# Object and classes

# บทนี้เรียนอะไรบ้าง

- แนะนำ Object-oriented programming
- บอกวิธีการสร้าง object จากคลาสที่จาวาให้มา
- สอนวิธีสร้างคลาสของเราเอง

# แนะนำ Object-Oriented Programming

- โปรแกรมประกอบขึ้นจากออบเจ็กต์
- แต่ละออบเจ็กต์จะมีฟังก์ชั่นให้เรียกใช้
  - คนเรียกใช้ไม่จำเป็นต้องรู้ว่าข้างในออบเจ็กต์ทำงานยังไง
  - ขอให้ใช้ได้ตามที่ต้องการก็พอ
- วิธีคิดในการโปรแกรม
  - เมื่อต้องการแก้ปัญหา ให้ออกแบบตัวข้อมูลที่ต้องใช้ก่อน
  - แล้ววิธีแก้ปัญหาค่อยตามมา

# คลาส

- คลาสคือ template ที่เราจะใช้สร้างออบเจ็กต์
  - พูดง่ายๆ มันเป็นตัวกำหนดว่า ออบเจ็กต์ที่สร้างจากมัน จะมีข้อมูลอะไรอยู่ภายในบ้าง
  - คลาสเปรียบเสมือนแม่พิมพ์ ส่วนออบเจ็กต์ก็เป็นสิ่งที่แม่พิมพ์พิมพ์ออกมา
- เมื่อเราสร้างออบเจ็กต์ขึ้นจากคลาส
  - เราเรียกเหตุการณ์นี้ว่า การสร้าง instance ของคลาส

- ภายในแต่ละออบเจ็กต์
  - ข้อมูล หรือดาต้า เราเรียกว่า instance fields หรือ instance variables
    - แต่ละออบเจ็กต์ จะมีค่าของแต่ละ instance field เป็นของตนเอง ไม่ใช้ค่าร่วมกับออบเจ็กต์อื่น
    - สถานะของค่าตัวแปรเหล่านี้ทั้งหมด ถือเป็น state ของออบเจ็กต์
    - ซึ่งเมื่อเรียกเมธอด state ของออบเจ็กต์อาจเปลี่ยนได้
  - ส่วนฟังก์ชั่นที่ออบเจ็กต์นั้นเรียกเพื่อจัดการข้อมูลข้างต้น เรียกว่า methods

- ห้ามเข้าถึงค่าของ instance field โดยตรง
- ต้องอ่าน และเขียนค่าตัวแปร จากเมธอดเท่านั้น
- ทั้งนี้เพื่อ
  - ป้องกันการยัดค่าผิดๆ ใส่ลงไปใน instance field
    - เช่น a.x = 100; จริงๆ x อาจจะมีค่าเกิน 99 ไม่ได้ ดังนั้นเขียนเมธอดคุมดีกว่า
  - เวลาจะเปลี่ยนพฤติกรรมการทำอะไร จะได้แก้ในเมธอด ทีเดียว ไม่ต้องมาแก้หลายๆที่
-

# extends

- ถ้าเราสร้างคลาสใหม่ โดยใช้การ extends จากคลาสเก่า
  - คลาสใหม่จะได้รับ ตัวแปร และเมธอด มาจากคลาสเก่าทั้งหมด
  - เราแค่เขียนตัวแปรกับเมธอดเพิ่ม ที่คลาสใหม่จะมีเท่านั้น ก็พอ
- การ extends นั้น ใช้หลักของการ inheritance ซึ่งจะสอนในบทต่อไป

# Relationship between classes

- แต่ละคลาส สัมพันธ์กันยังไงได้บ้าง
- Dependence (uses-a)
  - คลาส Aจะ depend on คลาส B ถ้าคลาส A มีการใช้ออบเจ็กต์ของคลาส B
- Aggregation (has-a)
  - ออบเจ็กต์ในคลาส A สามารถเก็บ ออบเจ็กต์ในคลาส B ได้ นี่เรียกว่าเป็น aggregation
- Inheritance (is-a)
  - Car extends จาก Vehicle
  - Car จะถือว่า เฉพาะเจาะจง หรือ specialise กว่า สามารถทำสิ่งที่ Vehicle ทำได้ และก็ทำอย่างอื่นได้เพิ่มเติมด้วย
  - Car is a Vehicle แต่ว่า Vehicle ไม่จำเป็นต้องเป็น Car

# การวาดความสัมพันธ์

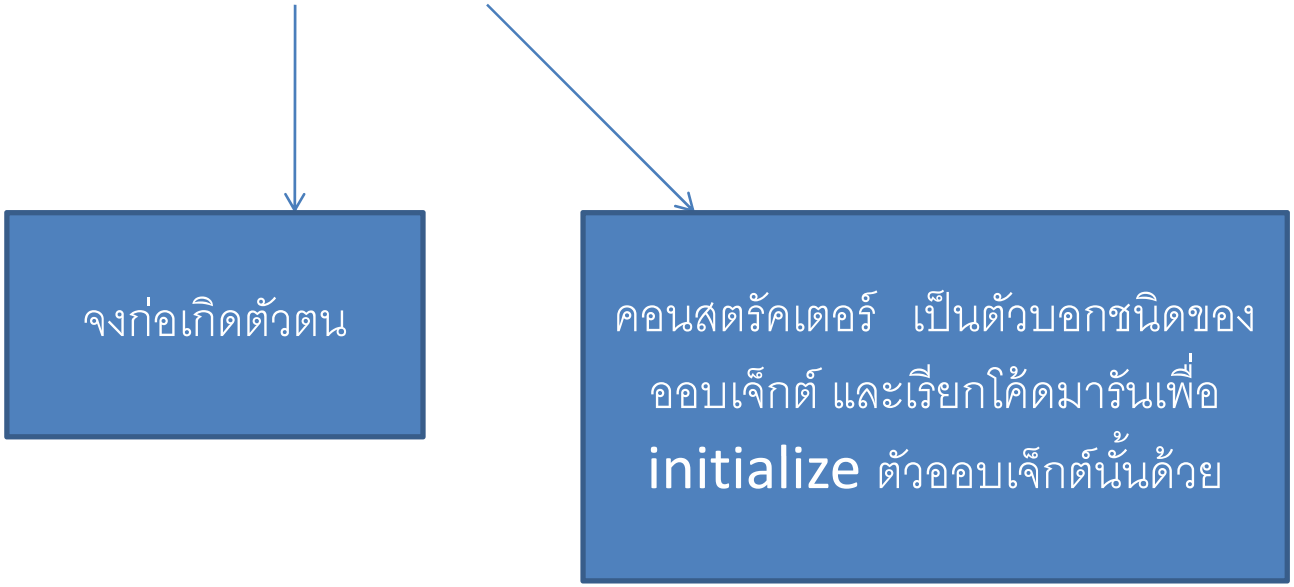
- เดี๋ยวนี้ใช้ UML Diagram
  - ปกติ คลาสจะวาดด้วยสี่เหลี่ยม
  - ความสัมพันธ์จะวาดเป็นเส้นต่างๆกันไป
  - ไปลองใช้ ArgoUML (http://argouml.tigris.org) หรือ Violet (http://violet.sourceforge.net)

# การใช้งานคลาสที่จาวาสร้างไว้แล้ว

- จะใช้งานออบเจ็กต์ได้นั้น ต้อง
  - สร้างออบเจ็กต์
  - กำหนดค่าต่างๆสำหรับข้อมูลภายในออบเจ็กต์
  - ใช้งานโดยการเรียกเมธอดที่ใช้กับออบเจ็กต์นั้นได้ (แน่นอนว่าต้องนิยามเมธอดก่อน)

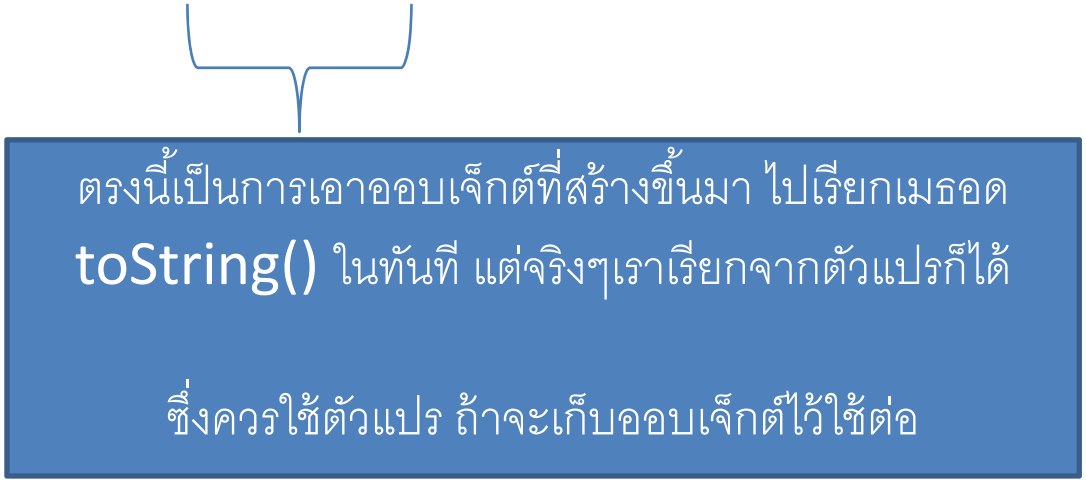
# มา zoom ดูที่การสร้างออบเจ็กต์

```
Date x = new Date();
```



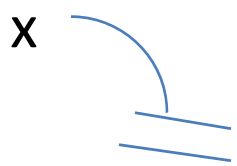
# พอสร้างเสร็จก็ใช้งานได้

- เช่นเอาไปปริ๊น (แต่ต้องนิยาม **toString()** ก่อน เพราะจาวา ปริ๊นออบเจ็กต์ไม่เป็น เราต้องบอกมันว่าจะปริ๊นยังไง)
  - System.out.println(new Date()); หรือ
  - String s = new Date().toString();



# สภาพ ณ เวลาต่างๆ

- Date x

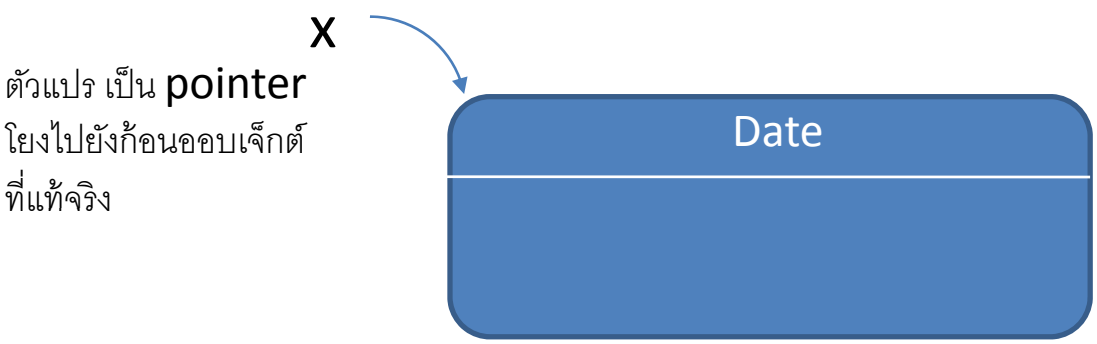



มีตัวแปร **x** ซึ่งสามารถชี้ไปที่ออบเจ็กต์ที่เป็นไทป์ Date ได้
แต่ยังไม่มีตัวออบเจ็กต์ที่จะให้ชี้เลย

- เมื่อยังไม่มีตัวตนออบเจ็กต์ จึงเรียกเมธอดไม่ได้
  - ฉะนั้นคำสั่ง s = x.toString(); จึง error

- ดังนั้นต้องสร้างออบเจ็กต์ให้มีตัวตน เพื่อเอา **x** ชี้ไปที่ออบเจ็กต์นั้น
- มีทางเลือกสองทาง
  - ใช้คอนสตรัคเตอร์: x = new Date();
  - ทำตัวแปรให้ชี้ไปที่ตัวแปรอื่นที่เคยสร้างออบเจ็กต์สมบูรณ์มาก่อนแล้ว: x = birthday;

- Date x = new Date();




- ถ้าเราทำต่อจากข้างบนนี้ โดยเขียนว่า Date x2 = x;


จะเห็นว่า ตัวแปรสองตัวจะชี้ไปที่ก้อนออบเจ็กต์เดียวกัน ดังนั้นถ้าเปลี่ยน Date object ตัวนี้ จาก x ก็จะเห็นความเปลี่ยนแปลงถ้าดูจาก x2 ด้วยเหมือนกัน

- เราสามารถบอกตัวแปรว่า ไม่ให้ชี้ไปไหน ก็ได้
  - X = null;
- สามารถใช้ null ในการตรวจสภาพออบเจ็กต์ได้
  - if (x != null){  }
- แต่อย่าลืมว่า ถ้าตัวแปรไหนเป็น null แล้ว จะเรียกเมธอดใช้ไม่ได้ error หมด

# ตัวอย่างคลาส

- Date
  - ออบเจ็กต์ไทป์นี้ เป็นข้อมูลจุดหนึ่งของเวลา
  - มีเมธอดให้เราเปรียบเทียบกับ Date อื่น ว่า Date ไหนมาก่อนมาหลัง เช่น

```
if(today.before(birthday))
    System.out.println("Have some time left.");
```

- GregorianCalendar
  - Extends มาจาก Calendar ที่นิยามปฏิทินอย่างทั่วไป
  - มีเมธอดเยอะ มีคอนสตรัคเตอร์หลายตัว เช่น

    new GregorianCalendar() ซึ่งสร้างออบเจ็กต์ที่เก็บวันเวลา ณ ตอนสร้าง

    new GregorianCalendar(1999, 11, 31) ซึ่งสร้างออบเจ็กต์ที่เก็บข้อมูล เวลาเที่ยงคืน วันที่ 31 ธค 1999 (เดือน เริ่มนับเดือนแรกที่เลข 0)

หรือว่าจะใช้คอนสตรัคเตอร์ที่ละเอียดกว่า ที่กำหนดเวลาได้ด้วย ก็ได้

สังเกต ว่าเราไม่จำเป็นต้องรู้ว่าข้างในเก็บวันเวลา เดือนปี ยังไง แค่เรียกใช้เมธอดได้ก็พอ

# Mutator and accessor methods

- จะจัดการกับค่าต่างๆที่เก็บไว้ จากตัวอย่าง Date ได้ต้อง
- อ่านได้ และเขียนได้
- สำหรับการอ่านค่า ใช้ accessor methods (หมายถึง เมธอดที่ใช้อ่านค่าต่างๆในออบเจ็กต์) ตัวอย่างเช่น

```
// construct d as current date
    GregorianCalendar d = new GregorianCalendar();

    int today = d.get(Calendar.DAY_OF_MONTH);
    int month = d.get(Calendar.MONTH);
```

- การเปลี่ยนค่าต่างๆภายในออบเจ็กต์ ใช้ mutator methods (เมธอดที่ใช้สำหรับเปลี่ยนค่า) ตัวอย่างเช่น

```
d.set(Calendar.DAY_OF_MONTH, 15);
d.set(Calendar.YEAR, 2001);
d.set(Calendar.MONTH,  Calendar.APRIL);
```

ทั้งนี้ขึ้นกับว่าคลาสนั้นมีเมธอดอะไรให้เปลี่ยนค่าของข้อมูลบ้าง อย่างคลาสปฏิทิน นี่ มีเมธอดที่ช่วยเลื่อนวันเวลา ด้วย คือ

```
d.add(Calendar.MONTH, 3);
```

เป็นการเลื่อนวันเวลา ของ d ออกไปสามเดือน

- โดยปกติแล้ว accessor จะมีคำนำหน้าว่า get ส่วน mutator จะมีคำนำหน้าว่า set ตัวอย่างเช่น

```
Date time = calendar.getTime();
```

เอาค่าวันเวลา (ที่เป็นชนิด Date) ออกจากปฏิทิน มาเก็บไว้ในตัวแปร time

สมมุติว่าออบเจ็กต์นี้มีอยู่แล้ว เป็นชนิด Calendar นะ

```
calendar.setTime(time);
```

เซตเวลา ของ calendar object

# ตัวอย่างเพิ่มเติม

- รู้วันเดือนปี และต้องการสร้าง Date ที่แทนวันเดือนปีนั้น
  - ตอนนี้ Date ไม่มีเมธอดที่ทำได้ จึงต้องใช้ calendar ช่วย

```
GregorianCalendar cal = new GregorianCalendar(year,
    month, day);
Date hireDay = calendar.getTime();
```

- มี Date object อยู่ ต้องการหาค่าต่างๆจาก Date object นั้น

```
GregorianCalendar cal = new GregorianCalendar();
Cal.setTime(hireDay);
int year = cal.get(Calendar.YEAR);
```

# ตัวอย่างโปรแกรม วาดปฏิทินของเดือนนี้

- calendar

# การนิยามคลาสไว้ใช้เอง

- คลาสที่เราจะนิยามจากนี้ไป ไม่จำเป็นต้องมี main
- แต่จะมี field กับเมธอด
- การเขียนโปรแกรม จะเกิดจากการใช้ main method ของคลาสหนึ่ง เรียกใช้เมธอดของคลาสอื่นๆ

- เดี๋ยวลองมาดูคลาสสำหรับเก็บข้อมูลลูกจ้าง เอาไว้ใช้กับการจ่ายเงินเดือนดู

```
class Employee{
    //constructor
    public Employee(String n, double s, int year, int month, int day){
            name = n;
            salary = s;
            GregorianCalendar calendar = new GregorianCalendar(year, month-1,
    day);
            hireDay = calendar.getTime();
    }

    //method
    public String getName(){
            return name;
    }
     …
     //instance fields
    private String name;
    private double salary;
    private Date hireDay;
```

# ตัวอย่างโปรแกรมที่ใช้งาน Employee

- EmployeeTest

- ใช้งานจริงๆแค่สร้างอาเรย์แล้วเติมรายละเอียดลูกจ้างสามคนลงไป
- จากนั้นลูปเพิ่มเงินเดือนทุกคน คนละ 5%
- แล้วพิมพ์รายละเอียดของทุกคนออกมา
  - อันนี้เรียกเมธอดหลายอัน เพื่อให้ได้ข้อมูลทุกส่วน
- สังเกต
  - ว่า main จะอยู่ใน EmployeeTest เท่านั้น
  - ชื่อไฟล์จะต้องตรงกับชื่อคลาสที่เป็น public
  - ในหนึ่งไฟล์มี public class ได้แค่อันเดียว แต่นอกนั้นจะมีกี่คลาสก็ได้
  - ตอนคอมไพล์ มันจะแยกไฟล์ .class ให้
  - เวลารัน ต้องรันคลาสที่มี main เสมอ

- ถ้าเราแยกไฟล์แต่แรก ตอนคอมไพล์ สามารถใช้

javac Employee*.java เพื่อคอมไพล์ทุกไฟล์ได้

หรือ เรียก

Javac EmployeeTest.java

จาวาจะคอมไพล์ EmployeeTest เมื่อมันเห็นว่า มีการใช้ Employee ภายใน มันจะไปหา Employee.class ซึ่งถ้าหาไม่เจอ มันจะคอมไพล์จาก Employee.java ให้เอง

นอกจากนี้ ถึงหาเจอ แต่ไฟล์ .class นั้นเก่ากว่า .java ที่มี มันก็จะคอมไพล์ใหม่ให้เหมือนกัน

# มาดู Employee กันอย่างละเอียดหน่อย

- สังเกตได้ว่า
  - ทุกเมธอดเป็น public นั่นคือ เมธอดทุกเมธอดในทุกคลาสสามารถเรียกเมธอดเหล่านี้ได้
  - ทุกตัวแปรเป็น private นั่นคือ เมธอดที่อ่านค่าจากตัวแปรเหล่านี้ได้โดยตรงจะต้องเป็นเมธอดจากคลาส Employee เท่านั้น
    - ไม่ควรให้ตัวแปรเป็น public เพราะส่วนไหนของโปรแกรมจะมาเปลี่ยนข้อมูลก็ได้ เปลี่ยนอย่างไรก็ได้ อาจจะมีการกระทำต้องห้ามที่เราไม่ทันคิด เกิดขึ้นได้
  - ตัวแปร สองตัว ก็เป็นออบเจ็กต์ name เป็น String ส่วน hireDay เป็น Date
    - ใช่แล้ว ออบเจ็กต์สามารถใส่ออบเจ็กต์อื่นๆ ซ้อนๆกันได้

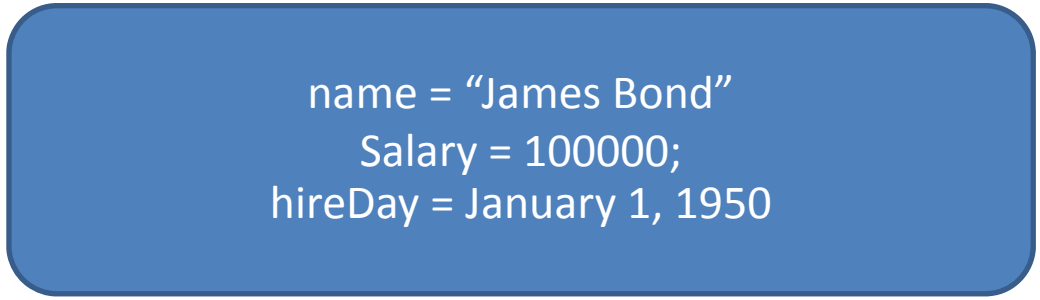
คอนสตรัคเตอร์ไม่รีเทิร์นค่า

# constructors

```
public Employee(String n, double s, int year, int
   month, int day)
  {
    name = n;
    salary = s;
    GregorianCalendar calendar = new
   GregorianCalendar(year, month - 1, day);
    // GregorianCalendar uses 0 for January
    hireDay = calendar.getTime();
  }
```

ชื่อต้องเหมือนชื่อคลาส คอนสตรัคเตอร์ใช้เซตค่าภายในออบเจ็กต์ที่พึ่งถูกสร้าง ให้เป็นไปตามที่เราต้องการ

new Employee(“James Bond, 100000, 1950, 1, 1); จะได้ object ของคลาส Employee ที่มีข้างในเป็น

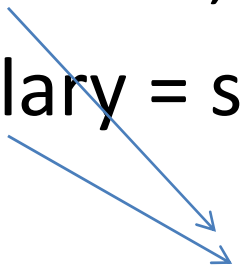



คอนสตรัคเตอร์ เรียกใช้ได้พร้อม new เท่านั้น ดังนั้นจะไม่สามารถใช้กับออบเจ็กต์ที่ถูกสร้างขึ้นแล้วได้

# ข้อควรระวัง

- อย่าให้ local variable มีชื่อเดียวกับ field ที่เรานิยามไว้

```
public Employee(String n, double s, int year, int
    month, int day)
  {
     String name = n;
     double salary = s;
```



จะกลายเป็นให้ค่ากับ local variable ไป ไม่ได้ให้ค่าอะไรกับ fields

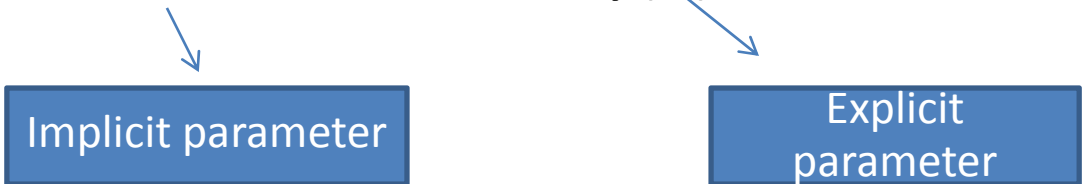
# Implicit and explicit parameters

จาก

```
public void raiseSalary(double byPercent)
  {
    double raise = salary * byPercent / 100;
    salary += raise;
  }
```

ถ้าเราเรียก

```
Employee number007 = new Employee(“James Bond,
   100000, 1950, 1, 1);
number007.raiseSalary(5);
```



- ซึ่งโค้ดที่รันไป จะเหมือนกับได้ทำ

```
double raise = number007.salary * byPercent / 100;
    number007.salary += raise;
```

ซึ่งถ้าเราอยากให้มีการอ้างถึง implicit parameter เราจะทำได้โดยใช้คำว่า this ตัวอย่างเช่น ในโค้ดของคอนสตรัคเตอร์เราอาจจะใช้

```
double raise = this.salary * byPercent / 100;
    this.salary += raise;
```

# คราวนี้มาลองดูเมธอดในคลาส Employee กันหน่อย

```
public String getName()
  {
    return name;
  }

  public double getSalary()
  {
    return salary;
  }

  public Date getHireDay()
  {
    return hireDay;
  }
```

พวกนี้เป็น accessor

- name ต้องไม่เป็น public เพราะทางผู้เขียนไม่ต้องการให้ name ถูกเปลี่ยนหลังจากให้ชื่อผ่านคอนสตรัคเตอร์ไปแล้ว ดังนั้น พอเป็น private จึงป้องกันการถูกเปลี่ยนชื่อได้

- salary เปลี่ยนได้ก็จริง แต่ถ้าเป็น private แบบนี้จะเปลี่ยนได้ด้วย raiseSalary เท่านั้น ซึ่งถ้าเกิดอะไรผิดพลาด เราก็จะได้ไปแก้ที่ raiseSalary ที่เดียว ถ้าเป็น public แล้วละก็ กว่าจะตามหาที่ผิดเจอ จะยากมาก

# การใช้เมธอดในการเปลี่ยนข้อมูลเท่านั้น มีประโยชน์จริงๆ

จากตัวอย่าง Employee ถ้ามีการเปลี่ยนรูปแบบของชื่อเป็น

String firstName;

String lastName;

เวลาเราต้องการดึงชื่อของ Employee ออกมา เราก็ทำแค่ไปแก้ getname เท่านั้น อาจแก้ให้เป็น return firstName + " " + lastName;

ซึ่งจะไม่ต้องมีผลกระทบกับส่วนอื่นของโปรแกรม

นอกจากนี้ เมธอดยังตรวจ error ไปในตัวได้ เช่นให้โค้ดของเมธอดเช็คก่อนเสมอว่า salary ต้องไม่ต่ำกว่า 0 ดังนั้น เมื่อบังคับว่าต้องเรียกเมธอดเพื่อเปลี่ยน salary ก็จะทำให้ค่า salary ไม่มีวันต่ำกว่า 0 อย่างแน่นอน

# ระวัง

```
public Date getHireDay()
  {
     return hireDay;
  }
```

จริงๆ โค้ดนี้ไม่เวิร์ค เพราะว่า พอเรารีเทิร์น Date ออกไปได้ เช่น

```
Date d = harry.getHireDay();
d.setTime(...);
```

พอเปลี่ยนค่าในตัวแปร d ค่า hireDay ที่เป็นของ harry ก็จะเปลี่ยนไปจริงๆ เลยกลายเป็นว่า ไม่ได้เปลี่ยนโดยใช้ setHireDay() หรือเมธอดที่ควรจะเตรียมไว้ในคลาส Employee

- ดังนั้น เพื่อป้องกันไม่ให้ ค่าภายในออบเจ็กต์ต้นฉบับ เปลี่ยนแปลง
  - ตัวออบเจ็กต์ ที่รีเทิร์นออกมาจากเมธอด ควรเป็นตัวก็อปปี้
  - ใช้การโคลน


```
public Date getHireDay()
  {
    return (Date) hireDay.clone();
  }
```

# Access privileges

- เมธอดสามารถเข้าถึง private data ของออบเจ็กต์ที่รันเมธอดนั้น
- นอกจากนี้ เมธอด สามารถเข้าถึง private data ของออบเจ็กต์ทุกออบเจ็กต์ที่เป็นชนิดเดียวกัน ตัวอย่างเช่น เมธอดที่เปรียบเทียบลูกจ้างสองคน

```
class Employee{
boolean equals(Employee other){
    return name.equals(other.name);
}
...
```

เป็นออบเจ็กต์อันอื่น แต่เพราะเป็นชนิดเดียวกัน java เลยให้อ่านค่าได้โดยตรง

# Private methods

- ส่วนใหญ่ เมธอด เราจะให้เป็น public
- แต่ private method ก็มีได้เหมือนกัน เพื่อ
  - เป็นเมธอดใช้ภายในคลาสเท่านั้น อาจเป็นส่วนที่แยกย่อยออกมาของเมธอดอื่น และตั้งใจให้ใช้โดยคลาสที่นิยามเท่านั้น

อย่าลืมว่า ถ้าเป็น public คลาสอื่นจะเรียกใช้ได้เสมอ

# Final instance fields

- คือ ตัวแปร ที่ เราต้องการให้เป็นค่าคงที่หลังจากการรันคอนสตรัคเตอร์แล้ว
- มักใช้กับตัวแปรที่เป็น primitive type หรือ เป็น immutable class (หมายความว่า คลาสนั้นไม่มีเมธอดที่เปลี่ยนสิ่งที่อยู่ข้างในออบเจ็กต์เลย ตัวอย่าง immutable class ก็เช่น String นั่นเอง)
- ระวัง

private final Date hireDate; หมายถึง ตัว pointer ที่ชื่อ hireDate จะไม่เปลี่ยนที่ชี้ แต่ตัวออบเจ็กต์จริงๆยังถูกเปลี่ยนข้างในได้

# Static fields and methods

- ถ้าให้ตัวแปร a เป็น static
  - จะมีตัวแปรนี้ตัวเดียวต่อหนึ่งคลาสเท่านั้น (นั่นคือ มี แม้ว่าจะยังไม่มีออบเจ็กต์เกิดขึ้นเลย)
  - ถึงแม้จะมีออบเจ็กต์ของคลาสนี้เกิดขึ้นมาหลายๆอัน ตัวแปร a ก็ยังมีตัวเดียวอยู่ดี (ทุกออบเจ็กต์ของคลาสนี้จะถือว่า แชร์ค่า a เดียวกัน) แต่ละออบเจ็กต์ไม่มีของมันเอง ตัวอย่างเช่น

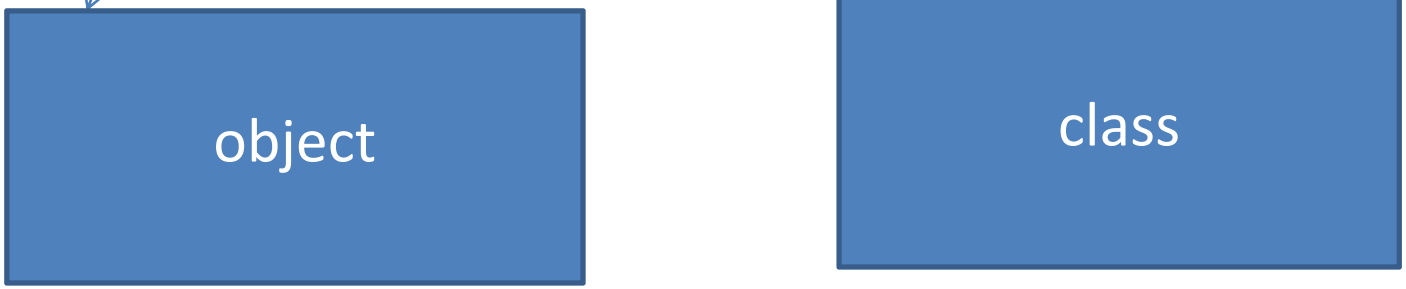
```
class Employee{
    private int id;
    private static int nextId =1;
    public void setID(){
      id = nextId;
      nextId++;
    }
...
```

ต่อหน้าถัดไป

สมมุติเรามีออบเจ็กต์ Employee ซึ่งเก็บในตัวแปร harry อยู่ก่อนแล้ว

harry.setId(); จะหมายความเหมือนกับ

```
harry.id = Employee.nextId;
Employee.nextId++;
```



# Static constants

- มีเยอะ เช่นในคลาส Math มี

public static final double PI = 3.141592653589….

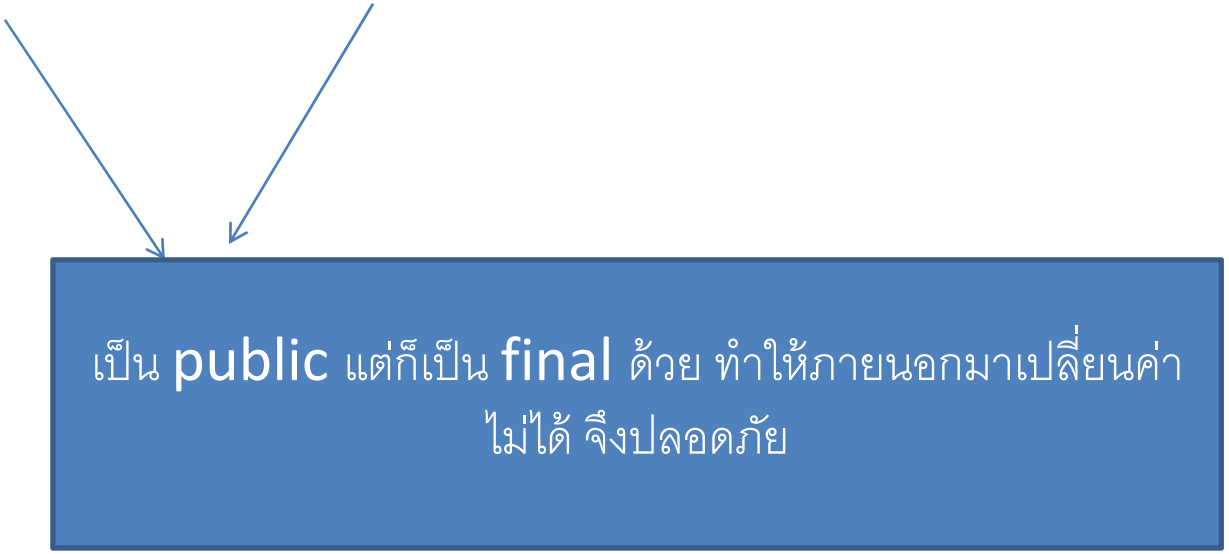
ซึ่งเราสามารถอ่านค่าได้ โดยไม่ต้องสร้างออบเจ็กต์ใดๆขึ้นมาก่อนเลย

ดังเช่น double x = Math.PI;

- ซึ่งถ้าไม่ใช่ static แล้วละก็ จะถือว่า PI เป็น instance field ของคลาส Math
  - ซึ่งจะอ่านค่า PI ได้คราวนี้ ต้องสร้างออบเจ็กต์ก่อน
  - แต่ละออบเจ็กต์ที่สร้าง ก็จะมี PI ของตนเอง (แม้ว่าจะเป็นค่าคงที่ตลอดก็ตาม)

- อีกตัวอย่างของ static constant ที่เราได้ใช้บ่อย คือ System.out ซึ่งจริงๆแล้วโค้ดนิยามของมันเป็นดังนี้

```
public class System{
    public static final PrintStream out = …;
    …
}
```



- จริงๆ คลาส System มีเมธอด setOut ซึ่งเปลี่ยนค่าของ out ได้ แต่ว่า มันเป็น native method ซึ่งเขียนด้วยภาษาอื่น เลยไม่โดนจาวาจำกัดการทำงาน

# Static methods

- คือเมธอดที่ไม่ทำการอ่านหรือเขียนลงบนออบเจ็กต์ เช่น

Math.pow(x, a);

- จากภายใน static method เราจะอ่าน instance field ไม่ได้
- จริงๆ เราเรียก static method จากออบเจ็กต์ก็ได้ แต่จริงๆไม่ควรทำ คนเขียนจาวามันทำไว้ให้เขียนสะดวก แค่นั้น แต่จริงๆความหมายมันผิด

- ใช้งานในกรณีใดบ้าง
  - เมธอดไม่ได้ต้องการอ่านหรือเขียนค่าตัวแปรภายในออบเจ็กต์
    - กรณีนี้อาจเกิดจากข้อมูลทั้งหมด ได้จาก พารามิเตอร์ของเมธอด
  - เมื่อเมธอดอ่าน หรือเขียน แค่ static field
  - Main method ใช้เป็นตัวทดสอบคลาสนั้นๆได้

# Method parameters

- Java ใช้ call by value
  - นั่นคือ ค่าพารามิเตอร์ของเมธอด จะเป็นค่าก๊อปปี้ ไม่ใช่ค่าของตัวแปรที่ให้

  add(a,b) -> ได้ค่าของ a กับ b ไปใช้งานภายในเมธอด ตัว a กับ b จริงๆ จะไม่เปลี่ยนแปลง

```
public static void tripleValue(double x){
   x = 3x;
}
```

ถ้าเราเรียกใช้โค้ดดังนี้

...

```
double percent =10;
tripleValue(percent);
```

percent จะยังคงมีค่า 10 อยู่หลังเรียกเมธอด เพราะไอ้ที่เมธอดมันเอาไปคำนวณน่ะเป็นตัวก๊อปปี้

x ข้างในโค้ด จะอยู่แค่ในโค้ดนั้นเท่านั้น

- แต่พารามิเตอร์ของเมธอด มีได้สองแบบคือ เป็น primitive กับเป็น object reference
- ซึ่งคราวนี้ เกิดการเปลี่ยนได้ เช่น

```
public static void tripleValue(Employee m){
   m.raiseSalary(200);
}
```

เมื่อเรียกใช้

```
Employee harry = new Employee(...);
tripleValue(harry);
```

จะเกิดการ copy pointer ไปใช้งาน ดังนั้นจะเกิด pointer ที่ชี้ไปที่ออบเจ็กต์เดียวกัน ดังรูป

การเปลี่ยน salary จาก m ก็จะเป็นการเปลี่ยนจาก harry ไปด้วย

# รูป


raiseSalary จะไปเปลี่ยนข้างในออบเจ็กต์ตัวจริง

- ระวัง เขียนอย่างนี้ไม่เวิร์ค

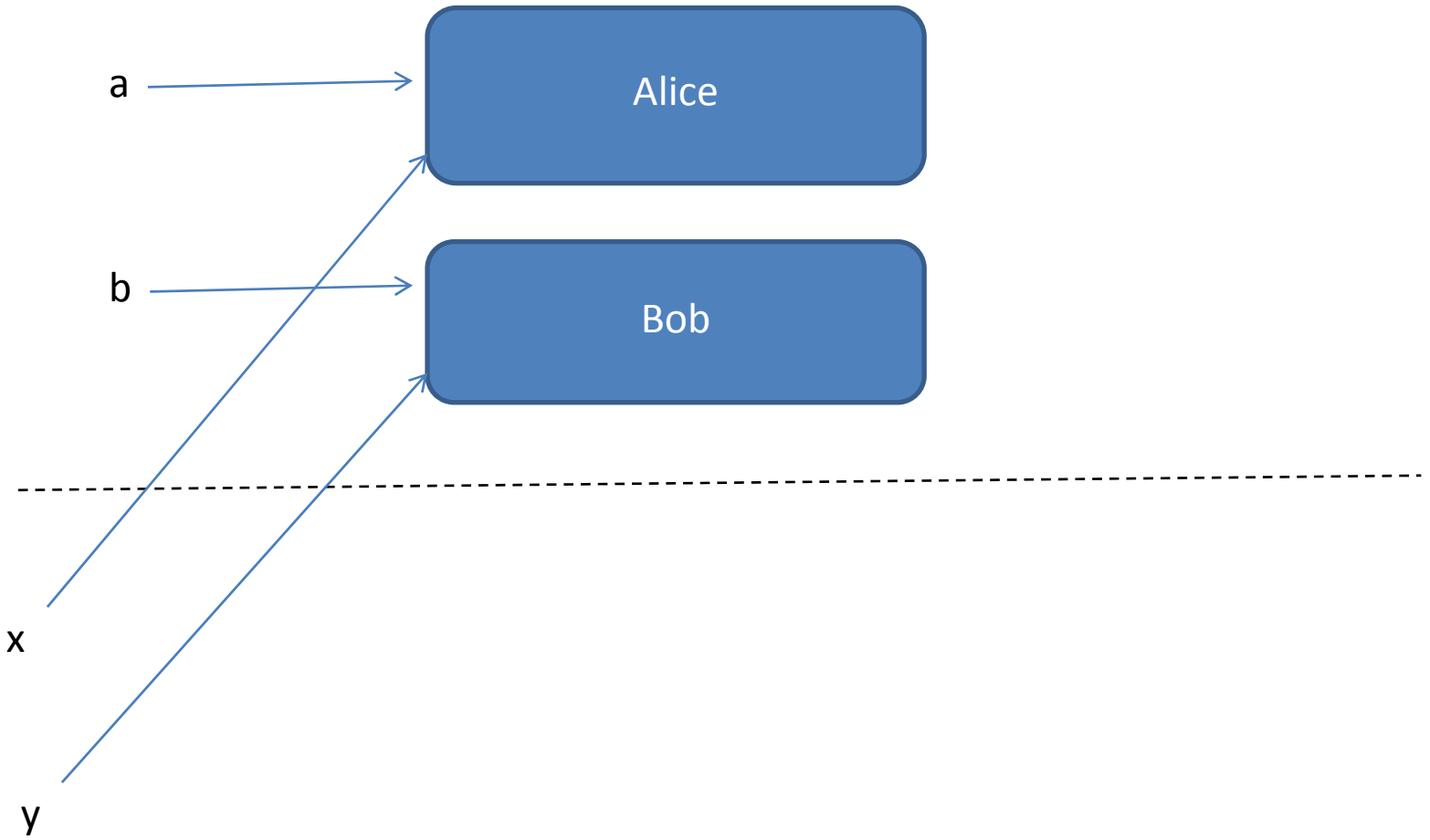
```
public static void swap(Employee x, Employee y)
  {
    Employee temp = x;
    x = y;
    y = temp;
  }
```

แล้วไปเรียกใช้แบบนี้

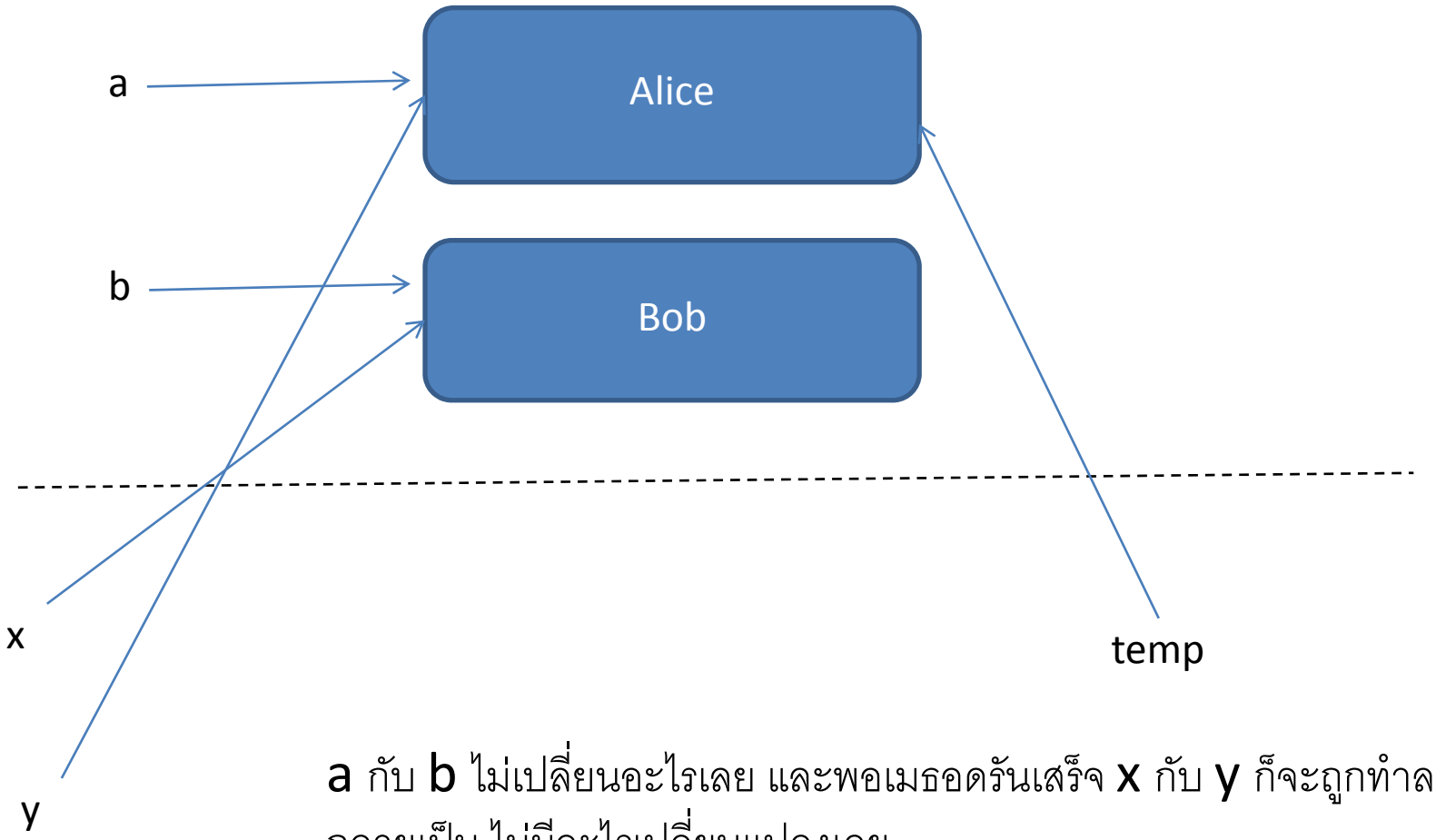
```
Employee a = new Employee("Alice", …);
Employee b = new Employee("Bob", …);
swap(a,b);
```

ตัวที่สลับ คือ pointer ที่เป็นตัวก๊อปปี้ ของทั้ง a และ b
ดังนั้น a กับ b จึงชี้ไปที่เดิมตลอด

# ดูรูป



# เมื่อเมธอดรันแล้ว




a กับ b ไม่เปลี่ยนอะไรเลย และพอเมธอดรันเสร็จ x กับ y ก็จะถูกทำลาย กลายเป็น ไม่มีอะไรเปลี่ยนแปลงเลย

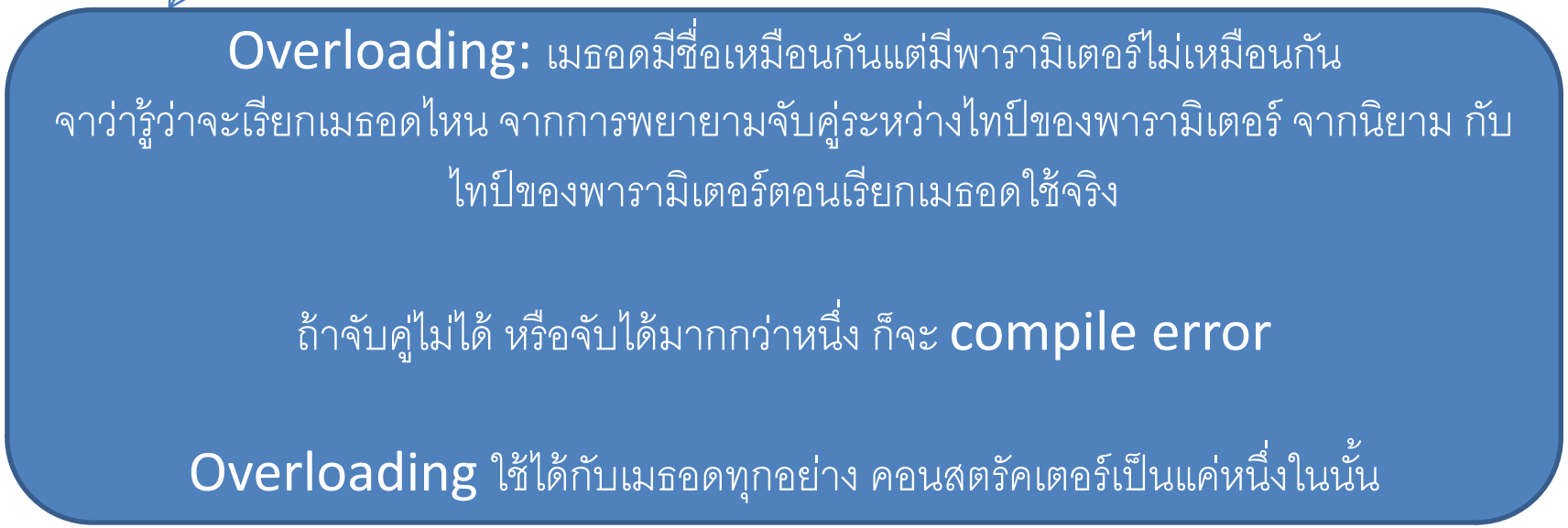
# สรุป เมธอดทำอะไรได้หรือไม่ได้บ้าง

- เปลี่ยนค่าตัวแปร ของพารามิเตอร์ที่เป็น primitive type ไม่ได้
- เปลี่ยนค่าภายใน ของออบเจ็กต์ ได้
- ให้pointer ชี้ไปที่ออบเจ็กต์อื่น ไม่ได้
- ดูตัวอย่างจากไฟล์ ParamTest.java

# Object construction

new GregorianCalendar()

new GregorianCalendar(1999, 11, 31)



- ชื่อเมธอด + พารามิเตอร์ เราเรียกรวมๆว่า signature
- Return type ไม่รวมไปด้วย ดังนั้น
  - ดังนั้นเมธอดที่ต่างกันแค่รีเทิร์นไทป์ จาวาจะนึกว่าเป็นนิยามซ้ำซ้อน
- ถ้าในคอนสตรัคเตอร์ เราไม่ได้เซตค่าฟิลด์ ค่าจะถูกเซตให้เป็นค่า default ของมัน เช่น
  - ตัวเลขจะเป็น 0
  - boolean เป็น false
  - Object reference เป็น null
  - อย่าลืมว่า field ไม่ใช่ local variable
  - และต้องระวัง ถ้าเราทิ้งให้ตัวแปรมีค่า null พอเรียกใช้ทีหลัง จะทำให้ error ได้ ตัวอย่างอยู่หน้าถัดไป

- สมมุติว่า ตัวแปร hireDay ของ Employee ถูกทิ้งให้เป็น null ไว้
- ตอนเราเรียก Calendar.setTime(harry.getHireDay());
  - จะมีการ throw exception ได้ เพราะว่า setTime ดันได้รับ null เป็นพารามิเตอร์ ซึ่งในโค้ดไม่สามารถนำไปใช้งานได้

# Default constructor

- คือ คอนสตรัคเตอร์ ที่ไม่มีพารามิเตอร์
- ถ้าในโปรแกรม เราไม่ได้เขียนคอนสตรัคเตอร์ซักตัว จาวาจะสร้างตัว default ให้เราเอง
  - ซึ่งจะเซ็ตค่าของ instance field ทั้งหมดให้เป็นค่า default
- ถ้าคลาสมีการนิยามคอนสตรัคเตอร์ แต่ว่าไม่มีการนิยาม default constructor มา จาวาจะไม่สร้างตัว default constructor ให้ ดังนั้นก็จะเรียกใช้ตัว default constructor ไม่ได้ด้วย

- ที่นิยามของตัวแปรแต่ละตัว เราสามารถ assign ค่าให้มันได้เลย โดยค่าเหล่านี้จะอยู่ก่อนที่คอนสตรัคเตอร์จะถูกเรียก ตัวอย่างเช่น:

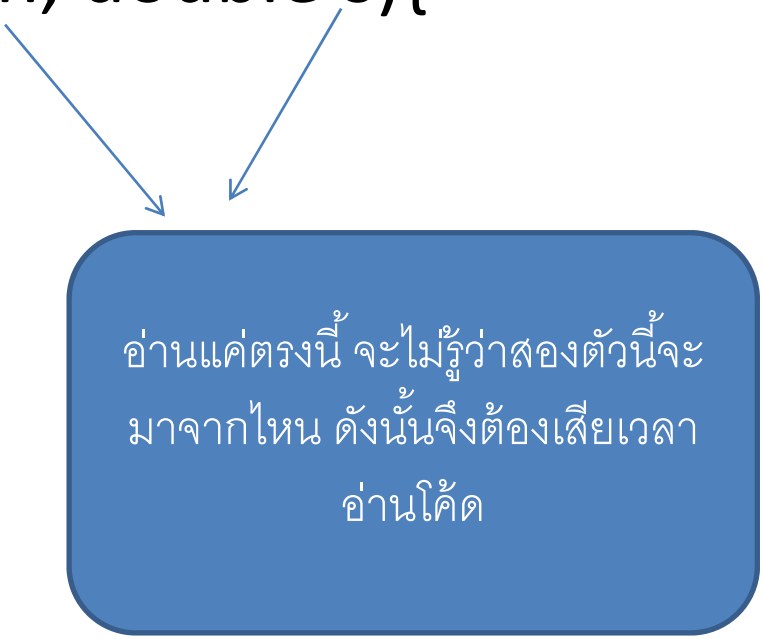
```
class Employee{
…
static int assignID(){
    int r = nextId;
    nextId++;
    return r;
…
private int id = assignId();

}
```

ตรงนี้ไม่ได้อยู่ในคอนสตรัคเตอร์ แต่เป็นตรงนิยามตัวแปรเลย ซึ่งจะถูกรันก่อนที่คอนสตรัคเตอร์จะทำงาน รันเมธอดก็ได้

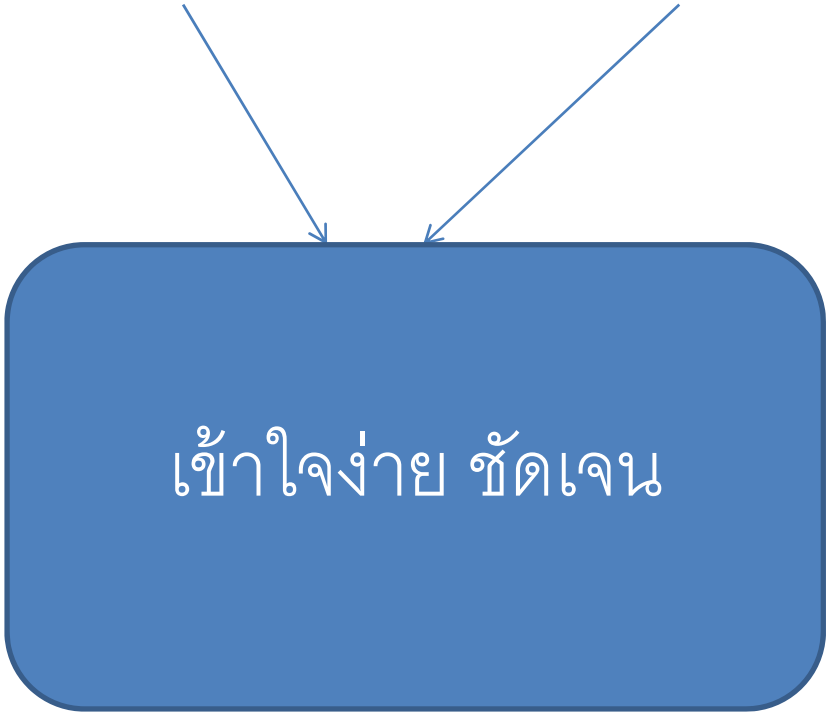
ตัวนี้จะเป็นตัวที่ initialize ค่าที่เราอาจไม่ต้องการมา initialize ในคอนสตรัคเตอร์ทุกตัว

# การตั้งชื่อพารามิเตอร์

```
public Employee(String n, double s){
    name = n;
    salary = s;
}
```



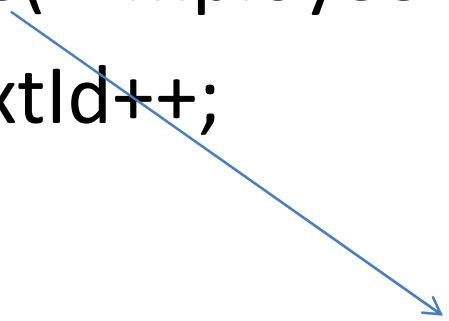
```
public Employee(String aName, double aSalary){
    name = aName;
    salary = aSalary;
}
```



```
public Employee(String name, double salary){
    this.name = name;
    this.salary = salary;
}
```

# การเรียกใช้คอนสตรัคเตอร์อื่นของคลาสเดียวกัน

```
public Employee(double s){
    this(“Employee #” + nextId, s);
    nextId++;
}
```



เรียกใช้คอนสตรัคเตอร์ตัวอื่นของคลาสเดียวกันได้ ดังนั้นถ้าเราเขียนคอนสตรัคเตอร์ตัวพื้นให้ละเอียด ก็สามารถถูกตัวอื่นเรียกใช้ได้หมด จะได้ประหยัดเวลาในการเขียนคอนสตรัคเตอร์ตัวหลังๆ

# Initialization block

```
private int id;
private String name = ""; // instance field initialization
private double salary;

// static initialization block
static
{
   Random generator = new Random();
   // set nextId to a random number between 0 and 9999
   nextId = generator.nextInt(10000);
}

// object initialization block
{
   id = nextId;
   nextId++;
}
```

อันนี้เราอยู่ในคลาส Employee นะ

static ก็ initialize ได้ตรงนี้เหมือนกัน

ใช้ initialize static variable แต่ตามปกติใช้กับการ initialize ที่ค่อนข้างซับซ้อนเท่านั้น

Execute เมื่อออบเจ็กต์ของคลาสนี้ถูกสร้าง ซึ่งตัวโค้ดนี้จะถูกรันก่อนคอนสตรัคเตอร์

- ลำดับการรันจะเป็นดังนี้
  - ตอนแรกค่าทุกค่าเป็น default
  - Field initializer และ initialization block จะถูกรัน ตามลำดับบรรทัดในโค้ดว่าอันไหนจะมาก่อนมาหลัง (static initialization ก็ทำตามลำดับนี้เหมือนกัน)
  - ถ้าบรรทัดแรกของคอนสตรัคเตอร์ เป็นการเรียกคอนสตรัคเตอร์อีกตัว คอนสตรัคเตอร์ตัวที่ถูกเรียกจะโดนรันก่อน
  - คอนสตรัคเตอร์
- มาดูตัวอย่างโปรแกรมที่ใช้คอนสตรัคเตอร์กัน
  ConstructorTest.java

# packages

- กลุ่มของคลาสที่ใช้ด้วยกัน เรียกว่า package
- ในจาวาเองก็มี
  - java.lang
  - java.util
  - java.net
- เราจัดแพคเกจเองได้ จัดเหมือนกับการจัดโฟลเดอร์ที่เก็บไฟล์
- ที่มีมาให้เราแล้ว จะอยู่ใน java หรือ javax

- ทำไมต้องมี package
  - เหตุผลสำคัญคือ จะได้ไม่มีปัญหาการที่คลาสชื่อซ้ำกัน
  - สมมุติมีคนคิดคลาส Employee มาสองคน ถ้าต่างคนต่างเอาเข้า package คนละอัน จาวาก็จะรู้ว่าเป็นคนละคลาสกัน ไม่สับสนว่ามีชื่อซ้ำ
  - แต่ทั้งนี้ ชื่อ package ก็ต้องตั้งให้ต่างกันด้วย
    - เพื่อที่จะทำให้ได้ชื่อเฉพาะมากๆ sun ให้ใช้ชื่อเว็บไซต์กลับหลัง เช่น com.vishnu
    - จากนั้นก็สามารถทำ subpackage ต่อได้ เช่น com.vishnu.progmeth
- สำหรับจาวาเอง java.util กับ java.util.jar ถือว่าเป็นสองแพ็กเกจที่ไม่มีอะไรเกี่ยวข้องกันเลย

# การ import

- คลาสหนึ่งๆ สามารถใช้งานคลาสที่อยู่ในแพ็กเกจเดียวกัน และคลาสในแพ็กเกจอื่นที่เป็น public
- แต่ว่า คลาสในแพ็กเกจอื่น จะมองไม่เห็น ต้องมีการบอก โดยมีสองวิธี
  - ใส่ชื่อแพ็กเกจข้างหน้าชื่อคลาสทุกคลาสที่เอามาจากแพ็กเกจอื่น เช่น

```
Java.util.Date today = new java.util.Date();
```

    - แต่วิธีนี้มันทำให้เขียนยืดยาว
  - วิธีที่สองคือการ import
    - เขียน import ด้านบนของไฟล์ ต่อจากนิยามpackage ที่เรานิยาม package ของคลาส
    - import java.util.*; จะเป็นการอิมพอร์ตทุกไฟล์จากแพ็กเกจนั้น

- แต่ว่า ใช้งานแบบนี้ไม่ได้

```
import java.*;
import java.*.*;
```

- และที่ต้องระวังอีกอย่างคือ ถ้าสองแพ็กเกจดันมีคลาสชื่อเหมือนกัน เช่น java.util.* กับ java.sql.* มีคลาส Date ทั้งคู่
  - ถ้าเราอิมพอร์ตทั้งคู่ เราจะใช้ Date ไม่ได้
  - แก้โดย import java.util.Date; เขียนอันนี้ลงไปเพิ่ม
  - หรือ ถ้าเกิดต้องใช้ Date จากทั้งสองคลาสจริงๆ ก็ต้องเขียนชื่อคลาสที่รวมชื่อแพ็กเกจด้วยตลอด

# Static import

import static java.lang.System.*;

จะทำให้เรียกใช้ static เมธอด และ static field จากคลาส System ได้ โดยไม่ต้องเอ่ยชื่อคลาสก่อน เช่น

out.println(“…”);

ใช้ได้ดีกับคลาส Math และการใช้งาน constant เพราะใช้ได้อย่างเป็นธรรมชาติมากขึ้น

# การนิยามว่าคลาสเรา อยู่ในแพ็กเกจอะไร

- ใส่ชื่อแพ็กเกจ ด้านบนสุดของ source file ตัวอย่างเช่น

```
package com.vishnu.progmeth;
public class Employee{……
}
```

ถ้าเราไม่ใส่ชื่อแพ็กเกจให้คลาสเรา จาวาจะถือว่า คลาสนั้นอยู่ใน

default package

# การเก็บไฟล์ในโฟลเดอร์

- เมื่อเรานิยามแพ็กเกจไปแล้ว การเก็บไฟล์ในโฟลเดอร์ของคอมพิวเตอร์เรา ก็ต้องทำตามชื่อของแพ็กเกจ เช่น
- ไฟล์ทั้งหมดที่อยู่ในแพ็กเกจ com.vishnu.progmeth จะต้องถูกเก็บในโฟลเดอร์ com\vishnu\progmeth ซึ่งต้องอยู่ใน base directory อีกทีหนึ่ง
- ต้องเก็บให้ถูก เพราะว่า .class ไฟล์จะถูกเก็บไว้ตามนี้เหมือนกัน ถ้าเกิด source file ไม่ได้อยู่ตามโฟลเดอร์นี้ จะคอมไพล์ได้แต่รันไม่ได้ เพราะการหา .class ไฟล์ จะหาจากโฟลเดอร์ที่เรานิยามแพ็กเกจ

# Package scope

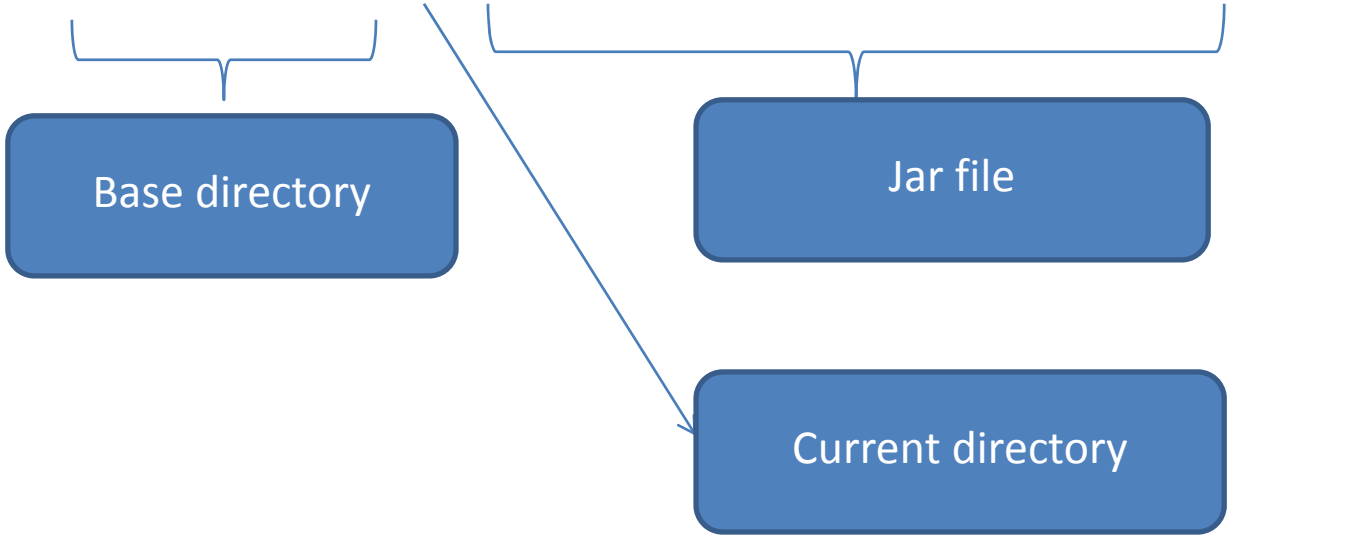
- ถ้าเราไม่ได้บอกว่า คลาส ตัวแปร หรือเมธอด เป็น public หรือ private จาวาจะถือว่า เมธอดอื่นๆ ที่อยู่ในแพ็กเกจเดียวกัน จะสามารถมองเห็นคลาส ตัวแปร หรือเมธอดนั้นได้
- เราสามารถ seal package ได้
  - เป็นการป้องกัน ไม่ให้คนอื่นมาเติมไฟล์ลงใน แพ็กเกจของเรา
  - ทำโดยการ seal jar (จะเห็นว่า eclipse จะมีคำสั่งพวกนี้ในตัว)

# Class path

- Jar file คือ zip ไฟล์ ที่เก็บคลาสไฟล์ต่างๆ และเก็บ subdirectory ไว้
- เราต้องตั้งค่า classpath

ใช้ * ได้

```
c:\classdir;.;c:\archives\archive.jar
```



- เวลาจาวาจะหาคลาส มันจะดูที่ตัว library ของมันก่อน
- แล้วค่อยมาดู class path
- สมมุติเรามีคลาสคลาสหนึ่งที่อิมพอร์ตดังนี้

```
import java.util.*;
import com.vishnu.progmeth.*;
```

ถ้าในไฟล์นี้มีการใช้งาน คลาส Employee จาวาจะพยายามไปดู

Java.lang.Employee, java.util.Employee, com.vishnu.progmeth.Employee และ Employee ใน current package โดยจะหาแต่ละไฟล์พวกนี้ จากทุกๆ โฟลเดอร์ที่นิยามใน class path