the scope of the members of a class: public and private:

Can only refer to from inside class.

```
public class MyMath {

    public static double add(double x, double y) {
        return (x + y);
    }

    private static final double logOfTwo = Math.log(2.0);

    public static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }

    public static double  z = log2(256.0);     // just an example

    public static void main(String[] args) {

        System.out.println("add(2,3) => " + add(2,3));
        System.out.println("log2(8.0) => " + log2(z));

    }
}
```

Can refer to from anywhere!

The scope of a private member of a class is only the inside the class itself;

The scope of a public member is the whole computer: any piece of code can access the member using either:

name-of-class . member

variable-name . member

Scope revisited: public and private members of classes

But scope also applies in the larger context of files and directories and the whole computer memory! There are two keywords we will use to define the scope of the members of a class: public and private:

The scope of a private member of a class is only the inside the class itself; this is the rule we learned previously for scope of members.

The scope of a public member is the whole computer: any piece of code can access the member using either:

name-of-class . member

variable-name . member

LETS' GO TO DR. JAVA TO SEE HOW THIS WORKS…..

Our goals in writing software include the following:

- The program should be **correct** and as **efficient** as possible;
- You should understand and improve (whenever you can) its behavior in the **worst case** and in the **average case**, both analytically and in practice;
- Your code should be **robust** in that users can not misuse it ("defensive programming");
- You (and, sometimes, your team) should **develop** the program as **easily** as you can (while observing the first goal!);
- When appropriate, you should **reuse** existing code and produce new code which can be reused easily later (by you or others). When using others' code, cite the source!
- You (or someone else) should be able to quickly **understand** the program when you look at it years later, and to **modify** and **maintain** it easily.

Our goals in writing software include the following:

- The program should be **correct** and as **efficient** as possible; be obsessive!
- You should understand (whenever you can) its behavior in the **worst case** and in the **average case**; be pessimistic!
- Your code should be **robust** in that users can not misuse it ("defensive programming"); assume everyone else is stupid!
- You (and, sometimes, your team) should **develop** the program as **easily** as you can (while observing the first goal!); be lazy!
- When appropriate, you should **reuse** existing code and produce new code which can be reused easily later (by you or others); even more lazy!
- You (or someone else) should be able to quickly **understand** the program when you look at it years later, and to **modify** and **maintain** it easily; assume others are stupid and lazy!

The principles of **Software Engineering** (e.g., CS 411) help us poor programmers achieve these goals:

Object-Oriented Design: break your problem (and its solution) into manageable-sized pieces---we'll talk about this in the rest of this lecture;

Abstraction: Simplify and generalize---solve the most general problem---we'll talk about this when we study Generics;

Step-wise Refinement: Develop your code a piece at a time, testing for correctness as you go along---**we'll be developing this skill throughout the semester!**

ALL of these principles will help you become excellent Java programmers by the end of CS 112!

Object-Oriented Design

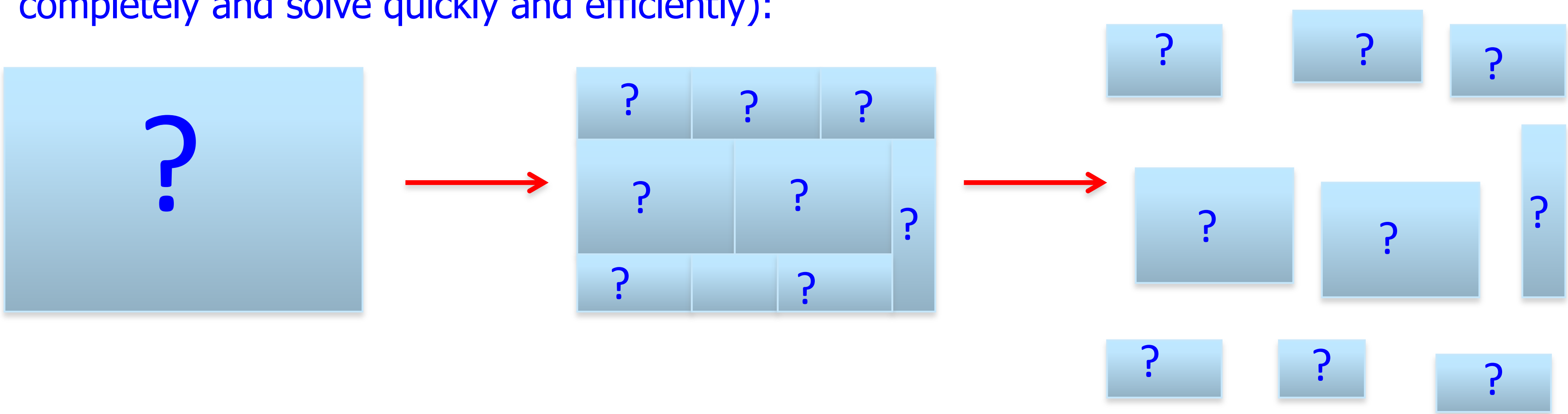
Computer Science

The basic goal of **Object-Oriented Design** is to control the **complexity** of software development, and it can be summed up in one phrase:

Object-Oriented Design

Divide and Conquer means just what it says:

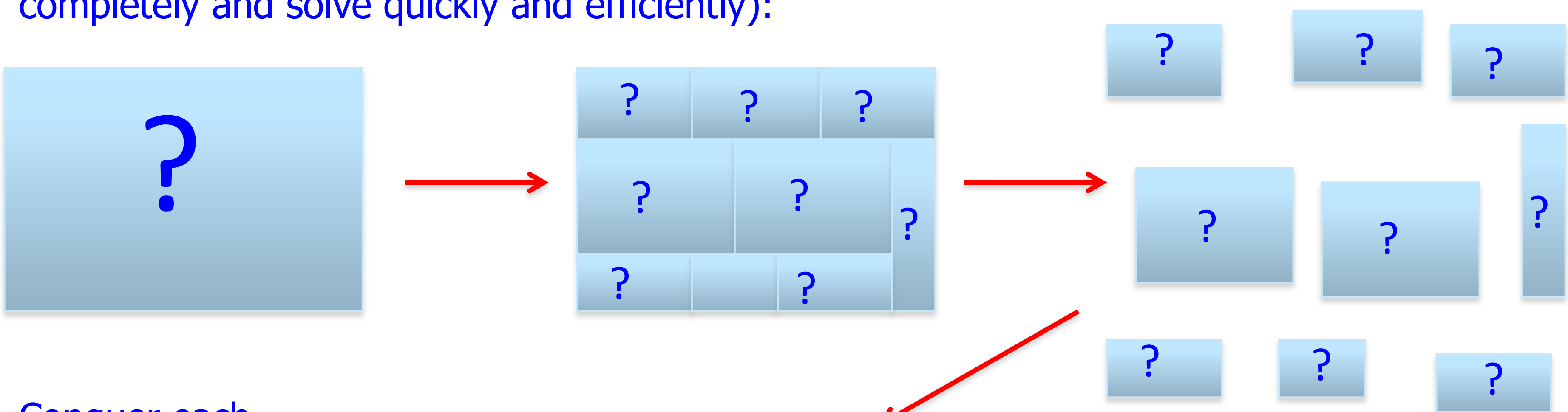
Divide the **problem** into manageable pieces (small enough for one person to understand completely and solve quickly and efficiently):



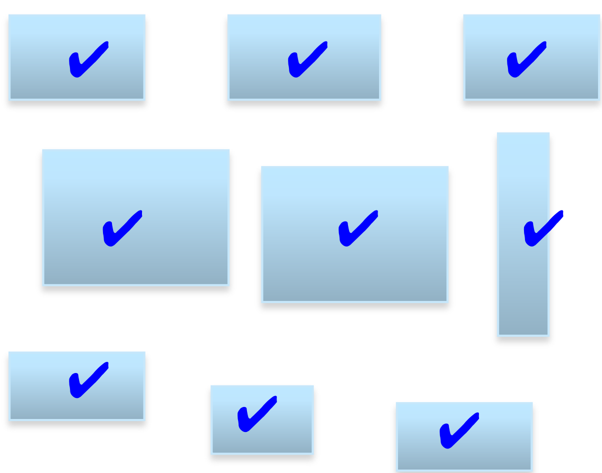
Object-Oriented Design

Divide and Conquer means just what it says:

Divide the **problem** into manageable pieces (small enough for one person to understand completely and solve quickly and efficiently):



Conquer each problem separately:

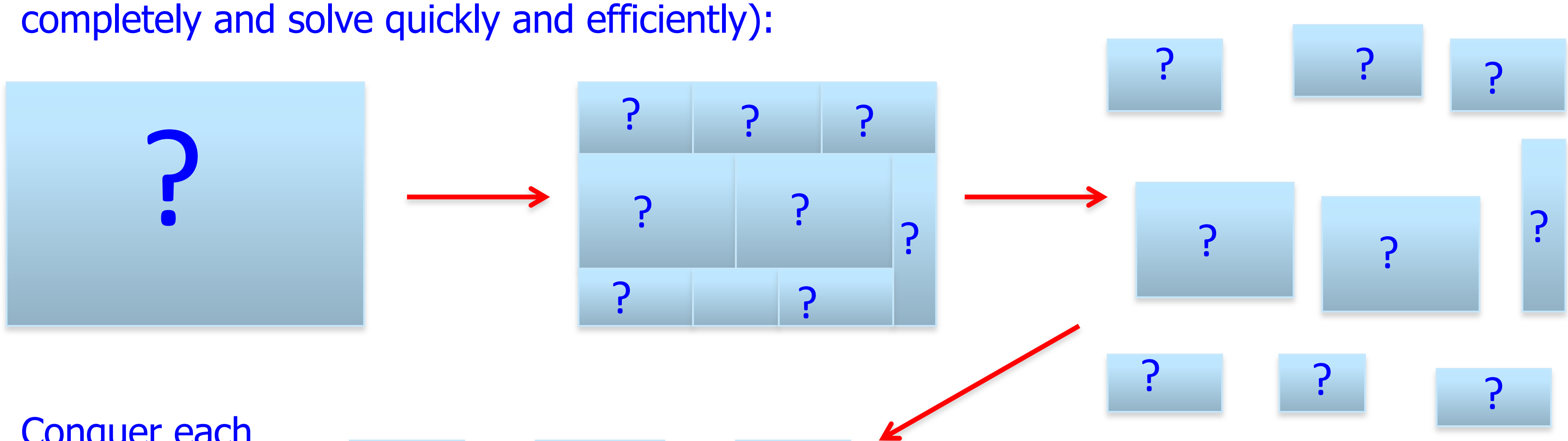


Object-Oriented Design

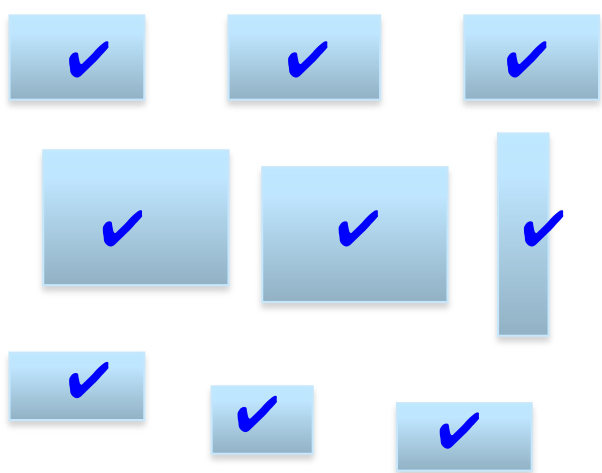
Computer Science

Divide and Conquer means just what it says:

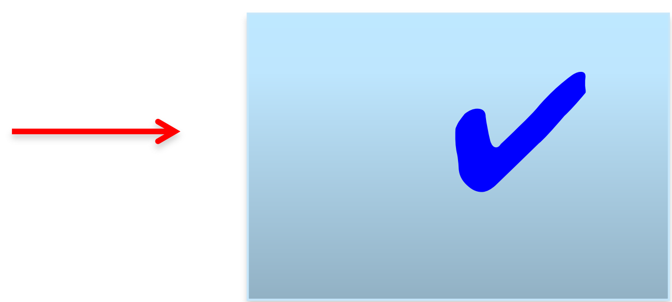
Divide the **problem** into manageable pieces (small enough for one person to understand completely and solve quickly and efficiently):



Conquer each problem separately:

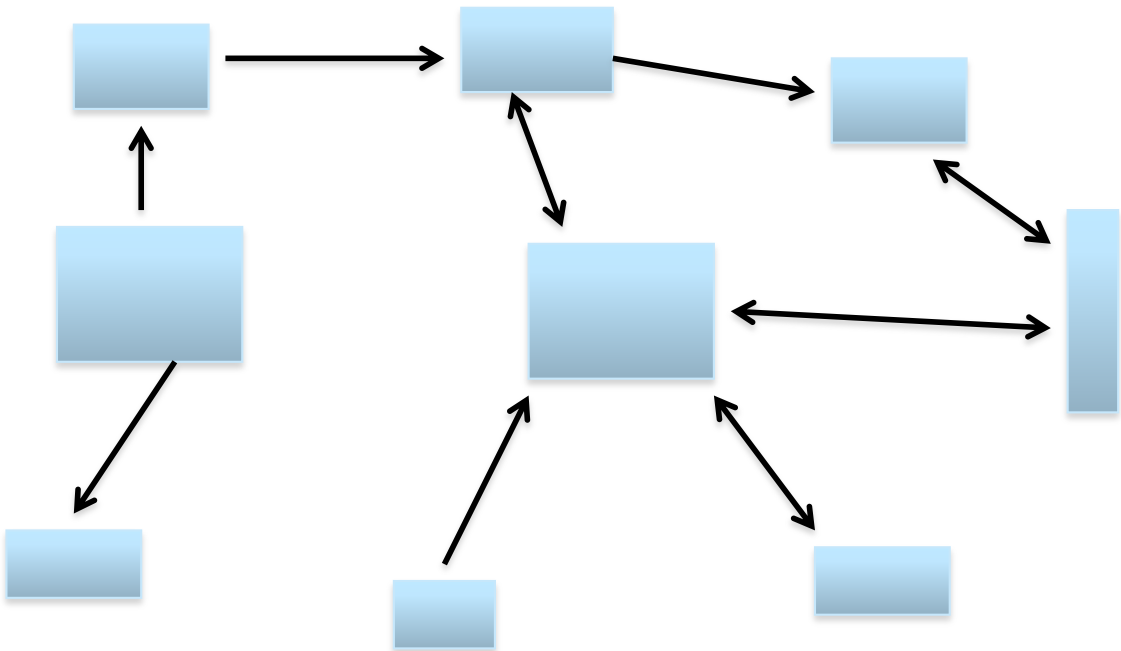


And then put it back together:



Object-Oriented Design

Critical to this process is the **interaction** between the parts of the solution:



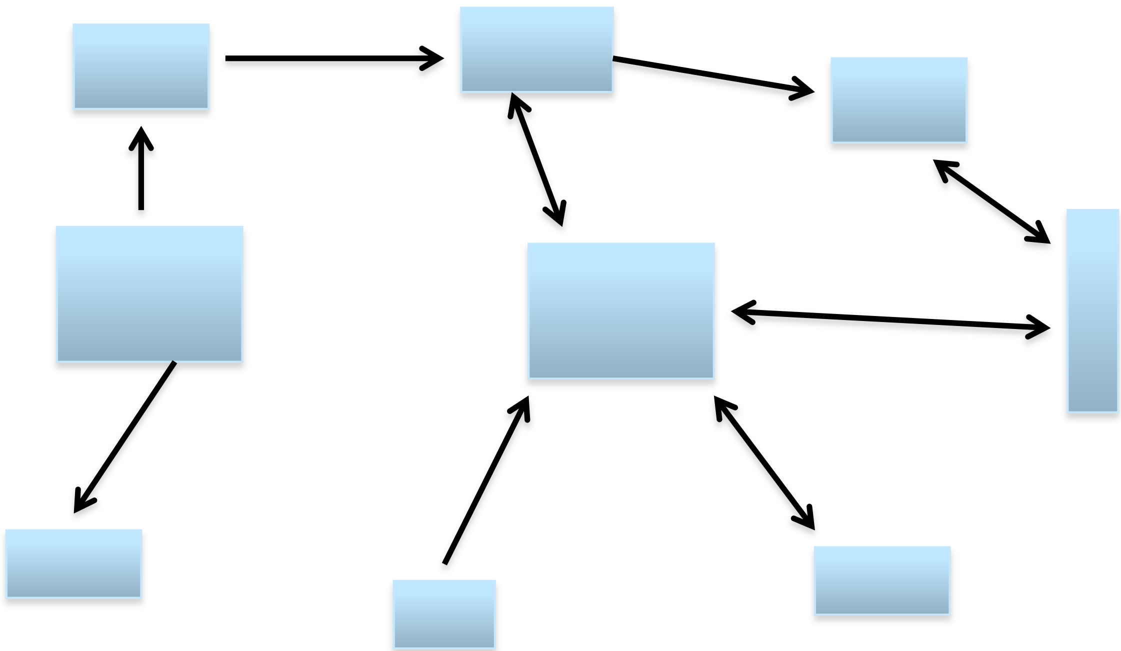
Each part may be simple, but if the communication between the parts is complex, the whole thing will still be too difficult to understand! Make the parts simple and their interaction simple!

Question: If you have N people, how many possible conversations can you have?

Object-Oriented Design

Computer Science

Critical to this process is the **interaction** between the parts of the solution:



Each part may be simple, but if the communication between the parts is complex, the whole thing will still be too difficult to understand! Make the parts simple and their interactions simple!

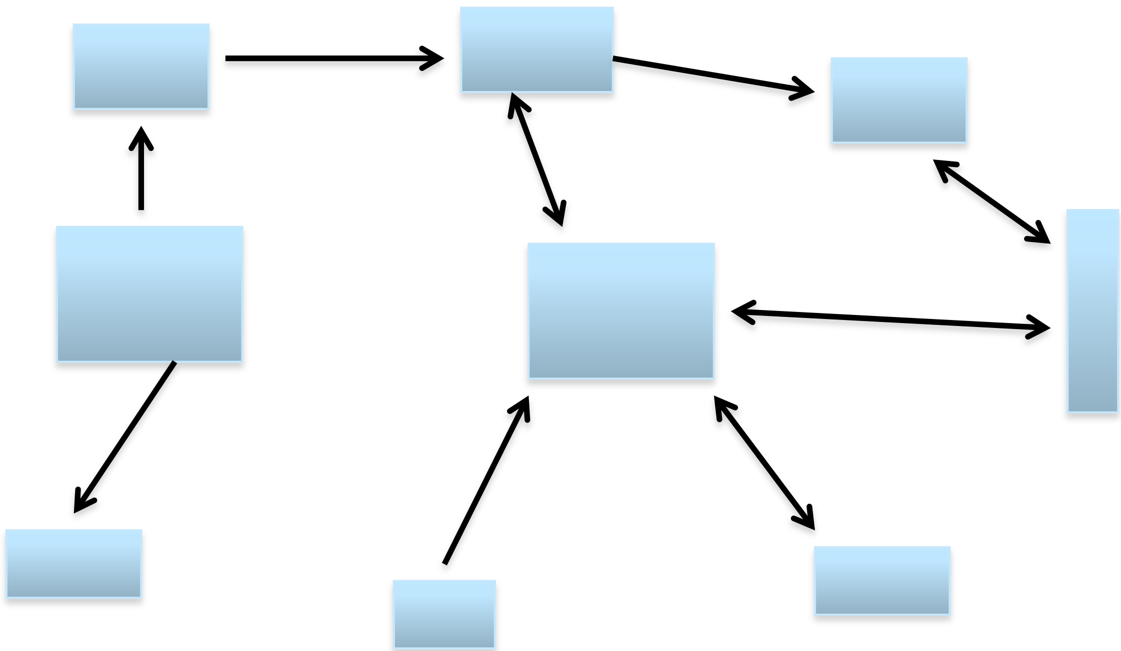
Question: If you have N parts, how many possible connections can you have?

Answer: 1 -> 0, 2 -> 1, 3 -> 3, 4 -> 6, ..., N -> 1+ 2 + ... + N-1 = N(N-1)/2 = ~ $N^2/2$ = Geometric growth!

Object-Oriented Design

Computer Science

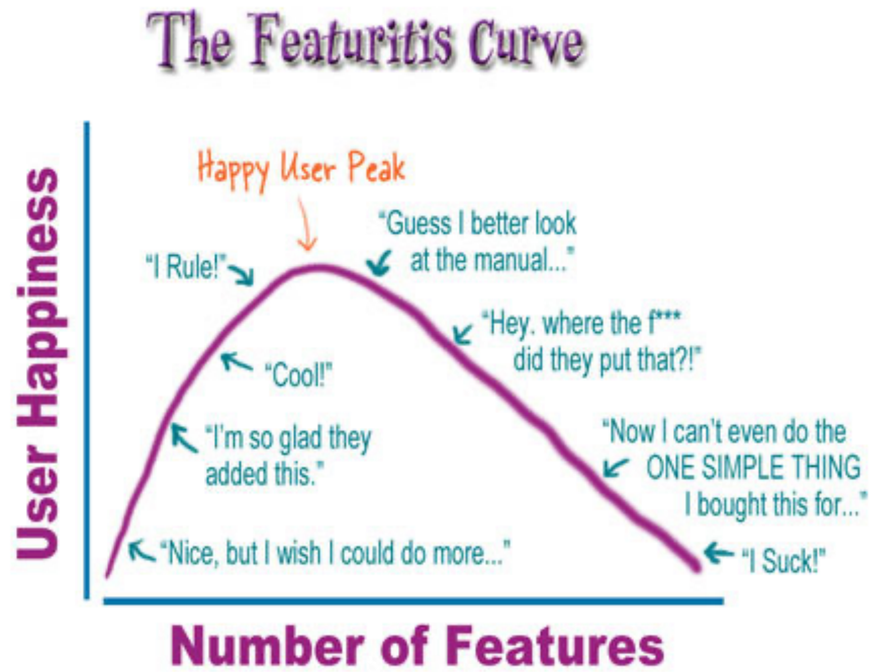
Critical to this process is the **interaction** between the parts of the solution:



Each part may be simple, but if the communication between the parts is complex, the whole thing will still be too difficult to understand! Make the parts simple and their interactions simple!

Punchline: The difficulty of communication grows geometrically as the number of parts increases. To "conquer" you must limit the number of "conversations"!

This leads to the KISS principle of system development which has many different forms:



"Less is more" – Mies van der Rohe

"Everything should be made as simple as possible, but no simpler" – A. Einstein

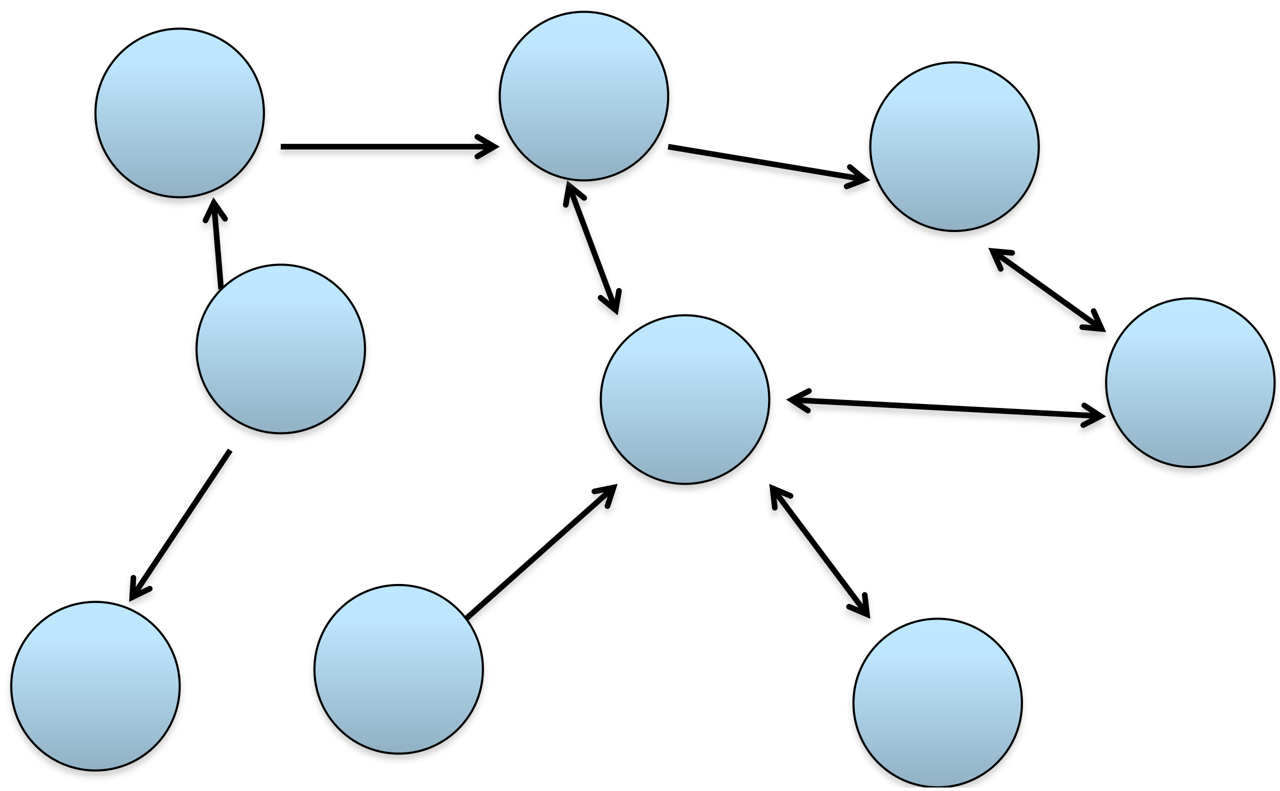
"Pluralitas non est ponenda sine necessitate" (Occam's Razor)

Object-Oriented Design

What does this mean for Java?

Part = Class

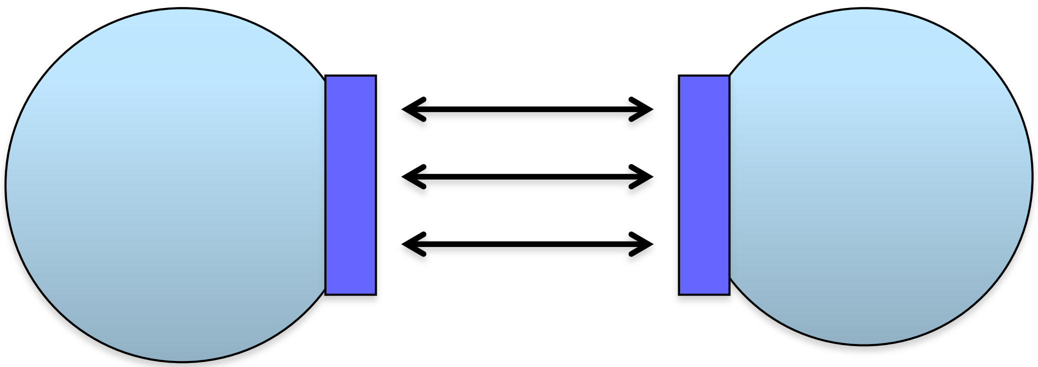
Interaction/conversation = Method call (or reference to a field)



Object-Oriented Design

The way we control communication is through the **interface** of a class:

Interface = collection of public methods and fields of a class



A class's interaction with other classes is through its interface, so:

To Keep It Simple, Stupid:

Keep the Interactions Simple, Stupid, by

Keeping the Interfaces Simple, Stupid!

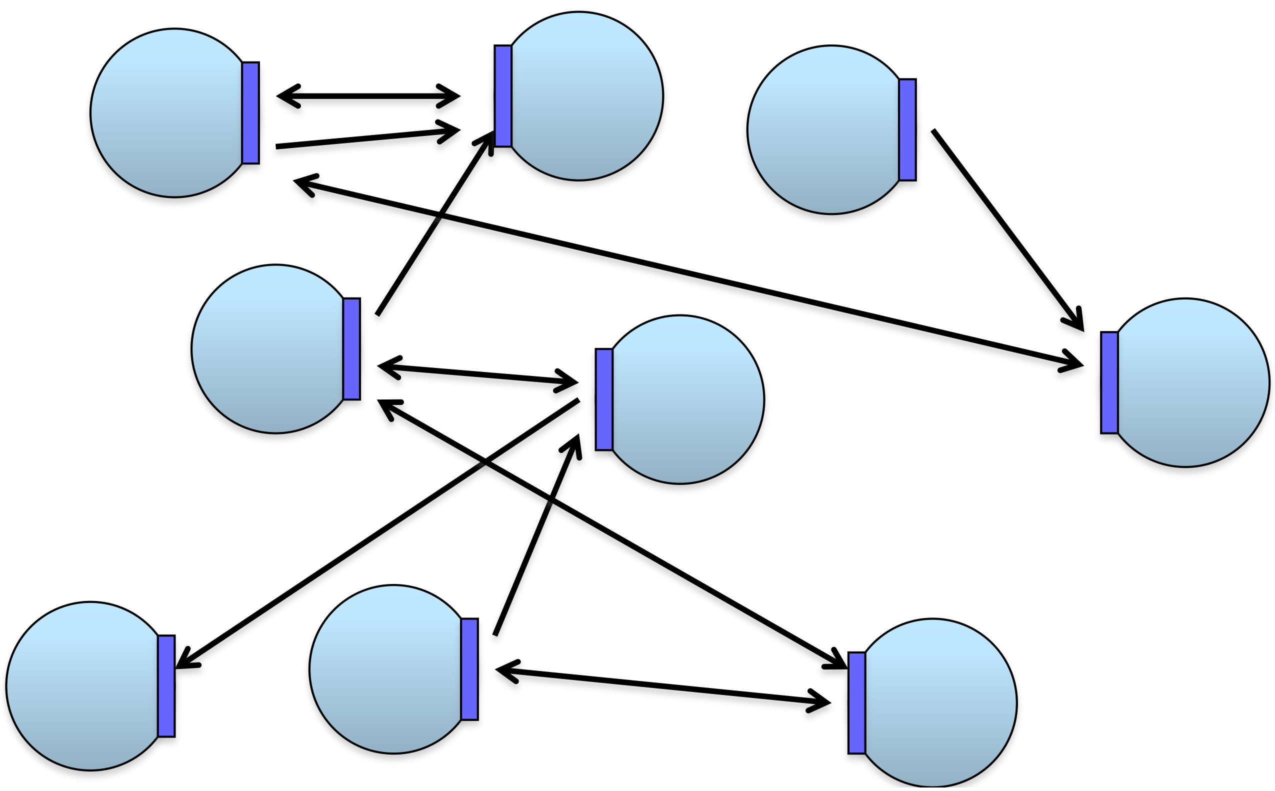
Object-Oriented Design

What does this mean for Java?

Part = Static Class or dynamic Object

Interaction = Method call (or variable reference)

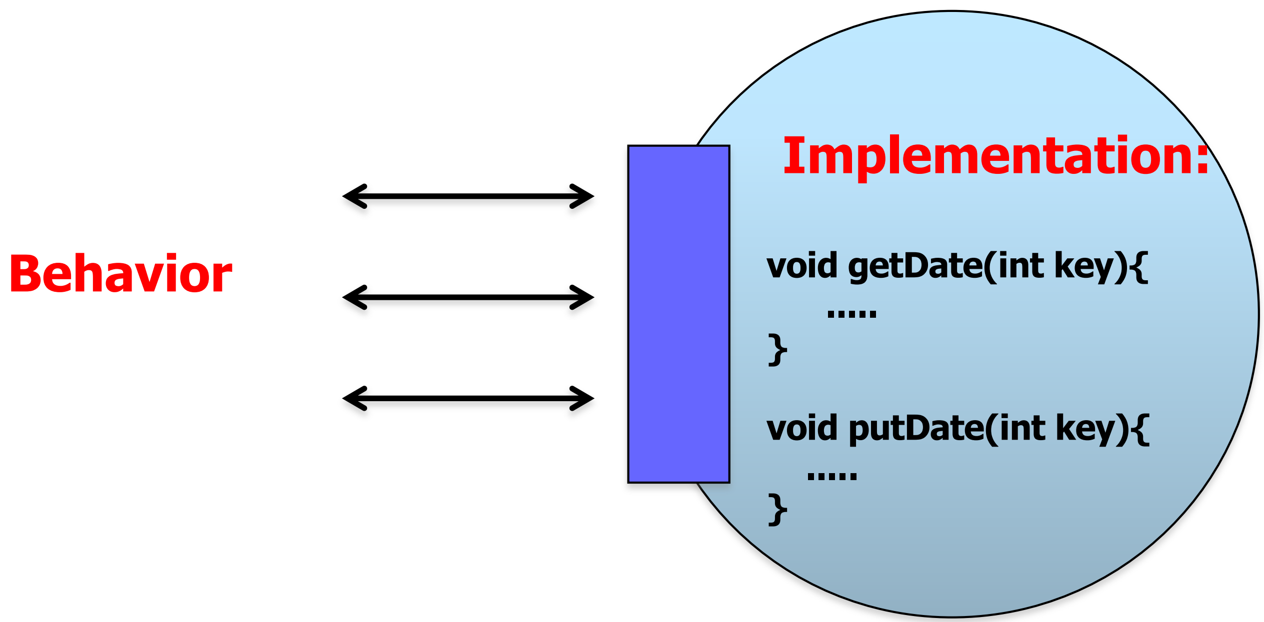
Interface = public members of class



Object-Oriented Design

Two more principles of Software Engineering:

ONE: Separate the **behavior** of a class (defined by its **interface**) from it's **implementation** (the private methods and fields).



Behavior is defined by Interface = collection of public methods and fields of a class

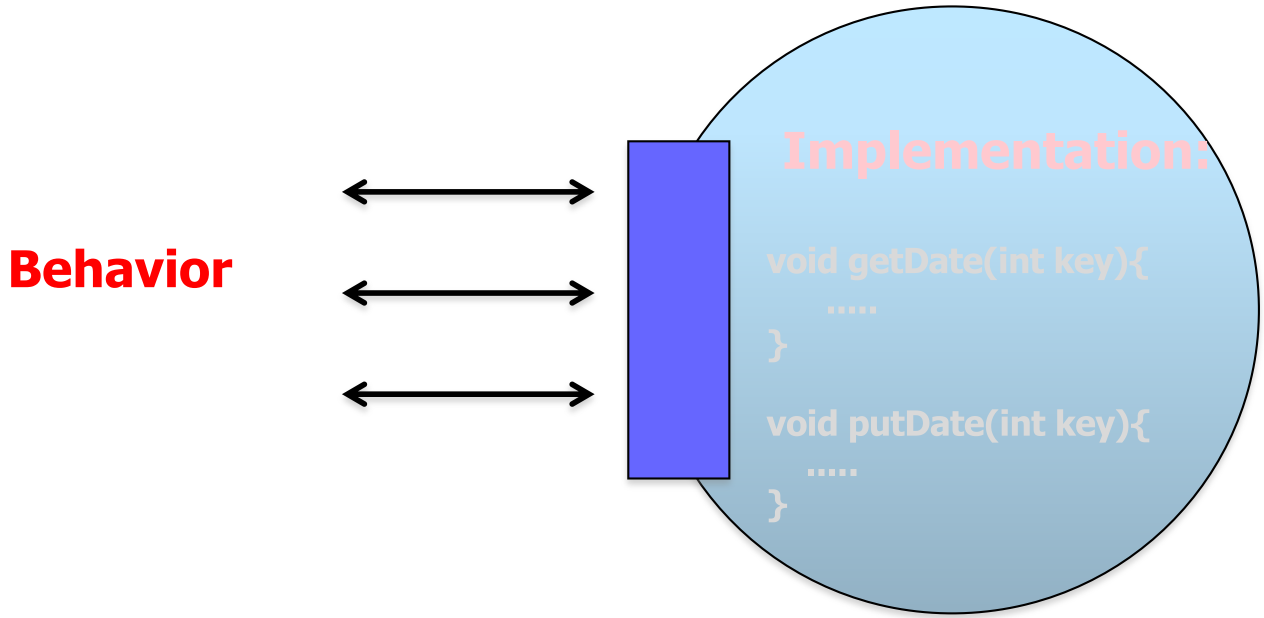
Implementation = collection of private methods and fields of a class.

Object-Oriented Design

Computer Science

Two more principles of Software Engineering:

ONE: Separate the **behavior** of a class (defined by its interface) from it's **implementation** (the private methods and fields).



Interface = collection of public methods and fields of a class

TWO: Protect your implementation by hiding as many details as possible from your (stupid) user! ONLY give them access through the Interface. This is called Information Hiding.

The MOST IMPORTANT thing you can as a Java programmer, therefore, is:

- When you divide, make the interactions as simple and easy to understand as possible;
- Make the interface follow KISS -- **provide as few public methods as possible;**
- Use Information Hiding: If you are not sure whether to make something public or private, **make it private;**

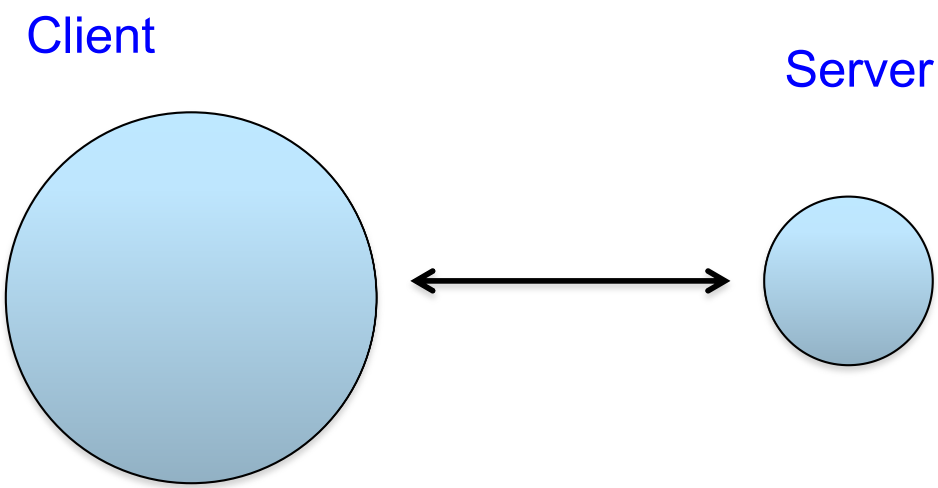
The **advantages of information hiding** are:

- Your code is easier to **understand**, and hence to **use**, and **reuse**;
- **Users can't screw up** your beautifully-crafted KISS code with their "improvements";
- Users can't get used to "back-door" ad-hoc features of your code;
- By separating the (simple) **behavior** of your system from the messy details of its **implementation**, you can **change the actual implementation** any time you want---as long as it behaves the same, this is a huge advantage for **maintenance and reuse.**

Object-Oriented Design: Design Patterns

Over the years, system designers have defined a number of **standard design patterns** for interaction between parts. One of the most useful is the

Client/Server Model:

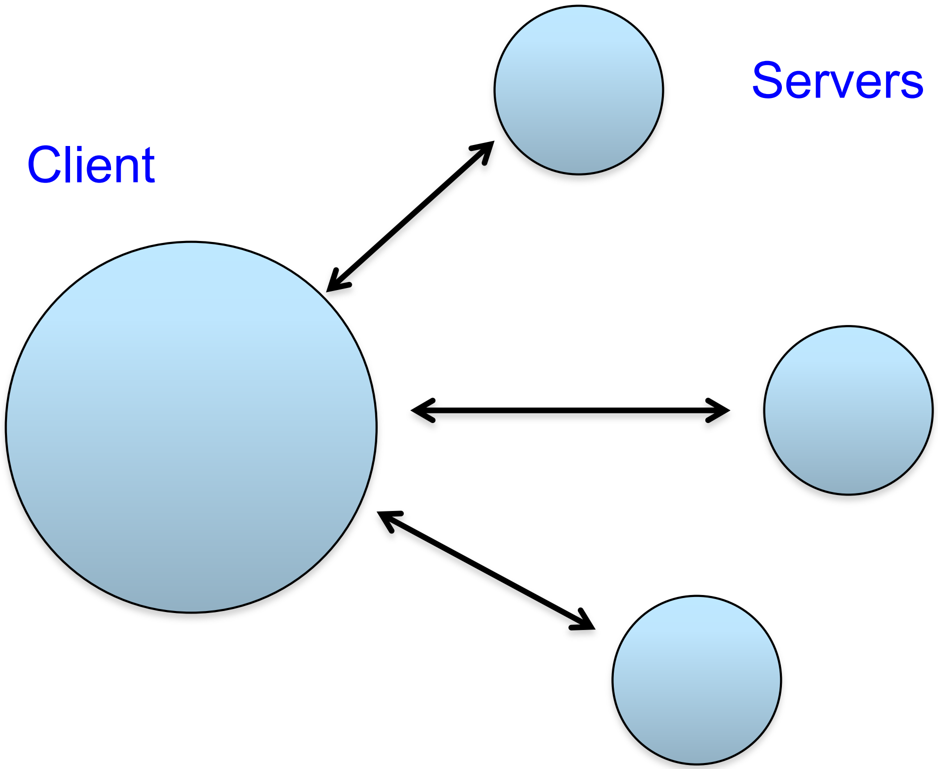


The Client needs services; the Server provides these services.

The Client controls the interaction.

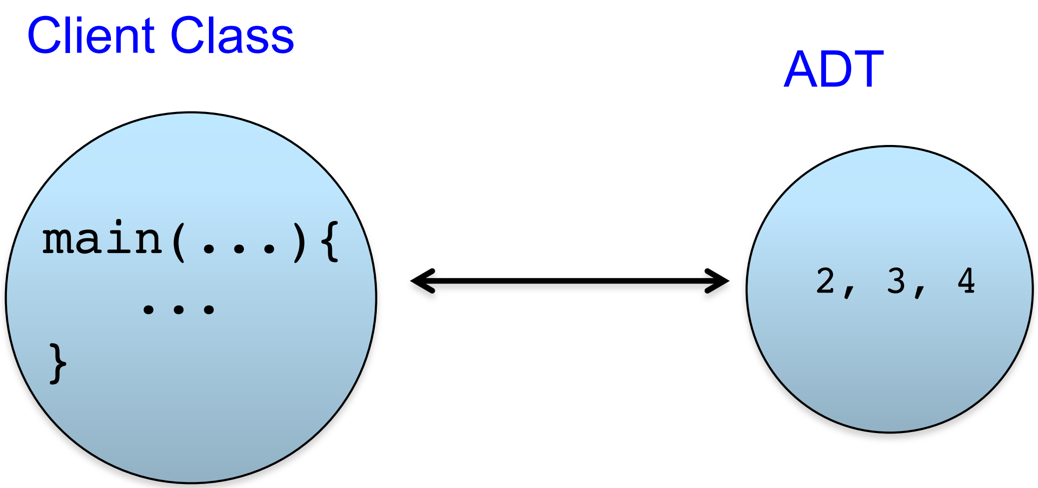
Object-Oriented Design

There may, of course, be many servers:



Object-Oriented Design

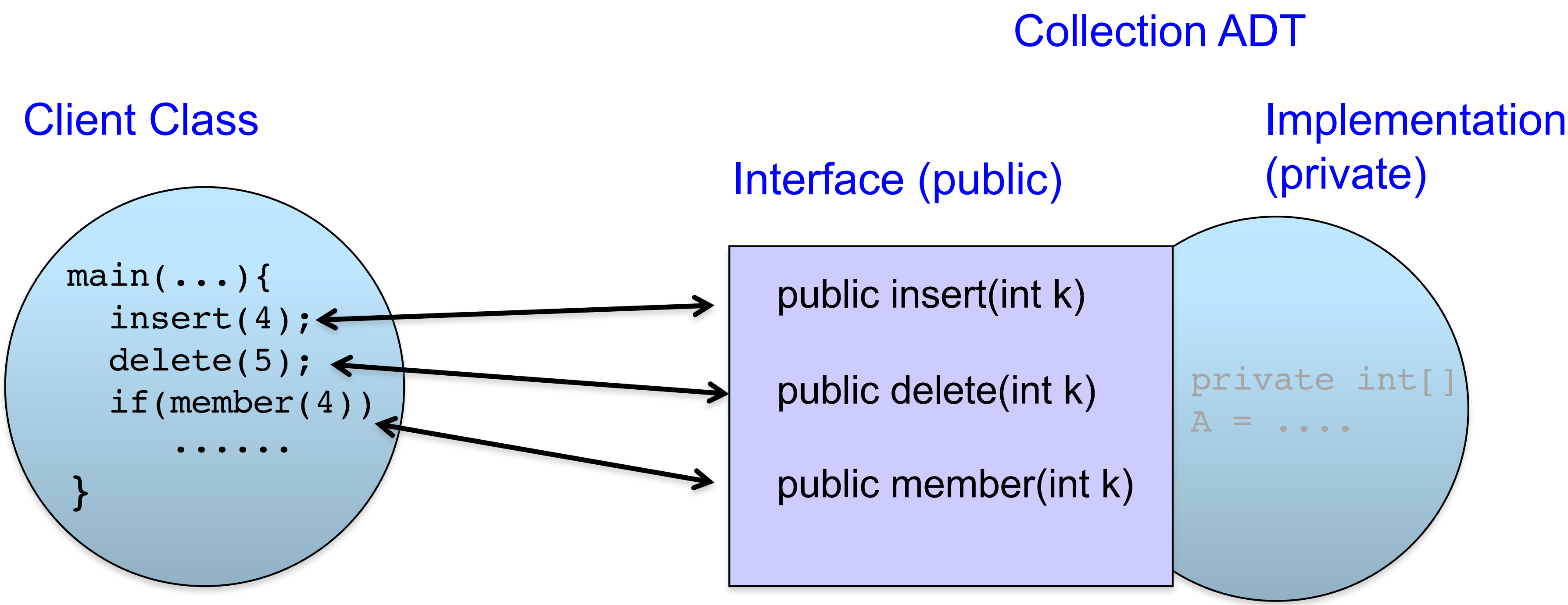
Very commonly, the client is the "main" program, where execution starts and ends, and the servers store data and manipulate this data. The servers are usually called "Data Types" or "Abstract Data Types":



Object-Oriented Design: The Collection ADT

Computer Science

The most basic Abstract Data Type is a Collection, which simply allows you to insert, remove, and check for membership among a collection of integers; the interface of this ADT simply contains public methods for these basic operations



OOD: The Collection ADT

Computer Science

Client.java

```
public class Client {
   public static void main(String [] args) {
      Collection C = new Collection();
      C.insert(2);
      C.insert(3);
      C.delete(2)
      if(C.member(2))
         System.out.println("Oh no....");
   }
}
```

Interface in Red

Implementation in Green

Collection.java

```
public class Collection {
   private int [] A = new int[10];
   private int next = 0;

   public void insert(int k) {
      A[next++] = k
   }

   public void delete(int k) {
        ...   etc.   .....
   }

   public boolean member(int k) {
      ..... etc.    .....
   }
}
```

OOD: The Collection ADT

Computer Science

Client.java

```
public class Client {
   public static void main(String [] args) {
      Collectable C = new Collection();
      C.insert(2);
      C.insert(3);
      C.delete(2)
      if(C.member(2))
          System.out.println(“Oh no....”);
   }
}
```

Collectable.java

```
public interface Collectable {
      public void insert(int k) ;
      public void delete(int k) ;
      public boolean member(int k) ;
}
```

Collection.java

```
public class Collection implements Collectable
{
   private int [] A = new int[10];
   private int next = 0;

   public void insert(int k) {
        A[next++] = k
   }

   public void delete(int k) {
          ...   etc.   .....
   }

   public boolean member(int k) {
        ..... etc.    .....
   }
}
```

OOD: Java Interfaces

Think of an **interface** as a **contract** between the Client and the ADT:

Client: "I need **insert**, **delete**, and **member** methods."

ADT: "No problem."

Client: "Wait, I just met you. How can I **trust** you?"

ADT: "We'll let Java check that the **contract** covers all your needs and that I provide everything in the contract"

Client: "What if I don't need everything in the contract? What if you offer more?"

ADT: "What do you care! As long as you get what you contracted for, you can run!"

Client: "You arrogant tech guys are all alike…. Ok, whatever, where do I sign?"

In mathematical terms, if C is what the client needs, F is what is listed in the interface, and D is what the ADT provides, we have: $C \leq F \leq D$.