Recursion

- A recursive method is a method that calls itself (possibly many times,)
- Recursion can be indirect:
  - p calls q which calls p
  - p calls q which calls r which calls p

```
public class E {
    static int p(int n)
    {
        if (n >= 0) return q(n-1);
        return 2;
    }
    static int q(int n)
    {
        if (n == 0) return 3;
        return 2 * p(n-2);
    }
}
```

Recursion (contd.)

- Indirect recursion might happen in different classes:

```
public class E {
    static int p(int n)
    {
        if (n >= 0) return F.q(n-1);
        return 2;
    }
}
public class F {
    static int q(int n)
    {
        if (n == 0) return 3;
        return 2 * E.p(n-2);
    }
}
```

Recursion (contd.)

- Non-static methods can also be recursive:

```
public class G {
    int n, r;
    G(int n)
    {
        this.n = n;
        r = 1;
    }
    void f()
    {
        if (n > 0) {
            r = r * n;
            n = n - 1;
            f();
        }
    }
}
```

Recursion and termination

- Recursion might not terminate: divergence
- Example 1: (diverges)

```
static void forever()
{
    System.out.println("1");
    forever();
}
```

- Example 2: (diverges)

```
static int g(int n)
{
    if (n >= 0) return g(n) * n;
    return 1;
}
```

Recursion and termination (contd.)

- Divergence in iterative programs: If the body of the loop doesn't change the variables of the condition, then the condition never changes, so if it was initially true, the loop never stops.

```
static int g(int n)
{
    r = 1;
    while (n >= 0) {
        r = r * n;
    }
}
```

- For recursion to terminate, the recursive call must "advance" towards the base case.

Recursion and termination (contd.)

- "Advance", not just change:
- Example 3: (diverges)

```
static int g(int n)
{
    if (n >= 0) return g(n + 2) * n;
     return n;
}
```

Divide and Conquer

- Given a problem

  1. Divide it into smaller subproblems
  2. Solve each subproblem
  3. Combine the solutions to form the solution of the original problem

Defining characteristics of OOP

- Qualities of well-developed software:
  - Modularity
  - Reusability
  - Integration
  - Abstraction
- A programming language is object-oriented if it supports:
  - Class definitions and class instantiation
  - Message-passing
  - Encapsulation
  - Polymorphism
  - Inheritance

Encapsulation and visibility

- Abstraction and visibility
- Hiding the state of an object
- Hiding (part of) the structure of an object (attributes and/or methods.)
- Hiding from clients
- Security: maintaining the integrity of data. Enforcing limited visibility so that clients cannot "corrupt" the state of an object, so that only the class of the object can change the object's state.
- Visibility modifiers (for attributes and methods): `public`, `private` and `protected`.
- Visibility modifiers are orthogonal (independent) of whether the attribute or method is static or not. So they can be combined in any way.

Visibility modifiers for attributes

- A normal attribute has the syntax:

  type variable ;

- With a modifier:

  modifier type variable ;

- So there are three forms:

  public *type variable* ;
  private *type variable* ;
  protected *type variable* ;

- The default is protected.

- In general:

  [*modifier*] [static] *type variable* [= *expression*];

Visibility modifiers for methods

- A normal method has the syntax:

```
type method(type1 param1, type2 param2,
            ..., typen paramn)
{
    statements;
}
```

- In general:

```
[modifier] [static] type method(type1 param1,
                                type2 param2,
                                ...,
                                typen paramn)
{
    statements;
}
```

Example

```
public class P
{
    private String s;
    public String t;
    private void m()
    {
        System.out.println(t + ";" +s);
    }
    public void k()
    {
        s = "first and "+t;
        m();
    }
}
```

Example (contd.)

```
public class Q
{
    public void m()
    {
        P p1 = new P();
        p1.t = “second”;     // OK, t is public
        p1.s = “third”;      // Error, s is private
        p1.k();                 // OK
        p1.m();                 // Error
    }
}
```

Encapsulation and modularity

```
public class Point
{
    private double x, y;
    public void set_xy(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double get_x()
    {
        return x;
    }
    public double get_y()
    {
        return y;
    }
}
```

Example (contd.)

```
public class Picture
{
    Point top_left, bottom_right;
    public Picture()
    {
        top_left = new Point();
        bottom_right = new Point();
    }
    void do_something()
    {
        top_left.set_xy(2.0,-5.0);  // OK
        top_left.x = -6.0;          // Error
        bottom_right.x = top_left.x-3.0; // Error
        // ...
    }
}
```

Hiding implementation

- Points have alternative characterizations:
  - Rectangular coordinates
  - Polar coordinates
- If a point is represented using rectangular coordinates, you would like to be able to know what are its polar coordinates and viceversa.
- We should implement a class in a way which makes its clients independent of the internals of the class. This is, we can implement a class in any way as long as its clients do not need to be changed. (Abstraction implies Modularity and Integration.)

Underlying rectangular representation

```
public class Point
{
    private double x, y;
    // ... same as before, plus the following:
    public double get_angle()
    {
        return Math.atan2(y, x);
    }
    public double get_magnitude()
    {
        return Math.sqrt(x*x + y*y);
    }
    public void set_angle_and_magnitude(double a,
                                        double r)
    {
        x = r * Math.cos(a);
        y = r * Math.sin(a);
    }
}
```

Underlying polar representation

```
public class Point
{
    private double angle, magnitude;
    public double get_angle()
    {
        return angle;
    }
    public double get_magnitude()
    {
        return magnitude;
    }
    public void set_angle_and_magnitude(double a,
                                        double r)
    {
        angle = a;
        magnitude = r;
    }

    // Continues below ...
```

```
    public void set_xy(double x, double y)
    {
        magnitude = Math.sqrt(x*x + y*y);
        angle = Math.atan2(y, x);
    }
    public double get_x()
    {
        return magnitude * Math.cos(angle);
    }
    public double get_y()
    {
        return magnitude * Math.sin(angle);
    }
}
```

Privacy is relative

- Private members (attributes and methods) can be accessed within the same class.

- An object in a class can access the private members of other objects in the same class.

```
public class A
{
    private int x;
    public void set_x(int some_x)
    {
        x = some_x;
    }
    public void m(A u)
    {
        System.out.println(x +" "+ u.x);  // OK
    }
}
```

Privacy is relative (contd.)

```
public class B
{
    void m()
    {
        A n = new A(), k = new A();
        // n.x = 8;   // Error
        n.set_x(17);
        k.set_x(29);
        n.m(k);
    }
}
```