Methods, classes, and objects

One way to view Object Orient Programming (OOP) is to recognize that the Orientation is different.

In traditional programming, programmers focus on functions and how the functions relate to one another.

In OOP, the focus is on data, and how data "elements" relate to one another:

In short, OOP is (tends to be) data centric as opposed to function centric

So, in the text's BankAccount example, we would focus on attributes of a bank account, e.g.

Balance (Could also include "accountNumber" and "customerName" fields)

With methods

```
deposit()
withdrawl()
getBalance()
```

Instance methods vs. class methods

An instance method is a method that is applied to each individual object.
A class method is applied to an attribute that is common to all objects of a given class.

A Simplified class example, consider a simple geometric object, a point

The point has two data attributes: x and y

There may also be a few operations, e.g.

```
Point          // a constructor to initialize the point object
getX()         // returns the x location of the point
getY()         // returns the y location of the point
move()         // moves the point by the specified amount
```

This point class would be defined as follows for programmers who wish to use objects of class point

```
Point()
Point(int x, int y)
public int getX()
public int getY()
public void move(int deltaX, int deltaY);
```

A simple example using the Point class:

```
public class usePoint
{
        public static void main(String[] args)
        {
        Point p = new Point(5,5);
        Point p1 = new Point(4,4);
        Point p2 = new Point();   // invokes default constructor

        System.out.println("Point p: " + "(" + p.getX() + "," + p.getY() + ")");
        System.out.println("Point p1: " + "(" + p1.getX() + "," + p1.getY() + ")");
        System.out.println("Point p2: " + "(" + p2.getX() + "," + p2.getY() + ")");

        p1.move(2,2);
        System.out.println("Point p1: " + "(" + p1.getX() + "," + p1.getY() + ")");
        }
}
```

Sample run:

```
$ java usePoint
Point p: (5,5)
Point p1: (4,4)
Point p2: (0,0)
Point p1: (6,6)
```

The implementation of class Point:

```
public class Point
{
        Point() // default constructor; invoked if parameters are not specified during object construction (using keyword "new")
        {
        x = 0;
        y = 0;
        }

        Point(int a, int b)
        {
        x = a;
        y = b;
        }

        public int getX()    // getX() and getY() are accessor methods
        {
        return x;
        }

        public int getY()
        {
        return y;
        }

        public void move(int deltaX, int deltaY)    // move() is a mutator method
        {
        x+= deltaX;
        y+= deltaY;
        }

private int x;
private int y;
}
```

Notes:

There are two constructors:

Point()

And

Point(int a, int b)

Both are legitimate constructors, just note that they differ when objects are created, e.g.:

Point p = new Point()

And

Point p1 = new Point(5,5);

Also note which methods are mutators, and which are accessors (accessors shown for informational purposes).

Important Note: The use of accessors is, in general, discouraged as it violates the principle of ***Information Hiding***

While somewhat involved, the point is that accessors should be AVOIDED whenever possible!

Typically there are better ways to provide information, without revealing how the information is represented — a concept that we will cover throughout the course

Finally,

There are not any class methods, all of the methods are instance methods,

In other words all of the defined methods apply to objects of Class Point — more later in the course

The Point class as defined is not bad, but it lacks some key capabilities, e.g., determining the distance between two points

We could modify the original Point class, but that might impact existing software that uses class point…

- There would be two versions of class Point floating about, this is NOT good software engineering

Difficulty in maintaining differences in releases of software; CM/support issues

- A change to the original Point class might introduce bugs, hence maturity may be compromised

A better way is to create a derived class using the OOP concept of inheritance

Existing code base and capabilities are retained AND new capabilities are introduced or added

```
class Pt extends Point
{
       Pt()
       {
       super(0,0);
       }

       Pt(int a, int b)
       {
       super(a,b);
       }

       private double square(int k)
       {
       return k*k;
       }

       public double distance(Point p)
       {
       double dist = Math.sqrt( square(getX()-p.getX()) + square(getY()-p.getY()) );
       return dist;
       }
}
```

Notes:

The new class, Pt, inherits all of the existing methods and internal data structures from class Point; Pt is "derived" from Point
The use of "super()" simply invokes the Point constructor such that the internal data for Pt is set

To use this extended class:

```
public class usePoint1
{
       public static void main(String[] args)
       {
       Point p  = new Point(5,5);
       Pt    p1 = new Pt(4,4);
       Pt    p2 = new Pt(9,9);

       System.out.println("Point p: " + "(" + p.getX() + "," + p.getY() + ")");
       System.out.println("Point p1: " + "(" + p1.getX() + "," + p1.getY() + ")");

       p1.move(3,1);
       System.out.println("Point p1, after the move: " + "(" + p1.getX() + "," + p1.getY() + ")");

       double theDistance = p1.distance(p2);
       System.out.println("The distance between p1 and p2 is: " + theDistance);
       }
}
```

Sample run:

```
% java usePoint1
Point p: (5,5)
Point p1: (4,4)
Point p1, after the move: (7,5)
The distance between p1 and p2 is: 4.47213595499958
```

However, the point class itself has some room for improvement that should have been there in the first place.

For example, what if I need to:

Compare two points
Make a copy of two points
Add a print capability to the Point class

Example use of the derived class Pt and the class Point with the ability to clone (make copies), do comparison, and provide a print() method

```
public class usePoint2
{
	public static void main(String[] args)
	{
	Point p = new Point(5,5);
	Point q = new Point(4,4);
	Pt p1 = new Pt(10,11);
	Pt p2 = new Pt(5,5);

	p2.move(2,2);   // The move method is inherited by objects of the derived class Pt

	System.out.print("Point p: ");
	System.out.println(p);                    // demo toString() method
	System.out.println("Point q: "  +  q);
	System.out.println("Point p1: " + p1);
	System.out.println("Point p2: " + p2);

	double distance = p1.distance(p2);
	System.out.println("The distance between p1 and p2 is: " + distance);

	Point p3 = (Point)q.clone();
	System.out.println("Cloned Point p3: " + p3);

	if (q.equals(p3))
		System.out.println("Points q and p3 are equal.");
	else
		System.out.println("Points q and p3 are not equal.");
	}
}
```

Notes:

- The toString() method is called System.out.println()
- Use of the equals(), toString and clone() methods
- Implementing clone and equals methods can be complex (as shown on the next page)

While it is necesary to understand the concept of cloning and comparing objects, you only need to be able to write simple comparison methods, I will provide (for this course) cloning methods as needed.

% java usePoint2

Point p: (5,5)
Point q: (4,4)
Point p1: (10,11)
Point p2: (7,7)
The distance between p1 and p2 is: 5.0
Cloned Point p3: (4,4)
Points q and p3 are equal.

The enhanced Point class:

```
class Point implements Cloneable
{
        Point()
        {
        x = 0;
        y = 0;
        }

        Point(int a, int b)
        {
        x = a;
        y = b;
        }

        public int getX()
        {
        return x;
        }

        public int getY()
        {
        return y;
        }

        public void move(int deltaX, int deltaY)
        {
        x+= deltaX;
        y+= deltaY;
        }

        // For many classes and programs it may be necessary to print object contents,
        // compare two objects of the same class, and copy objects.

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

        public boolean equals(Object obj)
        {
        if (!getClass().equals(obj.getClass()))
                return false;

        Point b = (Point)obj;

        return (x == b.getX() && y == b.getY());
        }

        // **** NOTE ****  You are NOT required to understand cloning for any test or quiz
        //

        public Object clone()
        {
        try
            {
            return super.clone();   // Extra cloning is required for classes defined within classes
            }
        catch(CloneNotSupportedException e)
            {
            return null;
            }
        }

private int x;
private int y;
}
```

Note the implementations of toString(), equals(), and clone()

Also note that these functions are applicable to derived classes, with limitations

Polymorphism and Dynamic Binding

Provides “ease of understanding” and reuse that is not available in non-OOP languages

Consider the class Point from before with a new function called print()

```
public void print()
{
System.out.println("Point: (" + x + "," + x + ")");
}
```

Consider the Point class from before and a derived class called EvenPt

The EvenPt class requires that all objects of type EvenPt can only be on even numbered vertices.

```
class EvenPt extends Point
{
        EvenPt()
        {
        super(0,0);
        }

        private int evenChk(int k)
        {
        if ( (k/2)*2 == k)
                return 0;
        else
                return 1;
        }

        EvenPt(int a, int b)
        {
        super(a,b);

        super.move(evenChk(a), evenChk(b));  // move point by 1, V/H to ensure on even Point
        }

        private double square(int k)
        {
        return k*k;
        }

        public double distance(Point p)
        {
        double dist = Math.sqrt( square(getX()-p.getX()) + square(getY()-p.getY()) );
        return dist;
        }

        public void move(int a, int b)
        {
        if ( (a/2)*2 == a && (b/2)*2 == b)  // Can ONLY move even amounts
                super.move(a,b);
        else
                System.err.println("ERROR: EvenPt objects must ALWAYS by on even points.");
        }

        public void print()
        {
        System.out.println("Even Point: <" + getX() + "," + getY() + ">");
        }
}
```

Notes:

- Super.move
- EvenPt move() method
- Constructors in derived classes MUST call their base class constructors before doing anything else!
- Use of System.err.println()

Code for useEvenPt:

```
public class useEvenPt
{
        public static void main(String[] args)
        {
        Point p = new Point(5,5);
        EvenPt p1 = new EvenPt(10,12);

        p1.move(2,2);   // The move method is inherited by objects of the derived class Point

        p.print();
        p1.print();

        // A simple example of polymorphism and dynamic binding

        Point[] pts = new Point[3];

        pts[0] = p;
        pts[1] = p1;

        pts[2] = new EvenPt(4,8);
        pts[2].move(1,3);

        for (int i = 0; i < pts.length; i++)
                pts[i].print();
        }
}
```

Notes:

The correct print() method is called for each object:

On Point p, the "(" and ")" are printed
On Point p1, "<" and ">" are printed

This is referred to as polymorphism, i.e., each object responds as appropriate to the method print(), e.g.:

```
p.print();
p1.print();
```

The corresponding print() method is invoked based on the class definition in each element of the array.

```
for (int i = 0; i < pts.length; i++)
        pts[i].print();
```

The for loop demonstrates *dynamic binding* and *polymorphism*…

Methods associated with each type of point are called as necessary AND newly derived types can be added without changing any part of the for-loop!

No need for a case statement/nested ifs – or changes to the existing for loop code!!

Sample run:

```
% java useEvenPt
Point: (5,5)
Even Point: <12,14>
ERROR: EvenPt objects must ALWAYS by on even points.
Point: (5,5)
Even Point: <12,14>
Even Point: <4,8>
```

A common error:

```
% java useevenpt
Can't find class useevenpt
```

Note that on Windows … the file name UseEvenPt.class is stored in mixed case, so it is necessary to specify: java useEvenPt