Principles of well-designed software

- Qualities of well-developed software:
  - Modularity: Components are independent modules.
  - Reusability: Components are reusable.
  - Integration: Components can be easily integrated with one another.
  - Abstraction: Irrelevant details, and details about how a component works, are hidden from the clients of the component.

Defining characteristics of OOP

- A programming language is object-oriented if it supports:
  - Class definitions and class instantiation (creating objects)
  - Message-passing (communication between objects)
  - Aggregation (object structure)
  - Encapsulation (visibility and abstraction)
  - Polymorphism (abstraction and reactivity)
  - Inheritance (abstraction and organisation)

Method overloading

- A simple form of polymorphism (*ad-hoc polymorphism*)
- There can be several (static or not) methods with the same name...
- ...but the type or number of parameters must be different

Example

```
public class CheckingAccount {
  double balance;
  CheckingAccount() { balance = 0.0; }
  void deposit(double amount)
  {
    balance = balance + amount;
  }
}
```

Example

```
public class MagicAccount {
  double balance;
  MagicAccount() { balance = 0.0; }
  void deposit(double amount)
  {
    balance = balance + amount + 100.0;
  }
}
```

Example

```
public class BankingApplication {
  public static void main(String[] args)
  {
    CheckingAccount a = new CheckingAccount();
    MagicAccount b = new MagicAccount();
    a.deposit(500.0);
    b.deposit(300.0);
  }
}
```

Example

```
public class CanadianDollars {
  double amount, rate;

  CanadianDollars(double a)
  {
    amount = a;
    rate = 0.75;
  }

  double USvalue()
  {
    return amount * rate;
  }
}
```

Example

```
public class Euros {
  double amount, rate;

  Euros(double a)
  {
    amount = a;
    rate = 1.24;
  }

  double USvalue()
  {
    return amount * rate;
  }
}
```

Example

```
public class MagicAccount {
  double balance;

  MagicAccount() { balance = 0.0; }

  void deposit(double amount)
  {
    balance = balance + amount + 100.0;
  }
}
```

Example

```
public class MagicAccount {
  double balance;

  MagicAccount() { balance = 0.0; }

  void deposit(CanadianDollars amount)
  {
    balance = balance
            + amount.USvalue() + 200.0;
  }
  void deposit(Euros amount)
  {
    balance = balance
            + amount.USvalue() + 100.0;
  }
}
```

Example

```
public class BankingApplication {
  public static void main(String[] args)
  {
    CheckingAccount a = new CheckingAccount();
    MagicAccount b = new MagicAccount();
    a.deposit(500.0);

    CanadianDollars dollars;
    Euros eus;
    dollars = new CanadianDollars(300.0);
    eus = new Euros(100.0);
    b.deposit(dollars);
    b.deposit(eus);
  }
}
```

Example

```
public class A {
  void f(int x)
  {
    System.out.println(“one: ”+x)
  }
  void f(boolean x)
  {
    System.out.println(“two: ”+x)
  }
}
public class B {
  void g()
  {
    A u = new A();
    u.f(5);
    u.f(false);
  }
}
```

Same for static methods

```
public class A {
  static void f(int x)
  {
    System.out.println("one: "+x)
  }
  static void f(boolean x)
  {
    System.out.println("two: "+x)
  }
}
public class B {
  void g()
  {
    A.f(5);
    A.f(false);
  }
}
```

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
        // ...
    }
}
```

Encapsulation

- A `public` variable or method can be accessed by any method in any class anywhere at any time
- A `private` variable or method can be accessed only by objects of the same class, this is, only by methods of the same class.
- ...but a `private` variable or method can be accessed by all objects of the same class
- A `protected` variable or method can be accessed by any method in any class *in the same package only, or by subclasses*

Encapsulation and modularity

```
public class BankAccount
{
  private double balance;
  public BankAccount() { balance = 0.0; }
  public void deposit(double amount)
  {
    balance = balance + amount;
  }
  public void widthdraw(double amount)
  {
    if (amount <= balance)
      balance = balance - amount;
  }
  public void transfer(BankAccount other)
  {
    this.balance = this.balance + other.balance;
    other.balance = 0.0;
  }
}
```

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);
    }
}
```

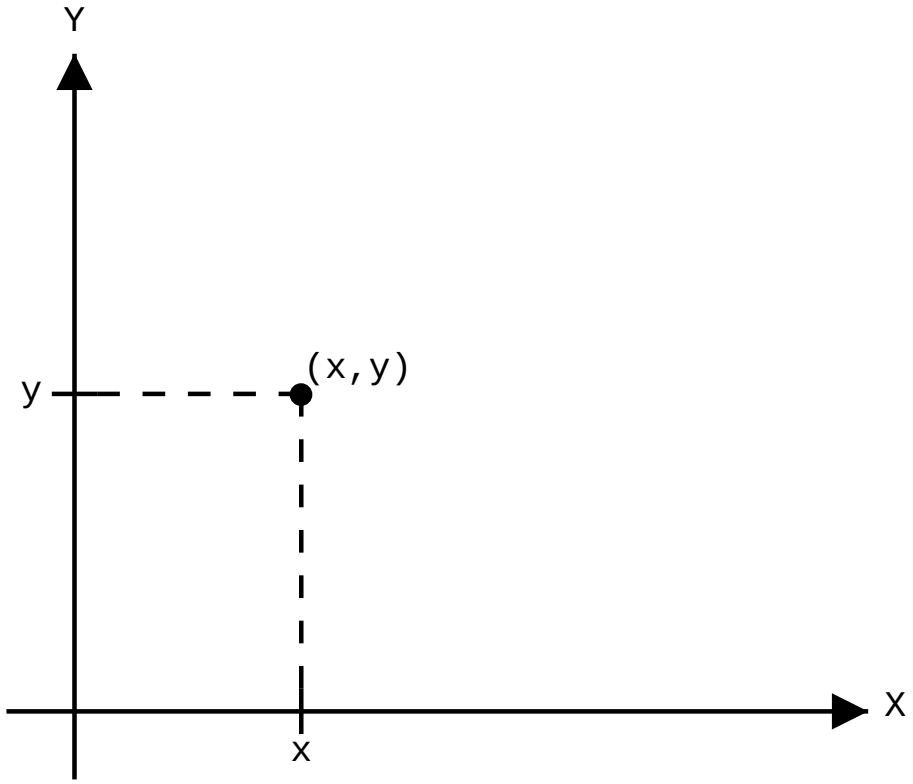
Encapsulation

- The purpose of encapsulation is abstraction: hiding details
- Which details do we want to hide?
  - Internal structure
  - Method implementations
- Security: keeping information consistent; keeping the state of an object consistent.
- Good design:
  - Declare attributes as private, and
  - Declare public accessor methods to only those attributes which you want the client to be able to observe.

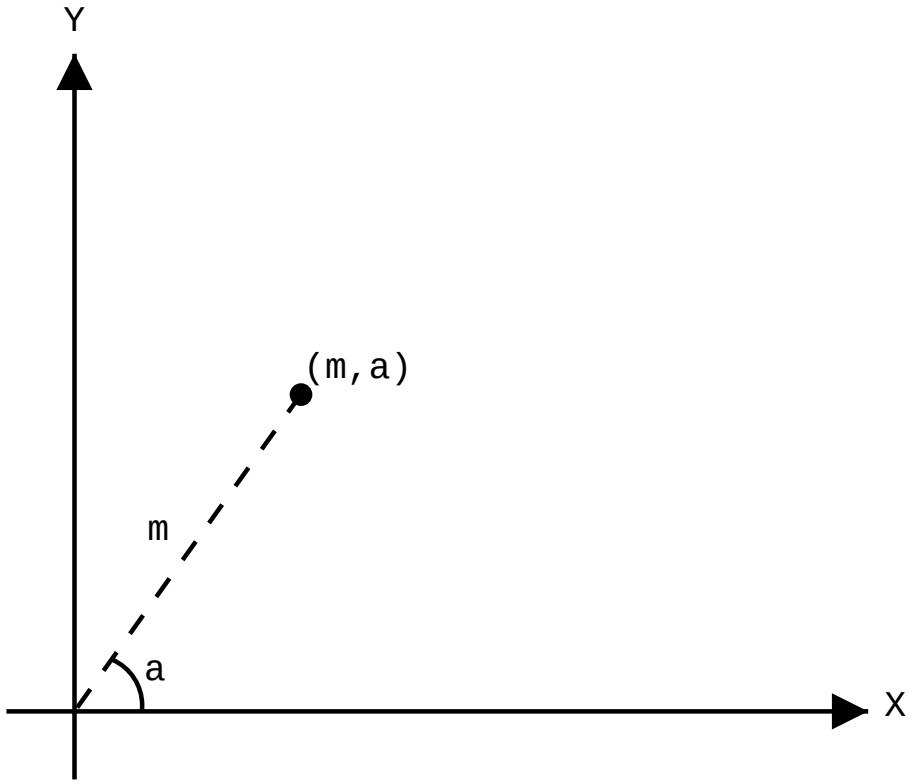
Hiding implementation

- 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.)

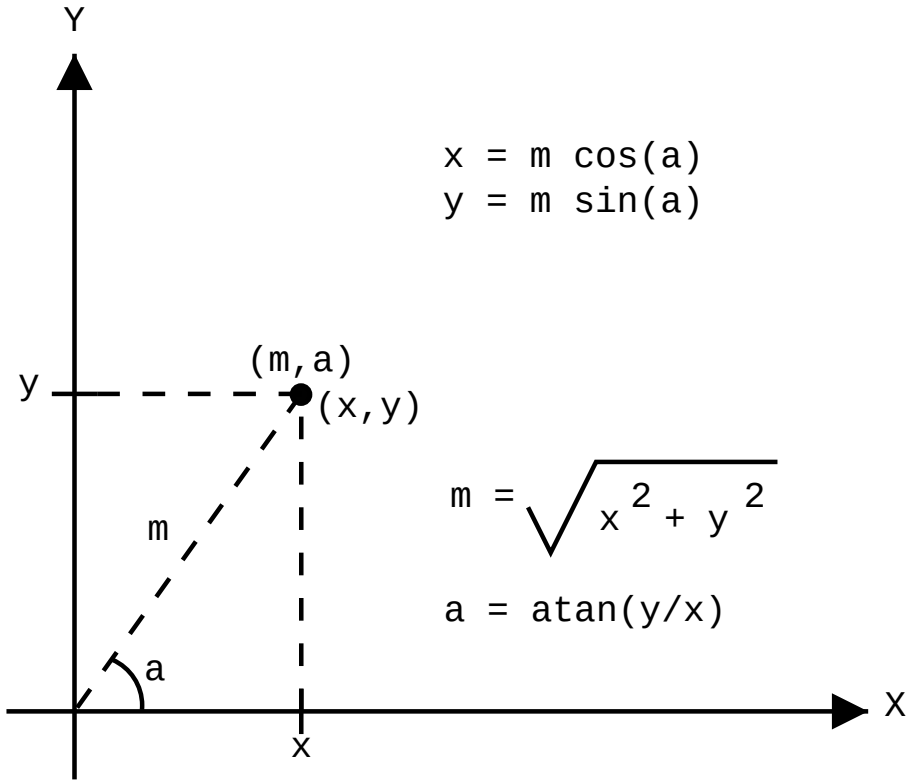
Point representation



Point representation



Point representation



Point operations

- Accessor methods:
  - get_x
  - get_y
  - get_angle
  - get_magnitude
- Mutator methods
  - set_xy
  - set_angle_and_magnitude

Underlying rectangular representation

```
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;
  }

  // Continues below ...
```

```
  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);
    }
}
```

Aggregation

- Analisys:
  - Identify relevant information: objects and types of objects (classe)
  - Identify relationships between objects
- Different kinds of relationships depending on the type of the objects involved
- For example:
  - Numeric relationships:
    * account balance > 0
    * car fuel > 10
    * hitPoints <= maxHitPoints
    * number of heads <= 2
    * number of fingers > 1
    * tax_payable = base + (income - cutoff)*rate
  - Structural relationships:
    * A bank account *has a* balance and an owner
    * A car *has an* engine
    * A person *has a* name and a head

Objects and Aggregation

```
public class Engine {
    // ...
}
public class Car {
    Engine engine;
    // ...
}
public class StreetSimulation {
    void p()
    {
        Car mycar = new Car();
        mycar.engine = new Engine();
    }
}
```

Objects and Aggregation

```
public class Engine {
    // ...
}
public class Car {
    Engine engine;
    // ...
}
public class StreetSimulation {
    void p()
    {
        Car mycar = new Car();
        mycar.engine = new Engine();
    }
}
```

Objects as first class values

- Objects can be attibutes of other objects

```
public class Rabbit {
  void jump() { ... }
}
public class Cage {
  Rabbit my_rabbit;
  void put(Rabbit a)
  {
    my_rabbit = a;
  }
  Rabbit get()
  {
    return my_rabbit;
  }
}
```

...elsewhere...

```
Rabbit bugs = new Rabbit();
Cage c = new Cage();
c.put(bugs);
Rabbit wester = c.get();
```

The end