香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2026

Problem Solving Using Object Oriented Programming

OOP Part 1

Overview

- What is Structured Programming?
- What is Object Oriented Programming?
- Objects and Classes
- Basic Structure of a Class
- Constructor, Instance Variables & Methods
- `new`, `.`, `toString`, `this`, `pubic`, `private`, `final`,...
- Creating objects of a class from another class

Structured Programming

- So far, we have been talking about fundamental programming concepts, mostly structured programming
- Structured Programming:
  - Selection (if-then-else, switch)
  - Repetition (for-loop, while-loop, do-while-loop)
  - Sequence (compound statement/block, methods/functions)

Ref: http://prezi.com/mfeeo_axdheq/structured-programming/

Object-Oriented Programming

- Structured Programming is fundamental to Object-Oriented Programming
- Object-Oriented Programming:
  - Models everything as **objects**;
  - Considers how objects *interact* with each other; and
  - Models how objects *change* through interactions

What is an Object?

- Objects – an encapsulation of (1) **fields** and (2) **methods**
- Fields – also known as **instance variables** / **data attributes**
  - the data;
  - any meaningful, useful, relevant information about the object
- Methods – known as **member functions** in other languages
  - actions that to retrieve the data or to modify the data.
  - interact with other objects and to achieve certain goal

Example of Objects

- Context – a simple drawing application
- Objects – circles, squares, rectangles, ellipses…
- Fields – size, location, color, …
- Methods –
  - actions that to retrieve the color/sharp or to change its color, fill…
  - interact with other objects like collision, bouncing

What is a Class?

- A class is a category of objects
- A class is a blueprint for creating objects
- A class is reusable



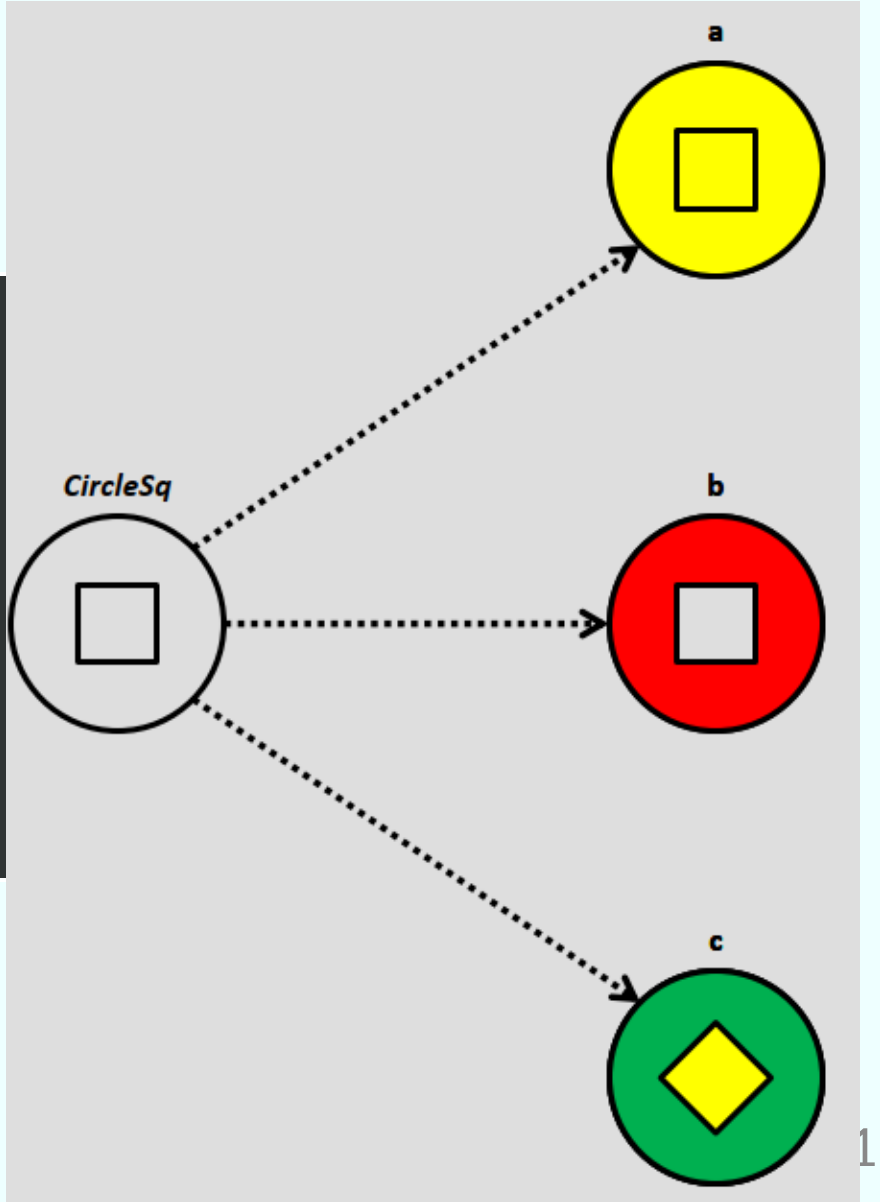
Shall see from some examples

What is a Class?

- A class is a category of objects
- A class is a blueprint for creating objects
- A class is reusable

```
a = new CircleSq();
b = new CircleSq();
c = new CircleSq();

a.setColor(yellow);
b.setCircleColor(red);
c.rotateSq(45);
c.setCircleColor(green);
c.setSqColor(yellow);
...
```



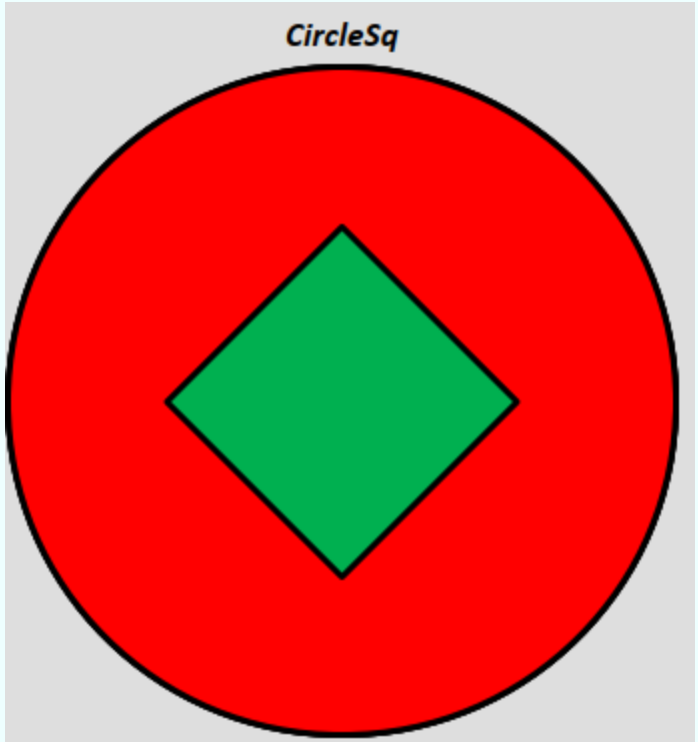
What is a Class?

For each `CircleSq` **object**, we have...

1. A circle with a square inside
2. Need to remember the color of the circle
3. Need to remember the color of the square
4. Need to remember the rotation angle of the square
5. . . .

So, for each `CircleSq`, we have a few properties:

- `circleColor`
- `sqColor`
- `sqAngle`

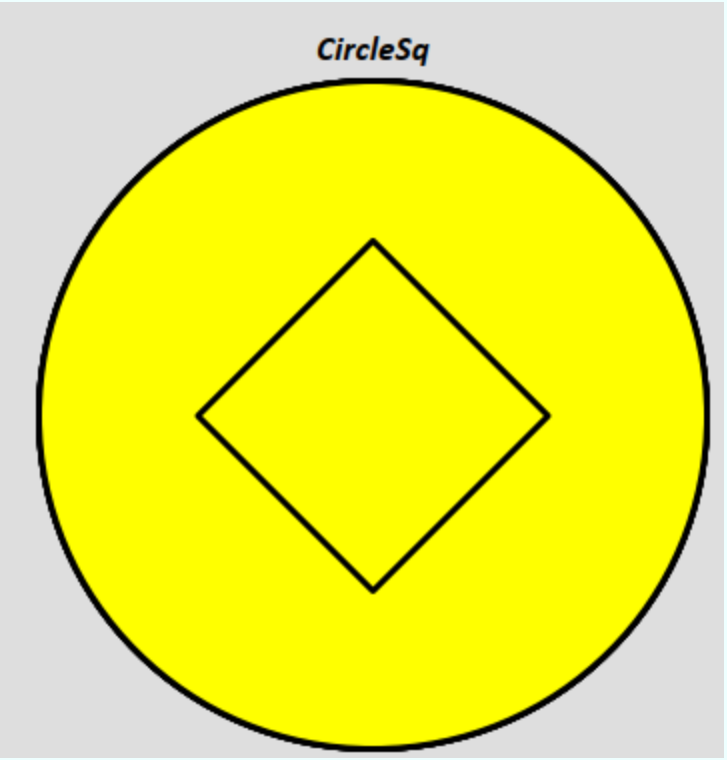


What is a Class?

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

And for a `CircleSq` **object**, we can...

1. Change the color of the circle only
2. Change the color of the square only
3. Change the color of the whole object
4. Rotate the square (rotateSq)
5. . . .



So, for each `CircleSq`, we have a few methods:

- `setCircleColor`
- `setSqColor`
- `setColor`
- `rotateSq`

 All `CircleSq` objects operate independently

What is a Class?

- `CircleSq` is a **class**.
- We can create **objects** based on a class
- **Fields** of an object – data we need to remember
  - `circleColor`
  - `sqColor`
  - `sqAngle`
- **Methods** of an object – operations we can apply to an object
  - `setCircleColor`
  - `setSqColor`
  - `setColor`
  - `rotateSq`

Person

- To deepen our understanding, let's see an example – the `Person` class
- Introducing:
  - Overall structure of a class
  - Constructor
  - Instance variables
- Do It Together Task:
  - Try adding a sayHello method

```
Hello, I am Fname Lname!
```

- Step by step, we will build this class together

Person (V1)

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Overall Structure of a Class

```
public class Person {
    private String lName;
    private String fName;
    private char gender;

    public Person(String firstN, String lastN, char gender) {
        fName = firstN;
        lName = lastN;
        gender = gender;
    }
    public void sayHello(String p) {
        System.out.print("Hello, " + p + "! ");
        System.out.println("I am " + fName + ". Nice to meet you!");
    }
}
```

Class Declaration


```
public class Person {
   ...
}
```

- This is the class declaration
- `public` - indicates the *visibility* of a class. There are four different types of visibility: `public`, package (empty), `protected`, `private`. Will learn it later!
- `class` - a keyword telling the copmiler we are defining a *class*.
- `Person` - the name of the class. Same rule for variable naming. By convention, first character is upper case. By convention, it is a noun.
- All objects created from this class follow this blueprint.

Fields


```
private String lName;
private String fName;
private char gender;
```

- They are **fields** for storing data of an *object*.
- Fields somethings are also called instance variables, data variables, data members, member variables, etc...
- All objects created from this class would have these variables; one object, one set
- `private` - indicates the visibility of a class - not accessible outside this object.

Methods


```
public Person(String firstN, String lastN, char gender) {
    fName = firstN;
    lName = lastN;
    gender = gender;
}
public void sayHello(String p) {
   System.out.print("Hello, " + p + "! ");
   System.out.println("I am " + fName + ". Nice to meet you!");
}
```

- Two methods of the class
- `Person` - having the same name as the class and no return type. This is a **constructor**.
- `sayHello` - another methods.

Constructor


```
public Person(String firstN, String lastN, char gender) {
    fName = firstN;
    lName = lastN;
    gender = gender;
}
```

- A **constructor** is a special method
- No return type
- Must have the same name as the method
- Whenever an object of this class is **instantiated** (i.e. `new`), the constructor is called
- Usually constructor is *public*.
- There could be more than one constructor for a class (with different parameters).

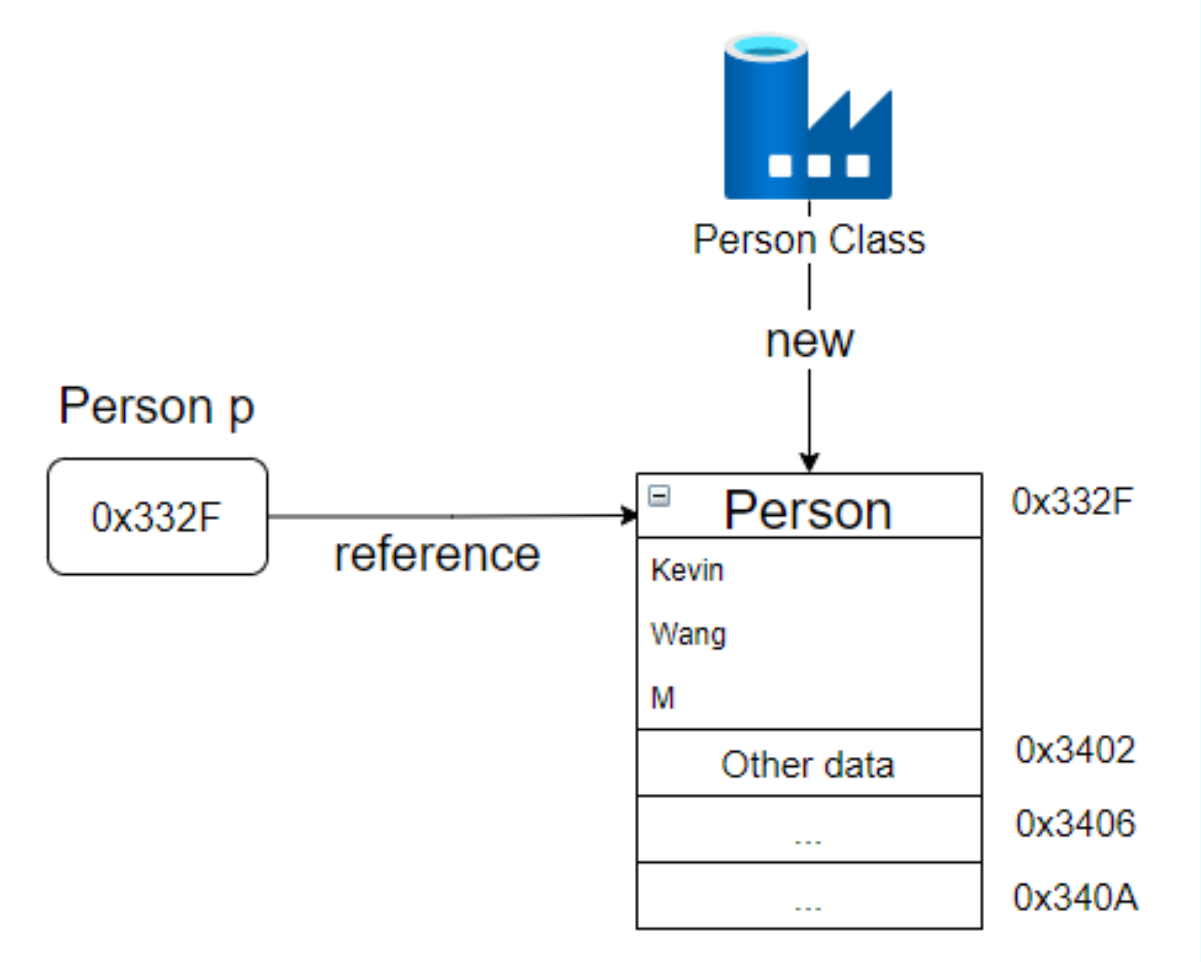
Instantiation


```java
public static void main(String[] args) {
  Person p = new Person("Kevin", "Wang", 'M');
  //...
}
```

- `Person p` - declares the variable `p` with the type `Person`.
- `new` - to create a new object.
- `Person("Kevin", "Wang", 'M')` - the constructor of the class `Person`. It should always follow the keyword `new`.
- The method `main` can be inside or outside the class `Person`.

Instantiation

- The `Person` class acts like a factory to produce(*instantiate*) an object using the **constructor**.
- `Person p` holds the *reference* of the created object.
- An object usually takes up a large block of memory (because it needs to store other fields inside the object).
- `new` operator is to create a new instance of Person.



Dot Operator


```
public static void main(String[] args) {
  Person p = new Person("Kevin", "Wang", 'M');
  p.sayHello("Java");
}
```

- The dot `.` operator always refer to the object on the left side of the dot.
- The method executed using the data from the corresponding object.
- `p1.sayHello("Java")` invokes the `sayHello` method of the new instance `Person`.
- Other than instance methods, the dot `.` operator could be used for referencing instance variables.

Person (V2)

- Adding new method toString

Adding a New Method - `toString`


```
public String toString() {
  return fName + " " + lName + " (" + gender + ")";
}
```

- `public` - visibility
- `String` - the type of **object** returned from the method
- Noted: `String` is a class.

Remember you can actually type `myString.charAt(0)`? This calls the method `charAt` of the object `myString`!

Adding a New Method - `toString`


```
public String toString() {
  return fName + " " + lName + " (" + gender + ")";
}
```

- `toString` needs public and no parameters.
- `gender` looks funny!!

More on `toString`

- The method `toString` is a special method defined in Java.
- When an object is printed, its `toString` method will be automatically called by `System.out.println()`

Invoking the `toString` method, you can:

- Explicitly invoked the method - `p.toString()`

```
Person p = new Person("Kevin", "Wang", 'M');
System.out.println(p.toString());
```

- Implicitly invoked the method and returns a `String`

```
System.out.println(p);
```

The gender Problem


```
Hello, Java! I am Kevin. Nice to meet you!
Kevin Wang ()
```

Wait! gender is not printed!

```
public Person(String firstN, String lastN, char gender) {
    fName = firstN;
    lName = lastN;
    gender = gender;
}
```

The `gender` problem

- Problem: is with the constructor - `gender = gender;`
- `gender` references to the parameter variable, not the **field** (see flipped lecture note Scope).
- Here, we are only updating the parameter variable `gender`.
- Use `this`

`this` is it


```
public Person(String firstN, String lastN, char gender) {
    this.fName = firstN;
    this.lName = lastN;
    this.gender = gender;
}
```

- The problem could be fixed by using `this`! ▶️

The problem explained:

- Inside a method, if a field is within the scope, we can access it implicitly, directly (e.g., `fName` or `lName`)
- Inside a method, if a field (e.g., `gender`) is hidden by another local/parameter variable (e.g., `gender` parameter), the field must be specified using the `this` keyword.

`this`

- The keyword `this` allow you to explicitly refer a **field** rather than a local variable or parameter.
- Recalls from the scope lecture, Java allows a field to have the same name as a local variable or have the same name as a parameter.
- `this` also provide a reference to the object itself (will see more later).

Person (V3)

- Adding getter and setter
- Encapsulation
- `public` & `private`
- `final` keyword

Adding getter and setter


```
public String getfName() {
  return fName;
}
public void setfName(String fName) {
  this.fName = fName;
}
```

- The above are the **getter** and **setter** for the field `fName`.
- A **getter** - to get the value of the field
- A **setter** - to set the value of the field

Adding getter and setter


```
public String getfName() {
  return fName;
}
public void setfName(String fName) {
  this.fName = fName;
}
```

- Typically, an object provides these *helper methods* so that other parts of the program cannot access its fields directly
- The object can control how others access its field (the concept of **encapsulation**)
- As an exercise, add the **getter** & **setter** for `lName` & `gender`

Encapsulation

- Encapsulation is an important concept of OO programming
- An object has all the data about itself
- “It is my own info, my own data, don’t touch!”
- An object manipulates its own data, and less likely to be corrupted by others

Encapsulation

- If others want data from me, use getter
- If others want to update data of me, use setter
- Note: getter and setter are not always provided, as the object may not want others to read or write its own data
- Encapsulation helps protecting an object's data integrity & consistency

`public` and `private`


```
private String fName;

public String getfName() {
    return fName;
}

public void setfName(String fName) {
    this.fName = fName;
}
```

- `public` and `private` are called the *visibility modifier* of variables/methods.
- `public` - all other classes can access it
- `private` - no object from other class is allowed to touch it.

public


```
public String getfName() {
    return fName;
}

public void setfName(String fName) {
    this.fName = fName;
}
```

- `public` – all other classes can access it
- Allow other objects to access my getter and setter methods directly
- May allow access to my other members as well

private


```
private String fName;
```

- private – no object from other class is allowed to touch it.
- Don't want others to access fName directly (e.g., don't want `p.fName = "Calvin";` or `p.gender = 'z';`)
- Protecting others from messing around my data
- Yet, objects from the same class can still access!!

Person (V4)

- Adding the `final` keyword
- Protect our field by final, remove unnecessary setters

The `final` keyword


```
private final String lName;
private final String fName;
private final char gender;
```

- A `final` variable **cannot be updated** once the variable is set for one time.
- Value must be set either **inside the constructor** or **inline initiation**.
- Even the object itself cannot update a `final` variable.
- A `final` variable can be read anywhere.
- Good for avoiding possible programming errors.

Mistakes


```
private final String fName = "Kevin";
public void setfName(String fName) {
  this.fName = fName; //error!
}
```

- This is an error because the field `fName` is final. It can only be set inside a constructor or during inline initiation.
- Also, fName is set to "Kevin" already. It cannot be updated.

Mistakes


```
private final String fName;
public Person() {
  //do nothing
}
```

- This is an error because neither the constructor nor the field inline initiation has set a value for `fName`.

Mistakes


```
private final String fName = "Kevin";
public Person(String fName) {
  this.fName = fName;
}
```

- This is an error because the variable cannot be updated once the variable is set for one time.

Similar, the following is an error too

```
private final String fName;
public Person() {
  fName = "Kevin"; //ok
  fName = "Sandy"; //error
}
```

`final` array

- When `final` mix with array, it could be tricky, e.g.:

```
private final int[] myArray = new int[5];
```

- What you cannot do for sure is to set the variable myArray in your code, like

```
myArray = new int[30]; //Error! because it is final
myArray = anotherArray; //Error! because it is final
```

`final` array

- However, you can actually set the value inside the final array like

```
private final int[] myArray = new int[5];

...
myArray[0] = 10; //OK
myArray[0] = 5; //OK
//myArray is still pointing to that array!
```

The `final` keyword applies to the variable myArray only. This holds the reference of the actually array.

`final` 2D array

- Similarly.. if it is not crazy enough

```
private final int[][] my2DArray = new int[5][4];
...
my2DArray[3][2] = 4; //OK
my2DArray[2][0] = 3; //OK
```

- Even this is allowed

```
my2DArray[2] = new int[8]; //OK
```

Person (V4)


```
public class Person {
  private final String fName;
  private final String lName;
  private final char gender;

  public Person(String firstN, String lastN, char gender) {
    this.fName = firstN;
    this.lName = lastN;
    this.gender = gender;
  }
  public void sayHello(String p) {
    System.out.print("Hello, " + p + "! ");
    System.out.println("I am " + fName + ". Nice to meet you!");
  }
  public String toString() {
    return fName + " " + lName + " (" + gender + ")";
  }

 //getter and remove setter
  public String getfName() {
    return fName;
  }

  public char getGender() {
    return gender;
  }
}
```

Person (V5)

- Create more `Person`s to interact each other
- Learn the keyword `this`
- Adding the `greet` method
- Adding the field `friends`
- Adding the methods `makeFriend` and `listFriends`

Greet


```
Person kevin = new Person("Kevin", "Wang", 'M');
Person sandy = new Person("Sandy", "Lo", 'F');
kevin.greet(sandy);
```

It should prints:

```
Hi, Sandy! My name is Kevin.
```

- Similar to `sayHello` except the method should be able to extract the name of the person we greet.

Greet

- At the first glance, we might work out something like this:

```
public void greet(Person a) {
  System.out.print("Hi, " + a.getfName());
  System.out.println("! My name is " + getfName() + ".");
}
```

- `a.getfName()` returns `fName` of the object `a`.
- `getfName()` is the same as `this.getfName()`, returns `fName` of this object.
- In the example `kevin.greet(sandy)`:
  - `a.getfName()` - Sandy
  - `fName()` - Kevin
- It works!

Greet

- In fact we don't need getter here, because `greet()` method is inside the class `Person`, we are allowed to use the private variable directly!

```
public void greet(Person a) {
  System.out.print("Hi, " + a.fName);
  System.out.println("! My name is " + fName + ".");
}
```

- `a.fName` refers to the field of the object `a`.
- `fName` is the same as `this.fName`, refers to `fName` of this object.
- It also works!

A private variable forbids only other **classes** access it but not other **objects**

Making Friends


```
//in Person class:
private Person[] friends = new Person[5];
//max 5 friends

...
//in main
kevin.makeFriend(sandy);
kevin.makeFriend(karsten);
kevin.listFriends();
karsten.listFriends();
```

```
Friends of Kevin:
Sandy Lo (F)
Karsten Kwong (M)

Friends of Karsten:
Kevin Wang (M)
```

- We want to keep a list of *friends* in a Person
- Friend is a *mutual* relationship
- Assume we can't make more than 5 friends

Making Friends


```
//in Person class
private Person[] friends = new Person[5];
int numOfFriend = 0;
```

- need an extra counter `numOfFriend` to tell how many friends I am having now

```
public void makeFriend(Person a) {
  if (numOfFriend < 5)
    friends[numOfFriend++] = a;
}

public void listFriend() {
  System.out.println("\nFriends of " + fName + ":");
  for (int i = 0; i < numOfFriend; i++)
    System.out.println(friends[i]);
}
```

Making Friends

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

- The code got a little problem

```
kevin.makeFriend(sandy);
kevin.makeFriend(karsten);
kevin.listFriends();
karsten.listFriends();
```

```
Friends of Kevin:
Sandy Lo (F)
Karsten Kwong (M)

Friends of Karsten:
```

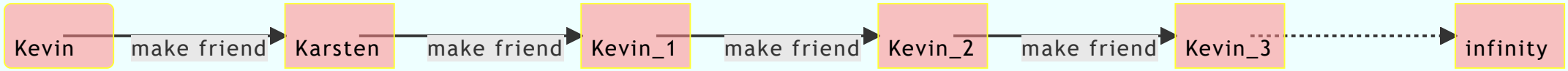
- Need to respond to the friend request

image: https://me.me/i/the-most-dangerous-creature-on-this-earth-is-a-fake-12511313

Make Friends - Attempt 1


```
public void makeFriend(Person a) {
  if (numOfFriend < 5)
    friends[numOfFriend++] = a;
  a.makeFriend(new Person("Kevin", "Wang", 'M'));
}
```

- It does not work!
- You are asking Karsten to make friend with a clone of Kevin.
- And in fact this will ends up in an **infinite recursion**!



Make Friends - Attempt 2

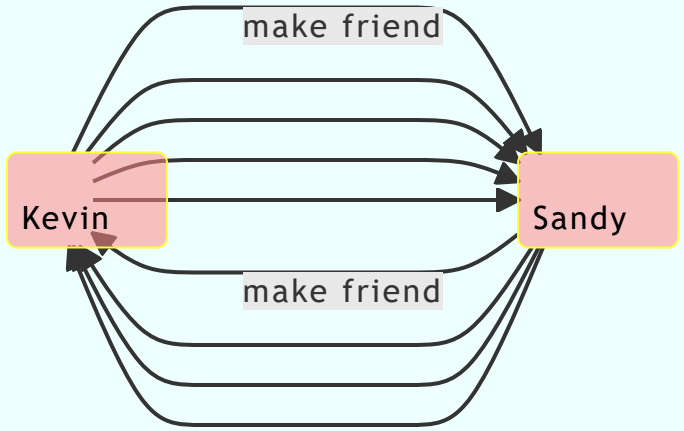

```
public void makeFriend(Person a) {
  if (numOfFriend < 5) {
    friends[numOfFriend++] = a;
    a.makeFriend(this);
  }
}
```

- The keyword `this` refer to the **reference** of the current object.
- The reference of an object is the address of the object.
- The statement `a.makeFried(this);` is to ask `a` make `this` as a friend.

This one is also incorrect, why?

Make Friends - Attempt 2


```
kevin.makeFriend(sandy);
kevin.listFriends();
sandy.listFriends();
```




```
Friends of Kevin:
Sandy Lo (F)
Sandy Lo (F)
Sandy Lo (F)
Sandy Lo (F)
Sandy Lo (F)

Friends of Sandy:
Kevin Wang (M)
Kevin Wang (M)
Kevin Wang (M)
Kevin Wang (M)
Kevin Wang (M)
```

- The methods `kevin::makeFriend` and `sandy::makeFriend` calls each other *recursively*.
- We had unintentionally discovered a recursion.
- Will talk about that in the very last part of our course.

Make Friends - Attempt 3


```
public void makeFriend(Person a) {
  if (numOfFriends == 5 || a.numOfFriends == 5) //full
      return;
  if (isFriend(a) || a.isFriend(this)) //already friend
      return;
  friends[numOfFriends++] = a;
  a.friends[a.numOfFriends++] = this;
}
```

- This requires a method `isFriend` which determines if `a` is a friend of mine.

The method `isFriend`


```
private boolean isFriend(Person a) {
  for (Person p : Friends) {
    if (p == a)
      return true;
  }
  return false;
}
```

- Why setting the method private? If it is not needed by other objects, just make it private
- The less fields and methods exposed, the cleaner the method would be
- Expose on the necessary

Final Version


```
public class Person {
  private final String fName;
  private final String lName;
  private final char gender;
  private final Person[] friends = new Person[5];
  private int numOfFriends = 0;

  public Person(String firstN, String lastN, char gender) { ... }
  public void greet(Person a) {  ...  }
  public void makeFriend(Person a) { ... }
  public void listFriends() { ... }
  public void sayHello(String p) { ... }
  public String toString() { ... }
  public String getfName() { ... }
  public char getGender() { ... }

  private boolean isFriend(Person a) { ... }
}
```

Are these allowed?

Assume the following codes are executed in the `Main` class.

1.

```
kevin.sayHello(sandy);
```

2.

```
kevin.makeFriend(kevin);
```

3.

```
kevin.greet(this);
```

4.

```
final Person p = new Person("final",
     "person", 'F');
p.makeFriend(kevin);
```

Are these allowed?

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Assume the following codes are executed in the `Main` class.

1.

```
kevin.sayHello(sandy);
```

✗ `sayHello` requires a String object, not a `Person` object

2.

```
kevin.makeFriend(kevin);
```

✓ Weird, but syntactically ok. Kevin will make himself friends twice!

3.

```
kevin.greet(this);
```

✗ `this` refer to the current object, which is an object of `Main` class.

4.

```
final Person p = new Person("final",
      "person", 'F');
p.makeFriend(kevin);
```

✓ `final` means the reference of `p` cannot be changed. The object itself is still *mutable*.

Errata

Update: 19/10/2021

- Page 13: Person 1 `sayHello` method correct to void method.