Creating and Using Objects

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....

Static World

Dynamic World

Note: no static keyword
Creating and Using Objects

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

Static World

<table>
<thead>
<tr>
<th>MyMath</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
</tr>
<tr>
<td>logOfTwo</td>
</tr>
<tr>
<td>log2</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>

Dynamic World

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 2.3</td>
</tr>
<tr>
<td>y: 4.5</td>
</tr>
</tbody>
</table>

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

name-of-class . name-of-member

for example:

MyMath.logOfTwo

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

variable-name . name-of-member

for example: p.x

Creating and Using Objects

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("(\d+, \d+)");
    }
}
```

> 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:

```
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("(\(x = \) + p.x + ",\(y = \) + p.y + ")");
    }
}
```

> run Geometry

(2.3, 4.5)

> run Geometry

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:

```
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("(\(x = \) + p.x + ",\(y = \) + p.y + ")");
    }
}
```

> run Geometry

(-1.9, 10.3)
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:

```java
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

```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)

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

```java
public class Point {
    double x;
    double y;

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

```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 + ")");
    }
}
```

> run Geometry
(2.3,4.5)
(-1.9,10.3)
Creating and Using Objects

You can even create an array of objects:

```
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 * 2);
        }
        for(int i = 0; i < 5; i++) {
            System.out.println("" + figure[i].x + "," + figure[i].y + ");
        }
    }
}
```

> run Geometry

```
(0.0,0.0)
(1.0,2.0)
(2.0,4.0)
(3.0,6.0)
(4.0,8.0)
```

Creating and Using Objects

```
Static World                                   Dynamic World
```

```
Geometry
```

```
figure
```

```
0:
1:
2:
3:
4:
```
Creating and Using Objects

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.
Creating and Using Objects

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

```java
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 dDist = x - q.x;
        double yDist = y - q.y;
        return Math.sqrt((xDist * xDist) + (yDist * yDist));
    }

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

Dynamic World

```
x: 23, y: 45
```

```
distToOrigin...
distance...
toString...
```

Creating and Using 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 ));
    }
}
```

```
> run Geometry
Distance from p to origin = 3.780216342871597
Distance from p to q = 6.479197481170026
(2.3,4.5)
Distance from (2.3,4.5) to (-1.9,10.3) = 6.479197481170026
```
Creating and Using Objects

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!

```java
public class Point {
    double x;
    double y;

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

    public static void main(String[] args) {
        Point p = new Point(2.3,4.5);
        Point q = new Point(-1.9,18.1);
        System.out.println(p);
        System.out.println("Distance from " + p + " to " + q + " = " + p.distance(q));
    }
}
```

> run Geometry
Distance from p to origin = 3.780211634287597
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:

```java
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

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;

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!
Software Engineering and Object-Oriented Design

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

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

**DIVIDE AND CONQUER**

Special Service Division

INFORMATION FILM
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

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:

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?
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:**

\[
1 \rightarrow 0, \ 2 \rightarrow 1, \ 3 \rightarrow 3, \ 4 \rightarrow 6, \ldots \ N \rightarrow 1+2+\ldots+N-1 = \frac{N(N-1)}{2}
\]

\[
= \sim N^2/2 = \text{Geometric growth!}
\]
Object-Oriented Design

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)

What does this mean for Java?

Part = Class

Interaction/conversation = Method call (or reference to a field)
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
Two more principles of Software Engineering:

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

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.
Object-Oriented Design

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:

- Client
- Servers

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":

- Client Class
- ADT

```java
main(...){
...
}
```

```
2, 3, 4
```
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.

Client Class

```
main(...){
    insert(4);
    delete(5);
    if(member(4))
        .......
}
```

Interface (public)

```
public insert(int k)
public delete(int k)
public member(int k)
```

Implementation (private)

```
private int[] A = ....
```

### OOD: The Collection ADT

**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....");
    }
}
```

**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. ..... 
    }
}
```

Interface in Red

Implementation in Green